Práctica 1: programación funcional en Java (II)

Nuevas tecnologías de la programación

Contenido:

1 Asignación a sectores y rutas

Se trata ahora de procesar los archivos correspondientes a la asignación de empleados a sectores y rutas, mediante el tratamiento de la información almacenada en los archivos:

- **asignacionSECTOR1.txt**
- **asignacionSECTOR2.txt**
- **asignacionRUTA1.txt**
- **asignacionRUTA2.txt**
- **asignacionRUTA3.txt**

Los métodos encargados del procesamiento de esta información serán **cargarArchivoAsignacionSector** y **cargarArchivoAsignacionRuta** respectivamente. Estos métodos reciben como argumento el nombre del archivo a procesar y deben devolver el número de errores obtenidos como consecuencia de la asignación. Se recomienda que estos métodos llamen a los métodos auxiliares **procesarAsignacionSector** y **procesarAsignacionRuta**. Estos métodos recibirán como argumento la cadena con el contenido de una línea del archivo (un dni), el patrón usado para partir la cadena en palabras (aunque sólo hay una en principio) y el sector o ruta a los que se hará la asignación. Estos métodos devolverán **true** si el dni se corresponde con el de algún empleado almacenado en el diccionario referenciado por el dato miembro **listado** y **false** en caso contrario.

La clase **ListadoEmpleados** dispondrá de un método que permita obtener los contadores de empleados asignados a cada sector y ruta: **obtenerContadoresSectorRuta**. Este método debe devolver un diccionario del tipo:

```
1 Map<Sector, Map<Ruta, Long>>
```

El método indicado en el punto anterior puede apoyarse en la existencia de un método auxiliar llamado **obtenerContadoresRuta**, que recibe como argumento un sector (uno de los posibles valores del enumerado) y devuelve los contadores de empleados asignados a cada ruta. De esta manera, su declaración sería:

```
1 public Map<Ruta, Long> obtenerContadoresRuta(Sector sector)
```

Aunque se ha indicado que es un método auxiliar para la obtención de los contadores globales, es interesante ofrecerlo como método público, porque puede usarse así de forma independiente para conocer cómo se reparten los empleados de un determinado sector.

También se implementará el método **obtenerContadoresSectores** que devolverá un array de valores de tipo **long** con el número de empleados asignados a cada sector. Internamente este método puede partir de la información aportada por **obtenerContadoresSectorRuta** y procesarla de forma conveniente para acumular los trabajadores asignados a cada posible valor del enumerado de ruta.

También se dotará a la clase de los métodos necesarios para buscar los empleados con algún problema en la asignación de sectores y rutas. En concreto se implementarán los métodos

- $List < Empleado > buscarEmpleadosSinSectorSinRuta()$: localiza todos los empleados para los que no hay asignación de sector ni ruta (en el dato miembro **sector** tienen asignado el valor **NOSECTOR** y en el dato miembro **ruta** el valor **NORUTA**)
- $List < Empleado > buscarEmpleadosSinRuta(Sector)$: devuelve todos los empleados para los que no hay asignación de ruta (en el dato miembro **ruta** tienen asignado el valor **NORUTA**) teniendo asignado el sector pasado como argumento
- $List < Empleado > buscarEmpleadosConSectorSinRuta()$: devuelve una lista con empleados con asignación de sector pero no de ruta. Este método debe hacer uso del método indicado en el punto anterior
- $List < Empleado > buscarEmpleadosSinSector(Rutaruta)$: devuelve una lista con empleados sin asignación de sector pero asignados a la ruta pasada como argumento
- $List < Empleado > buscarEmpleadosSinSectorConRuta()$: devuelve una lista con empleados sin asignación de sector pero con asignación de ruta. Este método debe hacer uso del método indicado en el punto anterior

2 Casos de prueba

Se dotará al sistema de los casos de prueba necesarios para comprobar que funcionan de forma correcta, mediante **Junit**. Para poder integrar esta herramienta en nuestro proyecto seguiremos los siguientes pasos:

- seleccionar **File**, **Project structure**
- seleccionar **Libraries** en las opciones que aparecen en la parte izquierda

- sobre la ventana en blanco que aparece junto al menú de opciones de la izquierda, hacer clic en el icono +
- en el menú desplegable seleccionar la opción **From Maven....**
- en el asistente que se despliega teclead **junit** en la ventana de texto de la parte superior. Al teclear dar **Enter** para desencadenar la búsqueda
- una vez localizadas todas las coincidencias seleccionar la entrada **junit:junit:4.12** e incorporarla al proyecto
- cread un directorio llamado **test** dentro de la carpeta del proyecto, a la misma altura de **src**
- en la ventana de propiedades del proyecto, seleccionad a la izquierda la opción **Modules** y asignar a la carpeta **test** el papel de carpeta de pruebas (usando los iconos junto a **Mark us**, en concreto el de color verde asociado a este tipo de recurso)

Todas las pruebas se implementarán en archivos ubicados en el directorio llamado **test**. Se incluye un ejemplo de los dos archivos que se recomienda usar. En el primero de ellos (**ListadoTestP1.java**) se realiza una inicialización previa a la ejecución de los casos de prueba que consiste únicamente en cargar los datos almacenados en **datos.txt**. El código que realiza esta tarea se denomina **inicializacion** y debe ser estático e ir precedido de la anotación **@BeforClass**. Esto indica que esta tarea se realiza una sola vez antes de ejecutar el resto del código. Cada caso de prueba debe ir anotado con **@Test** y básicamente consistirá en comprobar que el resultado devuelto por un método coincide con determinado valor conocido de antemano. En el código se han incluido únicamente los métodos de prueba para comprobar que se generó bien **listadoArchivo** (con 5000 empleados), que se detectan y cuentan adecuadamente los dnis repetidos.

```
1  import listado.Sector;
2  import org.junit.BeforeClass;
3  import org.junit.Test;
4
5  import static org.junit.Assert.*;
6
7  import listado.ListadoEmpleados;
8  import listado.Ruta;
9
10 import java.io.IOException;
11 import java.util.Map;
12
13 /**
14  * Práctica 1 NTP
15  */
16 public class ListadoTest {
17    private static ListadoEmpleados listado;
18
19    /**
20     * Codigo a ejecutar antes de realizar las llamadas a los métodos
21     * de la clase; incluso antes de la propia instanciación de la
22     * clase. Por eso el método debe ser estatico
```

```
23      */
24     @BeforeClass
25     public static void inicializacion() {
26        System.out.println("Metodo inicializacion conjunto pruebas");
27        // Se genera el listado de empleados
28        try {
29           listado = new ListadoEmpleados("./data/datos.txt");
30        } catch (IOException e) {
31           System.out.println("Error en lectura de archivo de datos");
32        }
33        ;
34     }
35
36     /**
37      * Test para comprobar que se ha leido de forma correcta la
38      * informacion de los empleados
39      *
40      * @throws Exception
41      */
42     @Test
43     public void testConstruccionListado() throws Exception {
44        assert (listado.obtenerNumeroEmpleadosArchivo() == 5000);
45     }
46
47     /**
48      * Test para comprobar la deteccion de dnis repetidos
49      */
50     @Test
51     public void testComprobarHayDnisRepetidos() {
52
53        assert (listado.hayDnisRepetidosArchivo() == true);
54     }
55
56     /**
57      * Test para comprobar el numero de empleados con correos
58      * repetidos
59      */
60     @Test
61     public void testComprobarContadoresDnisRepetidosArchivo() {
62
63        assert (listado.contarEmpleadosDnisRepetidos() == 4);
64     }
65 }
```

Debéis incluir el resto de métodos para probar la funcionalidad de detección de correos repetidos (debe devolver **true**) y conteo de correos repetidos (hay 315 empleados con problemas de correo).

El segundo archivo se denomina **ListadoTestP2** y debe servir para comprobar el correcto funcionamiento de la funcionalidad relacionada con la el diccionario de empleados (una vez reparados los datos detectados en el archivo). Se incluye el código base del segundo archivo de pruebas:

```
1 import listado.Empleado;
2 import listado.Ruta;
```

```java
import listado.Sector;
import org.junit.BeforeClass;
import org.junit.Test;

import listado.ListadoEmpleados;

import java.io.IOException;
import java.util.List;
import java.util.Map;

import static org.junit.Assert.assertArrayEquals;

/**
 * Práctica 1 NTP
 */
public class ListadoTestP2 {
   private static ListadoEmpleados listado;

   /**
    * Codigo a ejecutar antes de realizar las llamadas a los métodos
    * de la clase; incluso antes de la propia instanciación de la
    * clase. Por eso el método debe ser estatico
    */
   @BeforeClass
   public static void inicializacion() {
      System.out.println("Metodo inicializacion conjunto pruebas");
      // Se genera el listado de empleados
      try {
         listado = new ListadoEmpleados("./data/datos.txt");
      } catch (IOException e) {
         System.out.println("Error en lectura de archivo de datos");
      }

      // Se reparan los problemas y se pasan los datos al datos miembro
      // listado
      Map<String, List<Empleado>> dnisRepetidos=listado.obtenerDnisRepetidosArchivo()
      listado.repararDnisRepetidos(dnisRepetidos);
      Map<String, List<Empleado>> correosRepetidos = listado.obtenerCorreosRepetidosA
      listado.repararCorreosRepetidos(correosRepetidos);
      listado.validarListaArchivo();

      // Se leen ahora los archivos de asignaciones de sectores y departamentos
      try{
         long errores;
         listado.cargarArchivoAsignacionSector("./data/asignacionSECTOR1.txt");
         listado.cargarArchivoAsignacionSector("./data/asignacionSECTOR2.txt");
         listado.cargarArchivoAsignacionRuta("./data/asignacionRUTA1.txt");
         listado.cargarArchivoAsignacionRuta("./data/asignacionRUTA2.txt");
         listado.cargarArchivoAsignacionRuta("./data/asignacionRUTA3.txt");
      } catch(IOException e){
         System.out.println("Problema lectura datos asignacion");
         System.exit(0);
      }
   }
```

```
57
58      /**
59       * Test del procedimiento de asignacion de grupos procesando
60       * los archivos de asignacion. Tambien implica la prueba de
61       * busqueda de empleados sin grupo asignado en alguna asignatura
62       *
63       * @throws Exception
64       */
65      @Test
66      public void testBusquedaEmpleadosSinRuta() throws Exception {
67         // Se obtienen los empleados no asignados a cada asignatura
68         // y se comprueba su valor
69         int res1, res2, res3;
70         res1=listado.buscarEmpleadosSinRuta(Sector.NOSECTOR).size();
71         res2=listado.buscarEmpleadosSinRuta(Sector.SECTOR1).size();
72         res3=listado.buscarEmpleadosSinRuta(Sector.SECTOR2).size();
73         System.out.println("res1: "+res1+" res2: "+res2+ " res3: "+res3);
74         assert (res1 == 418);
75         assert (res2 == 432);
76         assert (res3 == 399);
77      }
78
79      /**
80       * Prueba para el procedimiento de conteo de grupos para cada una
81       * de las asignaturas
82       */
83      @Test
84      public void testObtenerContadoresSector1() {
85         // Se obtienen los contadores para la asignatura ES
86         Map<Ruta, Long> contadores = listado.obtenerContadoresRuta(Sector.SECTOR1);
87         contadores.keySet().stream().forEach(key -> System.out.println(
88            key.toString() + "- " + contadores.get(key)));
89         // Se comprueba que los valores son DEPNA = 49, DEPSB = 48, DEPSM = 53, DEPSA
90         Long contadoresReferencia[] = {401L, 437L, 403L, 432L};
91         Long contadoresCalculados[] = new Long[4];
92         assertArrayEquals(contadores.values().toArray(contadoresCalculados),
93            contadoresReferencia);
94      }
95
96      /**
97       * Prueba del procedimiento general de obtencion de contadores
98       * para todas las asignaturas
99       *
100      * @throws Exception
101      */
102     @Test
103     public void testObtenerContadoresSector() throws Exception {
104        // Se obtienen los contadores para todos los grupos
105        Map<Sector, Map<Ruta, Long>> contadores =
106           listado.obtenerContadoresSectorRuta();
107
108        // Se comprueban los valores obtenenidos con los valores por referencia
109        Long contadoresReferenciaSector1[] = {401L, 437L, 403L, 432L};
110        Long contadoresReferenciaSector2[] = {428L, 425L, 388L, 399L};
```

```
111         Long contadoresReferenciaNoSector[] = {446L, 414L, 409L, 418L};
112 
113         // Se comprueban los resultado del metodo con los de referencia
114         Long contadoresCalculados[] = new Long[4];
115         assertArrayEquals(contadores.get(Sector.NOSECTOR).values().
116            toArray(contadoresCalculados), contadoresReferenciaNoSector);
117         assertArrayEquals(contadores.get(Sector.SECTOR1).values().
118            toArray(contadoresCalculados), contadoresReferenciaSector1);
119         assertArrayEquals(contadores.get(Sector.SECTOR2).values().
120            toArray(contadoresCalculados), contadoresReferenciaSector2);
121     }
122 
123     // Aqui habria que completar los casos de prueba para el resto de
124     // metodos a ofrecer por la clase Listado
125 }
```

Se han incluido algunas pruebas para ver la forma de comprobar el resultado de algunos métodos que devuelven arrays como argumentos. Se adjuntan también los siguientes números para que se usen como comprobación en los diferentes casos de prueba (podéis incluir otros si se considera necesario):

- los contadores de empleados asignados a cada ruta son respectivamente: (401, 437, 403, 432) para el primer sector; (428, 425, 388, 399) para el segundo y (446, 414, 409, 418) sin sector asignado (su dato miembro **sector** tendría valor **NOSECTOR**)
- el total de trabajados asignados a cada sector es 1673 (**SECTOR1**), 1640 (**SECTOR2**) y 1687 (**NOSECTOR**)
- el número de empleados sin ruta asignada en cada sector es 418, 432 y 399 respectivamente (para **NOSECTOR1**, **SECTOR2** y **SECTOR2**)
- el número de empleados con sector pero sin ruta es de 831
- el número de empleados sin sector pero con ruta asignada es de 1269

3 Detalles de implementacion

3.1 Tratamiento de archivos de asignacion (de sector y ruta

Nos fijamos en el tratamiento del archivo de asignacion de sector, ya que el correspondiente a asignacion de ruta sera similar. Ya que el patrón usado para trocear la información de una cadena considerando como separador el espacio en blanco se usará en varias tareas, sería adecuado definir un dato miembro estático en la clase **ListadoEmpleados**, de la forma siguiente:

```
1 private static Pattern patronEspacios=Pattern.compile("\\s+");
```

De esta forma se puede usar donde se precise sin necesidad de repetir su creación.

Teniendo en cuenta la existencia de este patrón el procedimiento de tratamiento del archivo de asignaciones a sectores podría hacerse en base a los siguientes métodos:

```
public long cargarArchivoAsignacionSector(String nombreArchivo) throws
↪  IOException
```

Este es el método base, que recibe como argumento el nombre del archivo a procesar. Su labor consistirá en:

- lectura de las líneas del archivo pasado como argumento y almacenamiento de las líneas leídas en una lista

  ```
  List<String> lineas = Files.lines(Paths.get(nombreArchivo)).
                              collect(Collectors.toList());
  ```

- la primera línea (almacenada en la posición 0) contendrá la cadena de caracteres que indica de qué sector se trata. Se necesita ahora un procedimiento que convierta la cadena (**String**) en el tipo **Sector**. Para ello se usará la siguiente sentencia:

  ```
  Sector sector = procesarNombreSector(lineas.get(0));
  ```

  que recibe como argumento la cadena con el contenido de la línea 0 del archivo. El funcionamiento de este método se explicará a continuación.

- una vez que se dispone del valor de sector (con el tipo adecuado) debe realizarse el procesamiento de todos los dnis escritos en al archivo mediante la creación de flujo a partir de la lista de líneas, descartar las dos primeras (mediante **skip** y aplicar el mapeo de forma que a cada línea se produzca una llamada al método que procesa la asignación del sector. Este método devolverá **true** o **false**, por lo que el mapeo produce un flujo de valores booleanos. Se filtran aquellos casos en que la respuesta haya sido **false** y se cuentan, de forma que se obtiene el contador de problemas en asignación de sector. El código que realiza esta tarea será:

  ```
  long errores=lineas.stream().skip(2).
           map(linea -> procesarAsignacionSector(sector, linea)).
           filter(flag -> flag == false).count();
  ```

Consideramos ahora los dos métodos auxiliares usados en la descripción anterior: **procesarNombreSector** y **procesarAsignacionSector**. El primero de ellos, como se indicó antes, recibirá como argumento un objeto de la clase **String** (el contenido de la primera línea del archivo) y devolverá un valor de tipo **Sector** que se corresponda con el contenido de la cadena. Su funcionamiento puede esquematizarse de la siguiente forma:

- se trocea la linea del archivo usando el patrón basado en la separación mediante espacios en blanco

  ```
  List<String> infos = patronEspacios.splitAsStream(cadena).
           collect(Collectors.toList());
  ```

  Esta lista contendrá un único elemento, almacenado en la posición 0, con la cadena que representa el sector al que se hace la asignación.

- se crea una condición para hacer el filtrado de todos los posibles valores del enumerado:

```
Predicate<Sector> condicion = sector ->
  ↪  (sector.name().equals(infos.get(0)));
```

- se usa esta condición para el filtrado:

```
sectorResultado = Arrays.stream(Sector.values()).
          filter(condicion).
          findFirst().get();
```

- el método finalizaría devolviendo el valor de la variable **sectorResultado**

Por su parte, el otro método auxiliar, **procesarAsignacionSector**, puede describirse de la siguiente forma:

- recibe como argumento un enumerado de tipo **Sector** y una cadena que representará el contenido de una línea del archivo de asignación, que contendrá únicamente un dni.
- se comienza entonces extrayendo las unidades informativas de la línea delarchivo (que será únicamente una como ya se ha indicado):

```
List<String> infos = patronEspacios.splitAsStream(datosAsignacion).
          collect(Collectors.toList());
```

- se localiza el empleado asociado a dicho dni

```
Empleado empleado = listado.get(infos.get(0));
```

- si la referencia es válida (hay realmente un empleado con dicho dni), se le asigna el sector y se devuelve **true**. En caso contrario se devuelve **false**

La funcionalidad correspondiente a asignación de ruta puede implementarse siguiendo la misma idea.

4 Material a entregar

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 hayais 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 queda fijafa para el día 2 de abril a las 23:55 horas. La entrega se hará mediante la plataforma **PRADO** (ya está abierta).