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 SE8. En la parte de código se han eliminado todos los acentos para evitar problemas con la codificación de caracteres.

2 Definición del problema

Se trata de implementar un sistema de gestión de empleados para analizar su asignación a divisiones y departamentos. La empresa se organiza en divisiones (ID, SW, HW y SER) y departamentos. Todas las divisiones presentan el mismo conjunto de departamentos: SB, SM y SA. Los identificadores de divisiones y departamentos se tratan mediante dos enumeraciones diferentes:

```
/**
 * Enumerado para representar los códigos de las divisiones de la empresa
 */
public enum Division {
    DIVNA, DIVID, DIVHW, DIVSW, DIVSER
}

/**
 * Enumerado con departamentos
 */
public enum Departamento {
   DEPNA, DEPSB, DEPSM, DEPSA
}
```

Los identificadores de división y departamentos que incluyen **NA** en su nombre se utilizan para indicar no asignación (ya sea a división, a departamento o ambos).

Se parte inicialmente de un archivo de empleados que contiene la información básica de cada uno de ellos, incluyendo **dni**, **nombre**, **apellidos** y **dirección de correo**. Todos estos datos están separados por comas, estando la información de cada empleado en una línea. El nombre de este archivo es **datos.txt**. A modo de ejemplo se incluyen las últimas líneas del mismo:

```
71515940,  MARIA TERESA,  MARIN VILA,  mamavi@acme.com
32871135,  MANUEL JESUS,  OLIVA VILA,  maolvi@acme.com
57599985,  MARINO,  VALLEJO ESTEVEZ,  mavaes@acme.com
53749950,  MARIA CONCEPCION,  VELEZ SIERRA,  mavesi@acme.com
68083877,  DANIELA,  BECERRA OLIVA,  dabeol@acme.com
09773355,  CAYETANO,  BORREGO ORDOÑEZ,  caboor@acme.com
22671082,  ABDESLAM,  CARBONELL DA SILVA,  abcada@acme.com
76738547,  CARMELO,  SERRANO CHAVES,  casech@acme.com
36401782,  MARIA PINO,  DE LA CRUZ ALBA,  madeal@acme.com
```

En archivos adicionales se ofrece la información de las asignaciones de estudiantes a divisiones y departamentos. Los nombres de los archivos de asignación tienen el formato **asignacionXX.txt** (donde XX alude a los códigos indicados previamente; al haber 5 códigos de división y 4 código de departamento, hay 9 archivos diferentes de asignación). En todos ellos los datos se organizan por líneas. En cada línea aparece un **dni**. La primera línea contiene el código de asignatura y la segunda línea está en blanco. A modo de ejemplo, las primeras líneas del arhivo **asignacionDIVSW.txt** son las siguientes:

```
DIVSW

35007339
31583157
76349795
24030436
58249259
63431355
06262915
```

También hay archivos que contienen información sobre empleados no asignados a división o departamento (los nombres de los mismos son **asignacionDIVNA.txt** (para divisiones) y **asignacionDEPNA.txt** (para departamentos). A partir de estos datos el sistema debe aportar la siguiente funcionalidad:

- generar una colección de objetos de la clase **Empleado**, que permita almacenar los datos relevantes de cada empleado (dni, nombre, apellidos, correo, división y departamento) y que se obtiene de procesar el contenido de los archivos de datos proporcionados (datos de empleados y asignaciones)

- la colección puede soportarse en una clase llamada **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. El dato miembro base de la clase podría ser el siguiente:

```
/**
  * Dato miembro para almacenar a los empleados como diccionario
  * con pares tipo (clave - valor) con el siguiente contenido:
  * <dni - Empleado>
  */
 private Map<String, Empleado> lista;
```

  De esta forma, 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 **lista** tenga almacenados todos los objetos de la clase **Empleado** (uno por cada línea de datos del archivo). Se recomienda crear un método auxiliar en esta clase, llamado **crearEmpleado**, que recibe como argumento una línea de datos del archivo y tras su análisis extraae la información básica necesaria para llamar al constructor de la clase **Empleado**. Con esta idea, el esquema de funcionamiento del constructor podría ser el siguiente:

```
creacion del diccionario sobre el dato miembro lista
obtener las lineas del archivo datos.txt
para cada linea
   - llamar al metodo auxuliar crearEmpleado, pasando la linea
     como argumento
   - almacenar el objeto de la clase Empleado en el diccionario
```

- la clase **Listado** contará con métodos específicos para asignación de división y departamento, procesando los archivos correspondientes Los nombres de estos métodos serán **cargarArchivoAsignacionDivision** y **cargarArchivoAsignacionDepartamento**. Estos métodos reciben como argumento el nombre del archivo a procesar. En definitiva, la creación completa de la lista de empleados, con todos sus datos, precisa el tratamiento de todos los archivos de asignación con estos métodos auxiliares. Esto puede hacerse en el mismo constructor de la clase.

- la clase **Listado** debe contar con un método denominado **toString** que compondrá una cadena de caracteres con toda la información relativa a los empleados.

- la clase **Listado** dispondrá de un método que permita obtener los contadores de empleados asignados a cada división y departamento: **obtenerContadoresDivisionDepartamento**. Este método debe devolver un diccionario del tipo:

```
Map<Division, Map<Departamento, Long>>
```

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

```
public Map<Departamento, Long> obtenerContadoresDepartamento(
                                                Division division)
```

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 una determinada división.

- la clase permitirá también obtener listados de empleados sin asignación a división o departamentos (o ambos). Para ello se implementarán los métodos que se indican a continuación:

```
/**
 * Metodo para buscar los empleados sin division asignada: es decir,
 * en el dato miembro division tendran el valor DIVNA
 */
public List<Empleado> buscarEmpleadosSinDivision()

/**
 * Metodo para buscar empleados con division asignada (no es DIVNA)
 * pero sin departamento: el valor del dato miembro departamento es
 * (DEPNA)
 */
public List<Empleado> buscarEmpleadosConDivisionSinDepartamento()

/**
 * Metodo para buscar todos los empleados no asignados a departamento
 * que pertenezcan a una determinada division
 * @param divisionObjetivo division de interes
 * @return lista de empleados sin departamento asignado
 */
public List<Empleado> buscarEmpleadosSinDepartamento(
                                              Division divisionObjetivo)
```

- esta clase también dispondrá de facilidades para determinar si existen dnis o correos electrónicos repetidos. En caso de existir debería proporcionar la lista de empleados en que ocurre tal circunstancia. Las declaraciones de estos métodos podrían ser las siguientes:

```java
/**
 * Metodo para determinar si hay dnis repetidos
 * @return
 */
public boolean hayDnisRepetidos()

/**
 * Metodo para obtener una lista de dnis repetidos, junto con la
 * lista de trabajadores asociados a cada dni repetido (en caso de
 * haberlos)
 */
public Map<String,List<Empleado>> obtenerDnisRepetidos()

/**
 * Metodo para determinar si hay correos repetidos
 */
public boolean hayCorreosRepetidos()

/**
 * Metodo para obtener una lista de dnis repetidos, junto con la
 * lista de trabajadores asociados a cada dni repetido (en caso de
 * haberlos)
 */
public Map<String,List<Empleado>> obtenerCorreosRepetidos()
```

- pueden incluirse todos los métodos auxiliares que se considere conveniente, tanto en la clase **Empleado** como **Listado**. Por ejemplo, en la clase **Listado** sería conveniente disponer de un método que indique si un empleado tiene tiene asignada división (y departamento).

- se usará **junit** para comprobar que la funcionalidad implementada no contiene errores. Se proporciona un conjunto mínimo de pruebas que deben integrarse en el proyecto y que deben pasarse sin errores para que se considere válido el trabajo de la práctica. Se deben incorporar pruebas adicionales de forma que toda la funcionalidad de la clase **Listado** se pruebe de forma adecuada.

3 Detalles de implementación

Se recomienda que se usen todos los mecanismos posibles de programación funcional. Esto implica que el programa debería contar con el menor número posible de iteraciones externas y variables. También es interesante implementando los métodos pedidos con la aproximación de programación imperativa para pasar posteriormente a la versión basada con programación funcional.

Con respecto a la forma de procesar el archivo de datos se dan las siguientes indicaciones para facilitar el trabajo. En primer lugar, para obtener un flujo con las líneas contenidas en un archivo de texto basta con usar las siguientes sentencias, pasando como argumento al método **get** el nombre del archivo a leer:

```java
// Se leen las lineas del archivo
Stream<String> lineas = Files.lines(Paths.get(nombreArchivo));
```

Una posible forma de proceder consistiría en disponer en la clase **Listado** de un método auxiliar (**crearEmpleado**) que reciba como argumento una línea leída del archivo y genera el objeto de la clase **Empleado** asociado. El análisis de la línea puede hacerse de la forma siguiente:

```
// Se define el patron para las comas que hacen de separadores
Pattern pattern = Pattern.compile("(,)");

// Se procesa la linea
List<String> infos = pattern.splitAsStream(linea).collect(Collectors.toList());
```

De esta forma el resultado es una lista de objetos de la clase **String**. Cada objeto contiene la información de uno de los datos miembro de interés (dni, nombre, apellidos y correo electrónico) que se usarán en la llamada al constructor de la clase **Empleado**.

La asignación de división y empleado se hará posteriormente. Esto hace necesario contar en la clase **Empleado** con los métodos que permitan asignar a un empleado división y departamento.

4 Casos de prueba

Como se ha indicando antes, se debe dotar al software de un conjunto de pruebas para garantizar su correcto funcionamiento. Se incluye aquí un ejemplo de posibles pruebas a realizar (sin ser completo ni exhaustivo), ya que puede servir de ayuda para realizar la implementación del código de la práctica.

```
import listado.Departamento;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

import listado.Listado;
import listado.Division;

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

/**
 * Práctica 1 NTP
 */
public class ListadoTest {
    private static Listado 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 Listado("./data/datos.txt");
        }catch(IOException e){
            System.out.println("Error en lectura de archivo de datos");
        };

```

```
33          // Una vez disponibles los empleados se leen las listas
34          // de asignaciones de empleados a cada grupo de las diferentes
35          // asignaturas consideradas
36          try {
37              listado.cargarArchivoAsignacionDivision("./data/asignacionDIVNA.txt");
38              listado.cargarArchivoAsignacionDivision("./data/asignacionDIVID.txt");
39              listado.cargarArchivoAsignacionDivision("./data/asignacionDIVSW.txt");
40              listado.cargarArchivoAsignacionDivision("./data/asignacionDIVHW.txt");
41              listado.cargarArchivoAsignacionDivision("./data/asignacionDIVSER.txt");
42              listado.cargarArchivoAsignacionDepartamento("./data/asignacionDEPNA.txt");
43              listado.cargarArchivoAsignacionDepartamento("./data/asignacionDEPSB.txt");
44              listado.cargarArchivoAsignacionDepartamento("./data/asignacionDEPSM.txt");
45              listado.cargarArchivoAsignacionDepartamento("./data/asignacionDEPSA.txt");
46          } catch (IOException e) {
47              System.out.println("Error en lectura de archivos de asignacion");
48          }
49          System.out.println();
50      }
51
52      /**
53       * Test para comprobar que se ha leido de forma correcta la
54       * informacion de los empleados (al menos que el listado contiene
55       * datos de 100 empleados)
56       * @throws Exception
57       */
58      @Test
59      public void testConstruccionListado() throws Exception{
60          assert(listado.obtenerNumeroEmpleados() == 1000);
61      }
62
63      /**
64       * Test del procedimiento de asignacion de grupos procesando
65       * los archivos de asignacion. Tambien implica la prueba de
66       * busqueda de empleados sin grupo asignado en alguna asignatura
67       * @throws Exception
68       */
69      @Test
70      public void testCargarArchivosAsignacion() throws Exception {
71          // Se obtienen los empleados no asignados a cada asignatura
72          // y se comprueba su valor
73          assert(listado.buscarEmpleadosSinDepartamento(Division.DIVNA).size() == 49);
74          assert(listado.buscarEmpleadosSinDepartamento(Division.DIVID).size() == 54);
75          assert(listado.buscarEmpleadosSinDepartamento(Division.DIVSW).size() == 42);
76          assert(listado.buscarEmpleadosSinDepartamento(Division.DIVHW).size() == 44);
77          assert(listado.buscarEmpleadosSinDepartamento(Division.DIVSER).size() == 49);
78      }
79
80      /**
81       * Prueba para el procedimiento de conteo de grupos para cada una
82       * de las asignaturas
83       */
84      @Test
85      public void testObtenerContadoresDepartamentos(){
86          // Se obtienen los contadores para la asignatura ES
87          Map<Departamento, Long> contadores = listado.obtenerContadoresDepartamento(
88                  Division.DIVSER);
89          contadores.keySet().stream().forEach(key -> System.out.println(
90                  key.toString() + "- " + contadores.get(key)));
91          // Se comprueba que los valores son DEPNA = 49, DEPSB = 48, DEPSM = 53, DEPSA = 41
92          Long contadoresReferencia[]={49L,48L,53L,41L};
93          Long contadoresCalculados[]=new Long[4];
94          assertArrayEquals(contadores.values().toArray(contadoresCalculados),
95                            contadoresReferencia);
96      }
97
```

```
98      /**
99       * Prueba del procedimiento general de obtencion de contadores
100      * para todas las asignaturas
101      * @throws Exception
102      */
103     @Test
104     public void testObtenerContadoresDivisionDepartamento() throws Exception {
105         // Se obtienen los contadores para todos los grupos
106         Map<Division, Map<Departamento, Long>> contadores =
107                 listado.obtenerContadoresDivisionDepartamento();
108
109         // Se comprueban los valores obtenenidos con los valores por referencia
110         Long contadoresReferenciaNA[]={49L,53L,53L,58L};
111         Long contadoresReferenciaID[]={54L,49L,42L,43L};
112         Long contadoresReferenciaHW[]={44L,43L,67L,62L};
113         Long contadoresReferenciaSW[]={42L,52L,45L,53L};
114         Long contadoresReferenciaSER[]={49L,48L,53L,41L};
115
116         // Se comprueban los resultado del metodo con los de referencia
117         Long contadoresCalculados[]=new Long[4];
118         assertArrayEquals(contadores.get(Division.DIVNA).values().
119                 toArray(contadoresCalculados),contadoresReferenciaNA);
120         assertArrayEquals(contadores.get(Division.DIVID).values().
121                 toArray(contadoresCalculados),contadoresReferenciaID);
122         assertArrayEquals(contadores.get(Division.DIVHW).values().
123                 toArray(contadoresCalculados),contadoresReferenciaHW);
124         assertArrayEquals(contadores.get(Division.DIVSW).values().
125                 toArray(contadoresCalculados),contadoresReferenciaSW);
126         assertArrayEquals(contadores.get(Division.DIVSER).values().
127                 toArray(contadoresCalculados),contadoresReferenciaSER);
128     }
129
130     // Aqui habria que completar los casos de prueba para el resto de
131     // metodos a ofrecer por la clase Listado
132 }
```

5 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 se fija para el día 27 de marzo. La entrega se hará mediante la plataforma **PRADO**.