Práctica 1: programación funcional en Java

Nuevas tecnologías de la programación

Contenido:

1 Objetivos

En esta primera práctica se ponen en juego los conocimientos adquiridos sobre programación funcional en Java. Para ello se consideran dos problemas de ámbitos diferentes: procesamiento de información y búsqueda en árboles mediante el uso de la recursividad. Nota: en todos los fragmentos de código se han eliminado todos los acentos para evitar problemas con la codificación de caracteres.

2 Problema de procesamiento de información

Se trata de implementar una aplicación que permita a una empresa de ventas examinar los datos de sus empleados (comerciales) y su asignación a dos sectores geográficos diferentes (cada uno con tres rutas distintas). La empresa cuenta con unos archivos que contienen errores y pretende hacerse una idea exacta de los problemas presentes en los datos para repararlos en la medida de lo posible.

Los datos de empleados se encuentran en un archivo denominado **datos.txt** que contiene una línea de descripción para cada empleado, con los siguientes datos (separados por comas): dni, nombre, apellidos y dirección de correo electrónico. La empresa sabe que hay algún error con los dnis y correos (en ambos casos puede haber repeticiones).

También existen ficheros auxiliares que indican la asignación de los empleados a los dos sectores geográficos en que trabaja la empresa. Cada sector geográfico cuenta con tres rutas diferentes. De esta forma, los archivos de asignación a sectores y rutas son:

- **asignacionSECTOR1.txt**: dnis de empleados asignados al primer sector. Todos los archivos de asignación contienen una primera línea que indica el sector o ruta de que se trata, una linea en blanco y un conjunto de líneas (una por empleado) con los dnis
- **asignacionSECTOR2.txt**: dnis de empleados asignados al segundo sector
- **asignacionRUTA1.txt**: dnis de empleados asignados a la primera ruta (de un sector o de otro, ya que ambos sectores cuentan con el mismo número de rutas)
- **asignacionRUTA2.txt**: empleados asignados a la segunda ruta
- **asignacionRUTA3.txt**: empleados asignados a la tercera ruta

Se sabe que estos archivos están incompletos y que hay empleados sin asignar a sector y/o ruta. De esta forma, se trata de implementar un sistema de gestión de datos para analizar la información de empleados y su asignación a sectores y rutas. Nos indican que los identificadores de sectores y rutas deben proporcionarse como los siguientes enumerados:

```
1  /**
2   * Enumerado para representar los sectores de actuacion de
3   * los comerciales
4   */
5  public enum Sector {
6     SECTOR1, SECTOR2, NOSECTOR
7  }
8
9  /**
10  * Enumerado para representar los códigos de las rutas
11  * de la empresa
12  */
13 public enum Ruta {
14     RUTA1, RUTA2, RUTA3, NORUTA
15 }
```

Los identificadores de división y departamentos que incluyen **NO** como prefijo se utilizan para indicar no asignación (ya sea a sector, a ruta o ambos). Estos dos enumerados están ya disponibles en el código ofrecido como punto de partida (bajo la carpeta **código de partida**). Se incluye a continuación, a modo de ejemplo, parte del contenido de los archivos con los datos de empleados (**datos.txt**) y de asignaciones respectivamente:

```
00848497, CARMELO, ARROYO PEREZ, caarpe@acme.com
36702450, MIREN, MARQUEZ PIZARRO, mimapi@acme.com
33545726, MELCHOR, CUADRADO BERMUDEZ, mecube@acme.com
18131803, MOHAMMED, CONDE PINEDA, mocopi@acme.com
94284911, UNAI, HUERTAS ESCUDERO, unhues@acme.com
98242276, CARLOS ANTONIO, COLLADO CABEZAS, cacoca@acme.com
35626728, DEBORA, ANTON CABELLO, deanca@acme.com
98138706, CARIDAD, CARDONA SALGUERO, cacasa@acme.com
22468058, VICTORINA, OLMEDO AMAYA, violam@acme.com
99504054, MARIA ALICIA, ROBLES RIOS, marori@acme.com
```

```
SECTOR1

36197438
86508453
03048427
42809822
00357527
78556255
46322393
70112778
50713222
21823519
12843257
```

Estos archivos también se ofrecen en la carpeta llamada **datos** de la práctica. La pieza básica de este sistema es la clase **Empleado** ya que los datos de los archivos deben usarse para crear objetos de esta clase y manipularlos de forma conveniente mediante el uso de flujos y el paradigma de programación funcional. El código de esta clase se ofrece en la carpeta llamada **código de partida**. Su contenido es el siguiente:

```java
package listado;

import java.util.HashMap;
import java.util.List;
import java.util.Random;
import java.util.regex.Pattern;
import java.util.stream.Collectors;

/**
 * Clase para gestionar empleados, dnis y grupos de practicas
```

```
11  * @author mgomez
12  */
13 public class Empleado {
14     /**
15      * Dato miembro para almacenar dni
16      */
17     private String dni;
18 
19     /**
20      * Datos miembro para almacenar apellido1, apellido2 y nombre
21      */
22     private String apellidos, nombre;
23 
24     /**
25      * Dato miembro para almacenar el correo
26      */
27     private String correo;
28 
29     /**
30      * Dato miembro para almacenar el sector asignado
31      */
32     private Sector sector;
33 
34     /**
35      * Dato miembro para almacenar la asignacion a ruta
36      */
37     private Ruta ruta;
38 
39     /**
40      * Dato miembro para almacenar un generador de numeros aleatorios
41      */
42     private static Random generador = new Random();
43 
44     /**
45      * Constructor de la clase
46      * @param dni
47      * @param apellidos
48      * @param nombre
49      * @param correo
50      */
51     public Empleado(String dni, String nombre, String apellidos, String correo) {
52         this.dni = dni;
53         this.nombre = nombre;
54         this.apellidos = apellidos;
55         this.correo = correo;
56 
57         // Por defecto no sde asigna ni sector ni ruta (representado
58         // con el valor NA en cada enumerado)
59         sector = Sector.NOSECTOR;
60         ruta = Ruta.NORUTA;
61     }
62 
63    /**
64     * Asigna el dni
```

```
65      * @param dni
66      */
67     public void asignarDni(String dni){
68        this.dni=dni;
69     }
70
71     /**
72      * Metodo para asignar el sector a un empleado
73      * @param sector
74      */
75     public void asignarSector(Sector sector){
76        this.sector =sector;
77     }
78
79     /**
80      * Metodo para asignar la ruta a un empleado
81      * @param ruta
82      */
83     public void asignarRuta(Ruta ruta){
84        this.ruta =ruta;
85     }
86
87    /**
88     * Modifica el dni actual generando un numero aleatorio
89     * entre 0 y 99
90     */
91    public void asignarDniAleatorio(){
92        System.out.println("A asignar: "+dni+generador.nextInt(100));
93        dni=dni+generador.nextInt(100);
94     }
95
96     /**
97      * Metodo que permite asignar el sector de forma aleatoria
98      */
99     public void asignarSectorAleatorio() {
100        // Se genera un numero aleatorio para el sector
101        int sectorAleatorio = generador.nextInt(Sector.values().length);
102
103        // Se asigna
104        sector =(Sector.values())[sectorAleatorio];
105     }
106
107     /**
108      * Metodo para asignar la ruta de forma aleatoria
109      */
110     public void asignarRutaAleatorio() {
111        int rutaAleatorio = generador.nextInt(Ruta.values().length);
112        // Se asigna el ruta
113        ruta =(Ruta.values())[rutaAleatorio];
114     }
115
116     public void generarCorreoCompleto(){
117        correo=nombre+apellidos+"@acme";
118        // Se eliminan espacios en blanco
```

```
119         correo.replaceAll("\\s+","");
120     }
121 
122     /**
123      * Metodo para obtener el dni
124      *
125      * @return
126      */
127     public String obtenerDni() {
128         return dni;
129     }
130 
131     /**
132      * Metodo para obtener el correo
133      */
134     public String obtenerCorreo(){
135         return correo;
136     }
137 
138     /**
139      * Recupera la dvision
140      * @return
141      */
142     public Sector obtenerSector(){
143         return sector;
144     }
145     /**
146      * Metodo que devuelve el ruta
147      * @return
148      */
149     public Ruta obtenerRuta() {
150         return ruta;
151     }
152 
153     /**
154      * Metodo toString
155      * @return
156      */
157     public String toString() {
158         String info = "-----------------------------------------------\n";
159         info = info + "DNI: " + dni + "  " + nombre + "  " + apellidos + "  " + corre
160         // Se muestra tambien informacion sobre sector y ruta
161         info = info + " " + sector.toString() + " : " + ruta.toString();
162         info = info + "\n-----------------------------------------------\n";
163         return info;
164     }
165 
166     /**
167      * Metodo toLine genera una linea con la informacion de los empleados,
168      * separada por comas
169      *
170      * @return
171      */
172     public String generarLineaSimple() {
```

```
173         String info = dni + ",  " + nombre + ",  " + apellidos + ",  " + correo;
174         return info;
175     }
176 
177     /**
178      * Metodo para generar los datos de un empleado, con dni y
179      * ruta
180      * @return
181      */
182     public String generarLineaDniRuta() {
183         String info = dni + " " + ruta.toString();
184         return info;
185     }
186 
187     /**
188      * Metodo para generar los datos de un empleado, con dni y
189      * sector
190      * @return
191      */
192     public String generarLineaDniSector() {
193         String info = dni + " " + sector.toString();
194         return info;
195     }
196 
197 
198     /**
199      * Metodo que devuelve un valor booleano indicando si un empleado
200      * pertenece a un sector
201      * @param sector
202      * @return
203      */
204     public boolean perteneceSector(Sector sector) {
205         // Se devuelve el resultado de la comparacion
206         return(this.sector == sector);
207     }
208 }
```

Esta clase contiene datos miembro para almacenar **dni**, **apellidos**, **nombre**, **correo**, **sector** y **ruta**. Posee además un dato estático para referenciar a un objeto de la clase **Random** para permitir la generación de números aleatorios. Con respecto a los métodos ofrece la siguiente funcionalidad:

- constructor recibiendo como argumento dni, nombre, apellidos y correo. Este será el constructor usado al leer el archivo **datos.txt**. Se aprecia que el constructor asocia los valores **Sector.NOSECTOR** y **Ruta.NORUTA** a los datos miembro correspondientes
- métodos para asignar dni, sector y ruta
- métodos para asignar sector y ruta de forma aleatoria (estos métodos se han usado para generar los datos, pero seguramente no tengáis que usarlos). También existe un método de asignación de dni de forma aleatoria que será usado, por comodidad, en el procedimiento de reparación de dnis repetidos. Este método agrega al valor actual de dni una cantidad aleatoria (entre 0 y 99)

- método para generar un correo completo. Este método permitirá reparar aquellos objetos de la clase en que se detecte repetición de correo electrónico. El procedimiento hace que el correo se forme a partir de nombre y apellidos (eliminando posibles espacios en blanco) para minimizar la probabilidad de coincidencia. Inicialmente los correos se asignan considerando las iniciales de nombre y apellidos (por lo que no es raro que haya correos repetidos)
- métodos de acceso a los valores de los datos miembro
- método **toString**, que devuelve una cadena con la información del objeto
- para algunos listados simples puede resultar interesante contar con métodos que generen cadenas con parte de los datos o con otro formato. Este es el objetivo de los siguientes métodos: **generarLineaSimple**, **generarLineaDniRuta**, **generarLineaDniSector**
- método para indicar si el empleado pertenece a un sector pasado como argumento

2.1 Funcionalidad básica a aportar

A partir de estos datos el sistema debe aportar la funcionalidad indicada a continuación.

2.1.1 Generación de colección de objetos de la clase Empleado

Esta colección debe permitir almacenar los datos relevantes de cada empleado (dni, nombre, apellidos, correo, sector y ruta) y se obtiene mediante el procesamiento del contenido de los archivos de datos proporcionados (datos de empleados y asignaciones).

La colección puede soportarse en una clase llamada **ListadoEmpleados**, que aportará dos datos miembro básicos (además de otros adicionales que podáis necesitar), indicados a continuación:

```
1   /**
2    * Dato miembro para almacenar a los empleados tal y como se encuentran
3    * en el archivo de datos.txt
4    */
5   private List<Empleado> listadoArchivo;
6
7   /**
8    * Dato miembro para almacenar a los empleados como mapa con pares
9    * (una vez reparados los datos leidos del archivo)
10   * <dni - empleado>
11   */
12  private Map<String, Empleado> listado;
```

El constructor de la clase recibirá como argumento la ruta del archivo de datos de empleados y debe construir y almacenar todos los objetos de la clase **Empleado** que sean necesarios en el dato miembro **listadoArchivo**. Los objetos se mantienen en esta estructura hasta el momento en que sean reparados, como se indicará a continuación. Tras reparar los datos se generará el diccionario referenciado por el dato miembro **listado**.

Por tanto, el objetivo básico del constructor de la clase será procesar el contenido del archivo **datos.txt**, de forma que al final de su trabajo el dato miembro **listadoArchivo** tenga almacenados todos los objetos de la clase **Empleado** (uno por cada línea de datos del archivo). Se recomienda crear un constructor en la clase **Empleado** que reciba como argumento una línea de datos del archivo y tras su análisis extraiga la información básica necesaria para llamar dar valor a los datos miembro. Con esta idea, el esquema de funcionamiento del constructor podría ser el siguiente:

```
1   creacion del diccionario sobre el dato miembro listado
2   obtener las lineas del archivo datos.txt
3   para cada linea
4      - llamar al constructor de Empleado, pasando la linea
5        como argumento; se genera asi un nuevo objeto de la clase
6        Empleado
7   almacenar el resultado del procesamiento sobre listadoArchivo
```

El constructor de la clase **Empleado** mencionado con anterioridad de realizar la siguiente tarea:

- crea un patrón para considerar la coma como separador
- se divide la cadena con la información completa de una línea en las cadenas con las unidades informativas (dni, nombre, apellidos y correo); las unidades informativas se almacenan en una lista
- los elementos almacenados en la lista se usarán para crear y devolver un objeto de la clase **Empleado**

2.2 Funcionalidad adicional (I)

Toda la funcionalidad se probará mediante casos de prueba (mediante **junit**). Cada caso de prueba debe indicar claramente el método asociado.

2.2.1 Conteo del número de empleados descritos en datos.txt

La clase **ListadoEmpleados** contará con un método denominado **obtenerNumeroEmpleadosArchivo**. Este método devolverá el tamaño de la lista asociada al dato miembro **listadoArchivo** (no hace falta hacer uso de las facilidades de programación funcional).

Y la prueba unitaria para este método debe verificar que el número de empleados descritos en el archivo es 5000.

2.2.2 Determinar si hay empleados con dni repetido

Se implementará el método **hayDnisRepetidosArchivo** que devuelva el valor **true** en caso de dnis repetidos y **false** en caso contrario. Aquí ya sí debe usarse el procesamiento de datos de **listadoArchivo** mediante flujos y programación funcional. El caso de prueba asociado debe comprobar que la llamada a este método devuelve **true**.

2.2.3 Obtención de datos sobre empleados con dnis repetidos

El método **obtenerDnisRepetidosArchivo** tendrá la siguiente declaración:

```
1 public Map<String, List<Empleado>> obtenerDnisRepetidosArchivo()
```

Como se observa, el método devuelve un diccionario de pares dni (cadena) y lista de empleados con dicho dni (valor). Para probar la funcionalidad de este método se dotará a la clase del método **contarDnisRepetidos**, que trabajará con la salida del método anterior y contabilizará el total de empleados con problemas debidos a la repetición de dnis. Su declaración es:

```
1 public int contarEmpleadosDnisRepetidos()
```

La prueba unitaria sobre este método debe indicar que hay 4 empleados afectados por repetición de dnis.

2.2.4 Reparar errores en dnis

El método

```
1 public void repararDnisRepetidos(Map<String, List<Empleado>>
  ↪ listaRepeticion)
```

se encargará de hacer el procesamiento de los datos de empleados afectados por la repetición de dnis. La solución pasaría, obviamente, por preguntar al usuario del sistema por el valor de dni a asignar. Sin embargo, para que la prueba del sistema sea más cómoda el proceso de reparación consistirá en asignar un valor aleatorio de dni, usando el método de la clase **Empleado** que proporciona esta funcionalidad. Al terminar la asignación aleatoria debe comprobarse de nuevo la presencia de dnis repetidos.

2.2.5 Comprobación de correos repetidos

La clase **ListaEmpleados** incorporará el método **hayCorreosRepetidosArchivo**, con la declaración:

```
1 public boolean hayCorreosRepetidosArchivo()
```

Este método determina un valor booleano indicando la existencia de correos repetidos. La prueba de este método debe verificar que realmente se da este problema.

2.2.6 Obtención de lista de datos de correos repetidos

El método **obtenerCorreosRepetidosArchivo** proporciona los datos de los empleados con problemas en el correo:

```
1 public Map<String, List<Empleado>> obtenerCorreosRepetidosArchivo()
```

El diccionario devuelto contiene pares de correo (cadena, como valor) y lista de empleados (valor) que comparten dicha cadena como correo. Al igual que en el caso de los dnis, debe aportarse un método que cuente los empleados con correo repetido. Este método se denominará **contarCorreosRepetidos**. Haciendo uso de este método debe comprobarse que existen 315 empleados con este problema.

2.2.7 Reparación de correos repetidos

Esta funcionalidad será aportada por el método

```
1 public void repararCorreosRepetidos(Map<String, List<Empleado>>
  ↪  listaRepeticiones)
```

que recibe como argumento el diccionario con los empleados con problemas de correo (obtenidos previamente mediante el método **obtenerCorreosRepetidosArchivo**), e iterará sobre esta colección para conseguir llamar sobre cada objeto el método **generarCorreoCompleto**. Tras reparar los errores en el correo debe comprobarse de nuevo si hay o no repetición en los correos.

2.2.8 Generación de colección de empleados

Una vez realizada la reparación de todos los problemas (tanto de dnis como de correos), se procede a almacenar la información en el dato miembro **listado**. Este trabajo es la responsabilidad del método:

```
1 public void validarListaArchivo()
```

3 Solución al problema de las 8 reinas

Este problema tiene una naturaleza completamente diferente al anterior pero también puede tratarse, al menos parcialmente, bajo el enfoque de programación funcional. También permite poner en juego el concepto de recursividad (básico en programación funcional).

3.1 Representación de un tablero

Los tableros usados podrán ser de cualquier dimensión. Se representarán mediante la clase **Tablero**, que tendrá dos datos miembro:

- **dimension**, de tipo entero, para indicar el número de filas y columnas del tablero (siempre se consideran tableros cuadrados).
- **contenido**, de tipo **ArrayList<Celda>**, de forma que sólo se almacenan en realidad aquellas celdas en las que se ha ubicado una reina.

Como se aprecia en la definición del dato miembro **contenido**, se usa la clase **Celda** para representar posiciones del tablero. Esta clase tiene como datos miembro **fila** y **columna** (para identificarla de forma única).

3.2 Métodos de la clase Celda

La clase **Celda** es sencilla y debería bastar con disponer de métodos que permitan:

- obtener los valores de los datos miembro **fila** y **columna**
- determinar si una celda está en conflicto con otra (puede ser un método estático o pertenecer a la clase). La comprobación del conflicto es simple: que no haya coincidencia en fila ni columna y que no estén ubicadas en la misma diagonal (la diferencia en valor absoluto entre filas y columnas no debe ser igual).

3.3 Métodos de la clase Tablero

Esta clase debe contar con los siguientes métodos:

- constructor, que recibe como argumento la dimensión deseada. Debe asignar el valor del dato miembro correspondiente y crear un almacén vacío de celdas, que será referenciado por el dato miembro **contenido**.
- **ponerReina**, recibiendo como argumentos el número de fila y columna donde se ubicará. Este método creará un objeto de la clase **Celda** con los datos indicados de fila y columna y lo almacenará en el almacén de celdas. No se realiza ninguna comprobación acerca de la validez de la posición indicada.
- **posicionSegura**: recibe como argumento un objeto de la clase **Celda** y comprueba si está en conflicto con algunas de las celdas ocupadas del tablero. Este método debe implementarse usando las posibilidades de la programación funcional. Devuelve un valor booleano con el resultado de la comparación.
- **toString**, que devuelve una cadena que permite visualizar el contenido del tablero. Por ejemplo, pueden usarse el carácter X para posiciones vacías (aquellas posiciones que no tienen celda asociada en el contenido) y R para aquellas que sí tienen reina.

3.4 Búsqueda de soluciones

La búsqueda de soluciones depende de una clase adicional llamada **Buscador**. A modo de ejemplo (podéis diseñar la clase como deseéis) podría contar con la siguiente funcionalidad:

- dato miembro **dimension**: permite crear los tableros necesarios para representar las posibles soluciones del problema. Este dato miembro se inicializará en el constructor de la clase.
- método **ubicarReina**, con la siguiente declaración:

```
public ArrayList<Tablero> ubicarReina(int fila)
```

Este método recibe como argumento la fila con la que debe trabajar y su funcionamiento se puede esquematizar de la siguiente forma:

```
caso base: si la fila es -1, entonces crear un tablero vacio y
almacenarlo en una lista de tableros (que solo tendra este elemento)
caso inductivo:
    recoger en soluciones la salida de otra llamada a ubicarReina,
    pero pasando como argumento fila-1 (es decir, el metodo se basa
    en que ya estan ubicadas las reinas en filas inferiores y
    ↪ procede
    a agregar otra reina mas)

    para cada una de las soluciones (tableros con reinas hasta
    ↪ fila-1):
        para cada columna:
            crear celda para (fila, columna)
            si la celda es segura en el tablero que representa la
            ↪ solucion
            considerada, ubicar la reina en el tablero y almacenarla
            ↪ para
            su devolucion
        fin para
    fin para

    devolver lista con todas las soluciones generadas
```

el bloque central de procesamiento de las soluciones devueltas por **ubicarReina** debe tratarse mediante programación funcional.

- para permitir una llamada simple al método de resolución, se agrega un método **resolver**, sin argumentos, que directamente produce una llamada a **ubicarReina(dimension-1)**, lo que desencadena el procedimiento de solución.

4 Entrega de la práctica

Al final de la realización de la práctica se entregará un archivo comprimido con el contenido completo de la práctica, tal y como se integra en el proyecto con el entorno de desarrollo que hayáis usado. Se incluirá también un pequeño documento indicando el entorno de desarrollo y una breve valoración de la práctica (si los conceptos vistos son novedosos, si os ha parecido de interés, problemas encontrados, etc) en tres o cuatro líneas. La fecha de entrega se fijará más adelante y la entrega se realizará mediante la plataforma PRADO.