Práctica 5

Genericidad, Colecciones, Lambdas y Patrones de Diseño

Inicio: Semana del 16 de abril.
Duración: 3 semanas.
Entrega: lunes 7 de mayo - 23:55h (todos los grupos)
Peso de la práctica: 30%

El objetivo de esta práctica es ejercitar conceptos de orientación más avanzados, como son

- *Diseño de clases genéricas, y altamente reutilizables*
- *Uso de la librería de colecciones Java*
- *Empleo de patrones de diseño*
- *Uso de expresiones lambda*

Introducción

En esta práctica desarrollaremos un conjunto de clases Java destinadas a la creación y acceso a grafos genéricos, compuestos de vértices que almacenen datos de un tipo establecido.

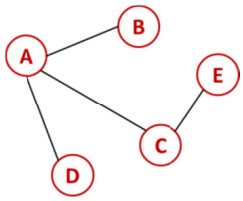
Para probar la implementación realizada, instanciaremos un grafo con datos de una red social con los personajes de la famosa novela Juego de Tronos del escritor estadounidense George R. R. Martin.

Sobre ese grafo desarrollaremos una serie de expresiones lambda que obtenga información recorriendo los vértices del grado de forma secuencial. Posteriormente, también sobre el grafo de prueba, utilizaremos el patrón de diseño Observer para desarrollar un programa que simule interacciones entre los personajes atendiendo a sus relaciones en el grafo, con el fin de confirmar cuáles de ellos son más relevantes en la novela.

Apartado 1. Grafos (3,5 puntos) [clases con tipos genéricos, colecciones Java]

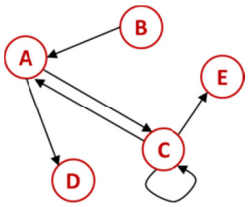
En matemáticas y ciencias de la computación, un grafo $G=\{V,A\}$ es un conjunto de objetos denominados vértices $V=\{v_1, v_2, \ldots, v_n\}$ unidos por enlaces llamados arcos $A=\{(v_i, v_j) \mid v_i, v_j \in V\}$.

Típicamente, un grafo se representa gráficamente como un conjunto de puntos (representando los vértices) unidos por líneas (arcos). La siguiente figura muestra un ejemplo de grafo etiquetado con 5 vértices y 4 aristas:



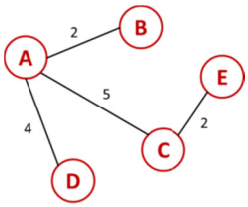
Todo arco del grafo anterior une dos vértices sin indicar un orden entre ambos, que quedan enlazados bidireccionalmente. Los grafos que tienen este tipo de arcos se denominan grafos no dirigidos.

Aquellos grafos cuyos arcos mantienen orden entre sus vértices se denominan grafos dirigidos. Gráficamente, el orden entre pares de vértices unidos por una arista se representa mediante una flecha cuyo sentido indica el orden. Cada arco distinguirá un vértice origen y un vértice destino.



Tanto los vértices como las aristas de un grafo pueden almacenar información diversa. Por ejemplo, en los grafos anteriores los vértices almacenan datos de tipo String: “A”, “B”, “C”, “D” y “E”.

En particular, se pueden asignar valores numéricos a los arcos, que tendrán un significado especial atendiendo a la información almacenada por el grafo: similitudes, distancias, fuerzas, etc. Esos valores suelen ser denominados pesos y los grafos que poseen arcos con pesos se denominan grafos ponderados.



En este apartado (**`package adsof1718.grafos`**), se pide implementar una serie de clases para la creación de grafos ponderados, dirigidos y no dirigidos. Más específicamente, se pide implementar:

- Una clase abstracta `Grafo`, que esté asociada a un grafo ponderado de arcos con **pesos** de tipo `double`. A la hora de crear un vértice o un arco dentro de un grafo, ha de asignársele un **identificador** único de tipo `int`.
- Dos clases no abstractas `GrafoDirigido` y `GrafoNoDirigido`, que hereden de `Grafo`, que estén asociadas a un grafo dirigido o a un grafo no dirigido respectivamente.

Las 3 clases han de implementarse con tipos "génericos", de tal manera que los datos que se almacenen en los vértices de un grafo sean de un tipo concreto, no necesariamente igual al de otros grafos. Por ello, la declaración de la clase Grafo sería:

```
public abstract class Grafo<T> {
   . . .
}
```

donde `T` es el *template* usado para referirse al tipo de datos de los vértices:

```
public class Vertice<T> {
   private final int id;  // identificador del vértice dentro del grafo
   private T datos;       // datos almacenados en el vértice
   . . .
}
```

En la clase Grafo se pide implementar/declarar los siguientes métodos (los que son abstractos se deberán implementar en las clases `GrafoDirigido` y `GrafoNoDirigido`):

```
public Vertice<T> addVertice(T datos)
protected Vertice<T> addVertice(int id, T datos)
public List<Vertice<T>> getVertices()
public int getNumVertices()

public abstract void addArco(Vertice<T> v1, Vertice<T> v2, double peso)
public boolean existeArco(Vertice<T> v1, Vertice<T> v2)
public int getNumArcos()

public abstract double getPesoDe(Vertice<T> v1, Vertice<T> v2)
public abstract List<Vertice<T>> getVecinosDe(Vertice<T> v) // devuelve los vértices que tienen un arco con v
                                                           // (en grafo dirigido, v ha de ser origen de los arcos)
public String toString()  // los vértices del grafo han de presentarse ORDENADOS POR IDENTIFICADOR
```

Además, se pide que dentro de un grafo el almacenamiento de los vértices y aristas se haga en variables de tipo Map. La declaración de los vértices podría ser como sigue:

```
protected Map<Integer, Vertice<T>> vertices;
```

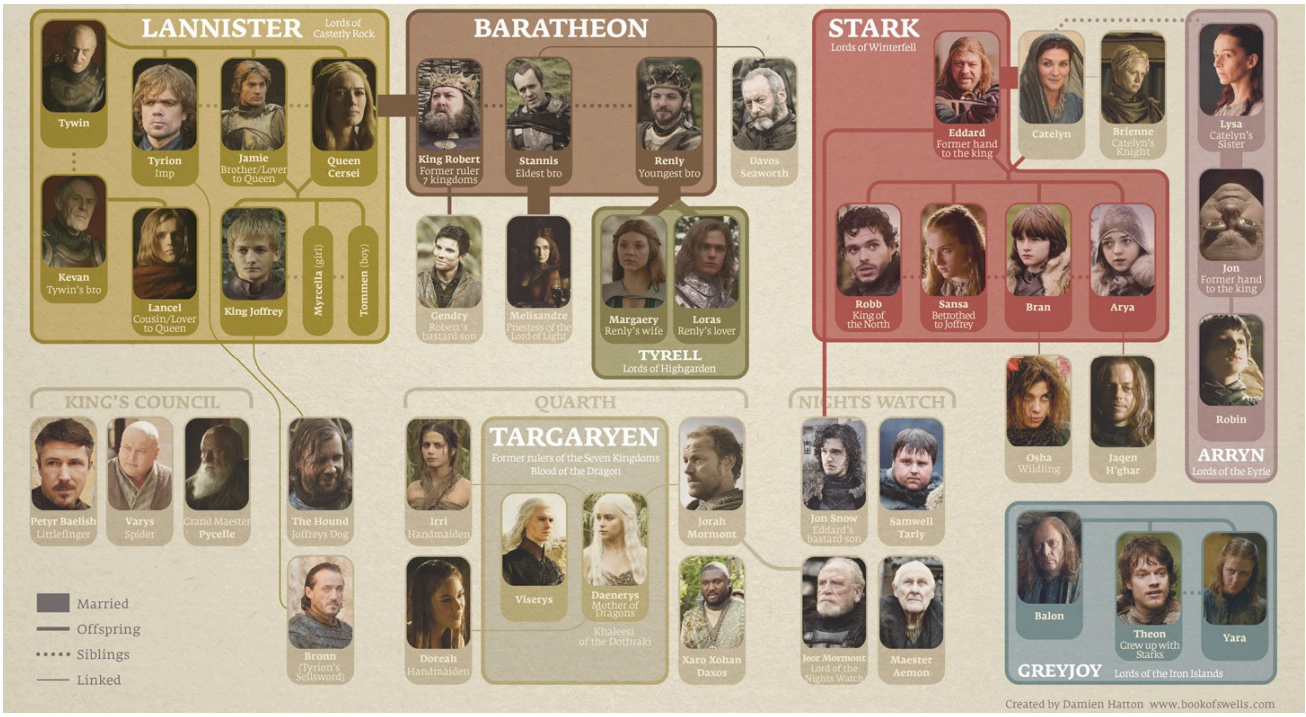
Como prueba preliminar del código desarrollado, se pide implementar y ejecutar el siguiente `main` que crea un grafo no dirigido/dirigido `g` con vértices de tipo `String`.

```
Grafo<String> g = new GrafoNoDirigido<>();
//Grafo<String> g = new GrafoDirigido<>();
System.out.println(g.getClass().getName() + "\n");

Vertice<String> v1 = g.addVertice("A");
Vertice<String> v2 = g.addVertice("B");
Vertice<String> v3 = g.addVertice("C");

g.addArco(v1, v2, 10);
g.addArco(v1, v2, 20);
g.addArco(v3, v1, 30);

System.out.println(g);

List<Vertice<String>> vecinos1 = g.getVecinosDe(v1);
System.out.println("Vecinos de " + v1 + ":");
for (Vertice<String> v : vecinos1) {
   System.out.println(v);
}

double peso1 = g.getPesoDe(v1, v2);
System.out.println("Peso entre " + v1 + " y " + v2 + ": " + peso1);
double peso2 = g.getPesoDe(v1, v3);
System.out.println("Peso entre " + v1 + " y " + v3 + ": " + peso2);
```

Apartado 2. Acceso a un grafo (3,5 puntos) [expresiones lambda]

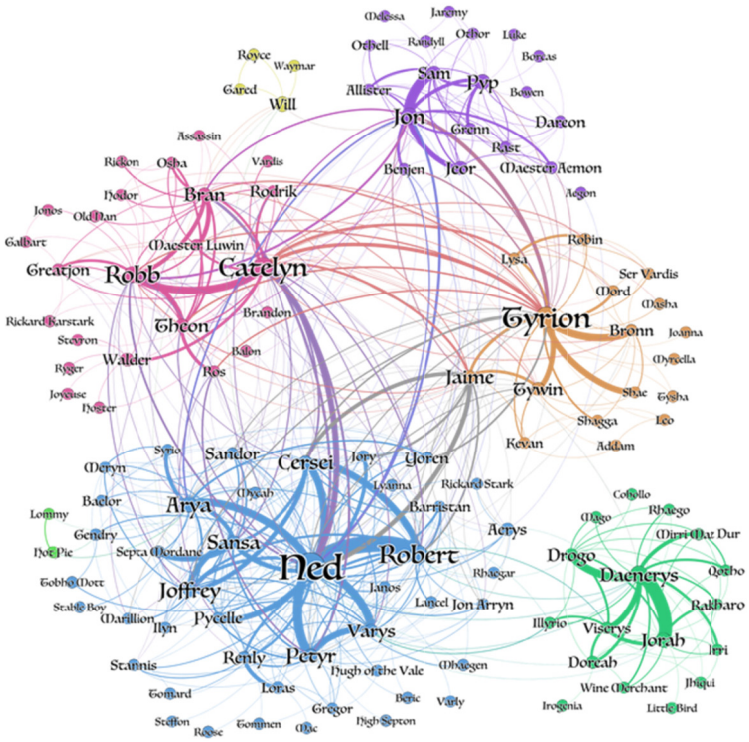
En este apartado (`package adsof1718.grafos.got`) vamos a crear un grafo ponderado no dirigido con datos de una red social entre los personajes de la popular serie de televisión y novela "**Juego de Tronos**" de George R. R. Martin. En particular, usaremos los datos disponibles en https://networkofthrones.wordpress.com para el primer libro de la novela (y primera temporada de la serie de televisión).

En la saga, los personajes pertenecen a "casas" o familias. La siguiente figura muestra personajes principales de las casas más relevantes: Lannister, Baratheon, Stark, Targaryen y Greyjoy.



En la red social proporcionada, dos personajes están conectados entre sí si "interactúan" (conversan o son citados) en un mismo párrafo al menos una vez. Así, representando cada personaje como un vértice del grafo de la red social, se pueden definir arcos con un peso cuyo valor es el número de veces que los pares de personajes correspondientes interactúan.

La siguiente figura ilustra el grafo a utilizar: los colores dan idea de las casas y el grosor de los arcos el peso con el que se unen los personajes.



Los datos de la red social se proporcionan en dos ficheros de texto plano:

- **got-s01-vertices.csv**, con los vértices del grafo, siguiendo el formato de línea: `id,nombre_personaje,casa_personaje`
- **got-s01-arcos.csv**, con los arcos del grafo, siguiendo el formato de línea: `id_personaje1,id_personaje2,peso`

Se pide que los datos sean almacenados en una clase `GrafoGOT`, que herede de `GrafoNoDirigido` y que tenga como tipo de datos de sus vértices una clase denominada `PersonajeGOT`, para guardar de cada personaje su nombre y su casa (`null` en caso de que el personaje no tuviese casa).

El constructor de `GrafoGOT` será el que lea los datos de los ficheros de entrada.

```
public GrafoGOT(String ficheroVertices, String ficheroArcos) throws Exception {
   . . .
}
```

En la clase `GrafoGOT`, se pide además implementar los siguientes métodos que usen (**únicamente siempre que sea posible**) expresiones lambda para obtener información diversa sobre la red social:

- **`public Vertice<PersonajeGOT> getVertice(String nombre)`**: devuelve el vértice que contiene el personaje cuyo nombre se indica como argumento de entrada.
- **`public List<String> casas()`**: devuelve una lista con los nombres de las casas sin repeticiones, ordenadas, y sin valores `null`.
- **`public List<String> miembrosCasa(String casa)`**: devuelve una lista con los nombres de los personajes de una casa indicada como argumento de entrada. En este caso, se pide explícitamente crear y usar un `Predicate`.
- **`public Map<String, Integer> gradoPersonajes()`**: devuelve los personajes y sus "grados", esto es el número de vecinos que tiene cada vértice.
- **`public Map<String, Integer> gradoPonderadoPersonajes()`**:devuelve los personajes y sus "grados ponderados", esto es la suma de los pesos de los arcos que unen a cada vértice con sus vecinos.
- **`public Map<String, Integer> personajesRelevantes()`**: devuelve los personajes y sus "grados ponderados", pero sólo para aquellos personajes cuyo grado ponderado es superior al grado ponderado medio en la red social.

Se pide implementar un `main` en `GrafoGOT` que haga una prueba de cada uno de los métodos anteriores. Utilizando de nuevo expresiones lambda, el `main` visualizará por pantalla los resultados de llamar a los métodos anteriores (excepto a `getVertice`).

Apartado 3. Notificaciones de eventos en un grafo (3 puntos) [patrón de diseño *Observer*]

En este apartado (**`package adsof1718.grafos.got.simulador`**) se va a desarrollar un "simulador" de interacciones entre personajes de Juego de Tronos mediante la red social creada en el apartado 2. En la simulación se guardará registro de cada interacción que se genere.

La simulación, muy sencilla, consistirá en repetir N (p.e., 10.000) veces el siguiente algoritmo:

1. Seleccionar un personaje (vértice) u al azar de la red social (grafo).
2. Para cada vecino v de u:
   - Calcular probabilidad de interacción entre u y v como: $p(u,v) = peso(u,v) / \sum_w peso(u,w)$, donde w son los "vecinos" de u, y $peso(u,v)$ es el peso del arco que une los vértices u y v.
   - Generar un número aleatorio $r \in [0,1]$.
   - Si $r < p(u,v)$, registrar "interacción" entre u y v.

El registro de interacciones deberá hacerse mediante un conjunto de clases que sigan el patrón de diseño Obsever:

- Una clase `ObservadorGOT`, que herede de una clase abstracta `Observador` (u `Observer`), y que para un personaje dado guarde registro del número de interacciones que ese personaje ha tenido con otros en la simulación, distinguiendo entre interacciones con personajes de su casa e interacciones con personajes de otras casas.

  En su constructor, un observador recibirá como entrada y **se dará de alta** en un `SimuladorGOT` (ver abajo).

  ```
  public ObservadorGOT(SimuladorGOT s, PersonajeGOT p)
  ```

  En un método `actualizar`, invocado por el simulador cuando haya habido interacción (evento) en la red social, solicitará al simulador los personajes origen y destino involucrados en la interacción, y si el origen es el personaje del observador, este último actualizará sus registros de interacciones.

  ```
  public void actualizar() {
     SimuladorGOT simulador = (SimuladorGOT) super.sujeto;

     PersonajeGOT origen = simulador.getOrigen();
     List<PersonajeGOT> destinos = simulador.getDestinos();
     . . .
  }
  ```

Finalmente, en el método `toString` de `SimuladorGOT`, se deberán presentar los números de interacciones con el siguiente formato (los valores son resultados de una simulación):

```
Jon Snow
  Interacciones: 704
    Con miembros de su casa: 78
    Con miembros de casa ajena
      : 410
      Baratheon: 3
      Cassel: 3
      Greyjoy: 6
      Lannister: 5
      Marsh: 28
      Mormont: 24
      Noye: 49
      Rykker: 21
      Targaryen: 20
      Tarly: 37
      Thorne: 20
```

- Una clase `SimuladorGOT` que herede de una clase abstracta `Sujeto` (o `Subject`) y que implemente en un método `interaccion` el **algoritmo presentado arriba** para un personaje dado.

```
public void interaccion(String nombre)
```

  En este método, si hubiese al menos una interacción del personaje con uno de sus vecinos, ha de llamarse al método `notificar` (de la superclase `Sujeto`), que enviará **notificaciones a sus observadores** sobre el evento, para que aquellos a los que les corresponda actualicen sus registros de interacciones.

Para probar la implementación desarrollada, se pide incluir un `main` en `SimuladorGOT` que realice una simulación con N ejecuciones del método `interaccion` del simulador con personajes elegidos aleatoriamente del grafo del apartado 2.

```
. . .
GrafoGOT g = . . .
SimuladorGOT simulador = new SimuladorGOT(g);
. . .
List<String> nombresPersonajes = . . .              // lista de personajes
for (int n = 0; n < N; n++) {
   String nombre = nombresPersonajes.get(. . .);    // se elige un personaje de forma aleatoria
   simulador.interaccion(nombre);
}
. . .
```

En el `main`, una vez realizada la simulación, se visualizarán por pantalla cada uno de los observadores dados de alta en el simulador.

Apartado 4. Creación de grafos (1 punto, opcional) [patrón de diseño *Factory method*]

En este apartado (**`package adsof1718.grafos.factoria`**) se pide implementar una clase `FactoriaGrafos<T>` que mediante un método estático `crearGrafo` cree y devuelva un grafo un tipo dado (p.e., dirigido, no dirigido). Esta clase ha de tener una enumeración `TiposGrafo` con los posibles tipos de grafos (p.e., `DIRIGIDO`, `NO_DIRIGIDO`). El método tiene como tipo de retorno una interfaz `IGrafo`, que ha de ser "implementada" por `Grafo` (y consecuentemente, por todas sus subclases), y que contiene los prototipos de método dados para `Grafo` en el apartado 1.

```
public static <T> IGrafo crearGrafo(TiposGrafo tipoGrafo) throws IllegalArgumentException
```

Para probar la factoría, se pide considerar un nuevo tipo de grafo, `COMPLETAMENTE_CONECTADO`, caracterizado por tener todos sus vértices unidos entre sí. Sin necesidad de llamar al método `addArco`, cada vez que en el grafo se añada un nuevo vértice se crearán para éste arcos con el resto de vértices y él mismo.

El nuevo tipo de grafo ha de implementarse en una clase

```
public class GrafoCompletamenteConectado<T> extends GrafoNoDirigido<T>
```

En las pruebas, se pide ejecutar el siguiente `main` en `PruebaFactoriaGrafos` con los 3 tipos de grafos considerados en la práctica. Para ello, hay que poner/quitar los comentarios de las 3 primeras líneas de código. Como se solicita en las normas de entrega, en la memoria habrá que incluir las salidas por pantalla de las ejecuciones de los `main` en todos los apartados de la práctica.

```
IGrafo<Integer> g = FactoriaGrafos.<Integer>crearGrafo(FactoriaGrafos.TiposGrafo.NO_DIRIGIDO);
//IGrafo<Integer> g = FactoriaGrafos.<Integer>crearGrafo(FactoriaGrafos.TiposGrafo.DIRIGIDO);
//IGrafo<Integer> g = FactoriaGrafos.<Integer>crearGrafo(FactoriaGrafos.TiposGrafo.COMPLETAMENTE_CONECTADO);

Vertice<Integer> v1 = g.addVertice(10);
Vertice<Integer> v2 = g.addVertice(20);
Vertice<Integer> v3 = g.addVertice(30);

g.addArco(v1, v2, 0);
g.addArco(v1, v3, 0);

System.out.println(g);

System.out.println("existe_arco(" + v1 + "," + v2 + ") = " + g.existeArco(v1, v2));
System.out.println("existe_arco(" + v2 + "," + v1 + ") = " + g.existeArco(v2, v1));
```

Normas de entrega:

- Se debe entregar:
  - un directorio **src** con el código Java de cada apartado, organizado en los `packages` establecidos en el enunciado.
  - un archivo PDF con una breve justificación de las decisiones que se hayan tomado en el desarrollo de la práctica, los problemas principales que se han abordado y cómo se han resuelto, y con las salidas que se muestren por pantalla en las ejecuciones realizadas en cada apartado.
- Se debe enviar un único fichero ZIP con todo lo solicitado, que ha de llamarse de la siguiente manera: GR<numero_grupo>_<nombre_estudiantes>.zip. Por ejemplo Marisa y Pedro, del grupo 2261, entregarían el fichero: GR2261_MarisaPedro.zip.