Laboratorio de Software - 2025
Práctica n° 4

Temas
- Arreglos de primitivos y objetos.
- Clases de la API para manejar colecciones de objetos
- Genéricos
- Expresiones Lambda

(1) Se quiere mantener un conjunto de alumnos ordenados por número de legajo, correspondientes a una materia en una Facultad. Defina una clase `Alumno` en el paquete practica4, que contenga información del alumno, entre otras cosas el legajo, apellido y nombre y DNI.
a) Defina una clase llamada `Materia` en el mismo paquete que mantenga información sobre la materia y la nómina de alumnos. Use colecciones genéricas.
b) Defina en la clase Materia dos métodos: `addAlumno(Alumno alumno)` que agregue a alumnos a la nómina de alumnos existentes y `getAlumnos()` que retorne el conjunto de alumnos inscriptos en la materia.
c) Pruebe las clases definidas creando agregando al menos 10 alumnos en una Materia. Recorra la colección e imprima sus elementos.
d) Su colección está ordenada, ¿qué requisito tuvo que cumplir para poder mantenerla ordenada?
Se utilizó SortedSet ya que no permite duplicados y permite ordenar. Además la clase Alumno tuvo que implementar la clase Comparator para hacer un comparador que funcione.
e) Si quisiera que el criterio de orden se mantenga sobre el apellido y nombre ¿cómo lo haría?
Creando un comparador más que haga el orden por apellido y luego por nombre.
f) Analice una solución que permita mantener una nómina no sólo de alumnos sino también de empleados, clientes, etc. ¿Cómo definiría la colección?
Con una jerarquía de clases.

(2) Considere la siguiente clase:

```
public class Veterinaria<E> {
    private E animal;

    public void setAnimal(E x) { animal = x; }

    public E getAnimal() { return animal; }

public class Animal{ }

public class Gato extends Animal { }

public class Perro extends Animal { }
```

Indicar cual es el resultado de las siguientes operaciones:

- `Veterinaria<Animal> vet = new Veterinaria<Gato>();`
  Error de compilación. Los genéricos no son covariantes, aunque Gato sea subtipo de Animal, `Veterinaria<Gato>` no es subtipo de `Veterinaria<Animal>`.
  El tipo debería ser `<? extends Animal>` para indicar que `vet` puede ser una `Veterinaria` de cualquier clase que extienda `Animal`.

- `Veterinaria<Gato> vet = new Veterinaria<Animal>();`
  Error de compilación. Los genéricos son invariantes. `Veterinaria<Animal>` no puede asignarse a `Veterinaria<Gato>`.
  Debe ser `<? super Gato>` para indicar que `vet` puede ser una `Veterinaria` de cualquier superclase de `Gato`.

- `Veterinaria<?> vet = new Veterinaria<Gato>();`
  `vet.setAnimal(new Gato());`
  El comodín `?` indica que el tipo genérico es desconocido, por lo que sólo se pueden realizar operaciones de lectura y no de escritura por la seguridad de tipos. Por tanto el `setAnimal` funcionará mientras el `getAnimal` no lo hará.

- `Veterinaria vet = new Veterinaria();`
  `vet.setAnimal(new Perro());`
  Se utiliza *raw type*, el tipo sin parámetros. Esto desactiva la verificación de tipos en tiempo de compilación. El compilador mostrará un warning pero se ejecutará.

- `Veterinaria vet = new Veterinaria<?>();`
  `<?>` solo puede usarse en declaraciones de tipo.
  No se puede instanciar usando un comodín porque no representa un tipo concreto.

- `Veterinaria<? extends Animal> vet = new Veterinaria<Gato>();`
  Compila. Sin embargo, no se puede usar setAnimal.

(3) Se desea recuperar colecciones de palabras similares y para ello se deben implementar un motor de comparación basado en una técnica particular de **string matching**. Esta técnica define una función de similitud entre dos cadenas que indicará cuán similares son dichas cadenas.
Las técnicas utilizadas por estos tipos de motores de comparación están basadas en *tokenización* (descomposición en unidades básicas), y como resultado del proceso obtendremos un conjunto de *tokens*, o lo que es lo mismo, de Q-gramas (sub-cadenas de tamaño Q). La siguiente figura muestra un ejemplo de tokenización en gramas de tamaño 2:

Cadena original c	Posición de la ventana	Q-Grama extraído	Lista de Q-gramas
Glucosa	[#G]lucosa$	[#G]	[#G]
	#[Gl]ucosa$	[Gl]	[#G][Gl]
	#G[lu]cosa$	[lu]	[#G][Gl][lu]
	#Gl[uc]osa$	[uc]	[#G][Gl][lu][uc]
	#Glu[co]sa$	[co]	[#G][Gl][lu][uc][co]
	#Gluc[os]a$	[os]	[#G][Gl][lu][uc][co]
	#Gluco[sa]$	[sa]	[#G][Gl][lu][uc][co][sa]
	#Glucos[a$]	[a$]	[#G][Gl][lu][uc][co][sa][a$]

Figura 1 - Ejemplo del proceso de tokenización: Dada la cadena "Glucosa" se introducen caracteres de inicio y de final de cadena (# y $ símbolos no existentes en el alfabeto utilizado) y se obtiene una lista de Q-gramas mediante el uso de una ventana de tamaño 2 que se desliza a través de los caracteres de la cadena.

Usted debe:
a) Implementar un diccionario de gramas que almacenará el resultado de la factorización de palabras en Q-gramas, con Q=2, donde las claves del diccionario serán las gramas y el valor almacenado en el mismo serán las listas de palabras que contiene el grama de la clave.

b) Implementar un motor de comparación basado en la siguiente técnica de **string matching**:

- Distancia de Levenshtein menor a tres: Las cadenas comparadas deben poseer una distancia de Levenshtein menor a tres. La distancia de Levenshtein, distancia de edición o distancia entre palabras es el número mínimo de operaciones requeridas para transformar una cadena de caracteres en otra. Por ejemplo, la distancia de Levenshtein entre "casa" y "calle" es de 3 porque se necesitan al menos tres ediciones elementales para cambiar uno en el otro. (http://es.wikipedia.org/wiki/Distancia_de_Levenshtein).

El motor de comparación definirá un método que recibirá como parámetro un diccionario de gramas de tamaño 2 y una cadena. Dicho método dividirá la cadena en gramas de tamaño 2, explorará recolectando las palabras almacenadas en el diccionario de gramas que contenga alguno de los gramas de la cadena enviada como parámetro y retornará como resultado la colección de palabras que cumplen con el criterio de comparación.

(4) Implementación de **Expresiones Lambda en Java**
a) Defina una clase llamada **Facultad** que contenga en su interior una lista de alumnos, donde de estos últimos se guarda la siguiente información:

- nro de alumno
- nombres
- apellidos
- edad
- materia aprobada
- nota de aprobación

Sobre la clase **Facultad** implemente los métodos necesarios utilizando expresiones Lambda a fin de poder realizar las siguientes consultas:

1. Obtener el estudiante con mayor nota.
2. Imprimir dos estudiantes de la lista.
3. El que tomó el curso llamado "Laboratorio de Software".
4. Obtener los alumnos, cuyo nombre empiece con el carácter "P" y la longitud de su nombre sea menor o igual a 6.

Finalmente implemente una clase **TestFacultad** que permita probar las consultas anteriores.

b) Implemente el método **ordenarPorNota()** en la clase **Facultad** del ejercicio anterior. Dicho método ordena la lista de estudiantes por nota de aprobación de mayor a menor utilizando una clase interna que implementa la interface **java.util.Comparator**.

1. Modifique el código para reemplazar la clase interna por una expresión lambda.
2. Modifique la implementación para que utilice el método estático Comparator.comparingInt(). ¿Qué recibe como parámetro?
3. ¿Es posible utilizar una referencia a método? Utilícela en caso de ser posible.