Práctica 1 - Búsqueda y ordenación

Esta segunda práctica parte del resultado del laboratorio 1 (clase `AlmacenLab1`) e

Objetivos

1. Poner en práctica conceptos básicos de herencia en Java
2. Implementar algoritmos de inserción eficiente de elementos en arrays o listas ordenados
3. Utilizar algoritmos de búsqueda en listas o arrays ordenados
4. Diseñar e implementar algoritmos eficientes en listas o arrays ordenados
5. Depurar un programa, para detectar y corregir errores:
   a. Utilizando casos de prueba con JUnit
   b. Desarrollando casos de prueba adicionales
   c. Añadiendo trazas/registros en el código
   d. Usar el depurador para detectar y corregir errores

Descripción de los datos

En esta práctica utilizaremos datos sobre registros de geolocalización (*checkins*) de usuarios. También tendremos información sobre amistad entre diferentes usuarios.

Los datos de todos los usuarios y registros se guardarán en una clase a la que llamaremos `GeoAlmacen`. Además, implementaremos una serie de métodos que nos permitirán buscar entre los usuarios y los registros.

Cada usuario tiene un identificador, una lista de registros y una lista de amigos. Un registro contiene el identificador del usuario, una localización (con latitud y longitud) y el tiempo en que se hizo el registro (`LocalDateTime`).

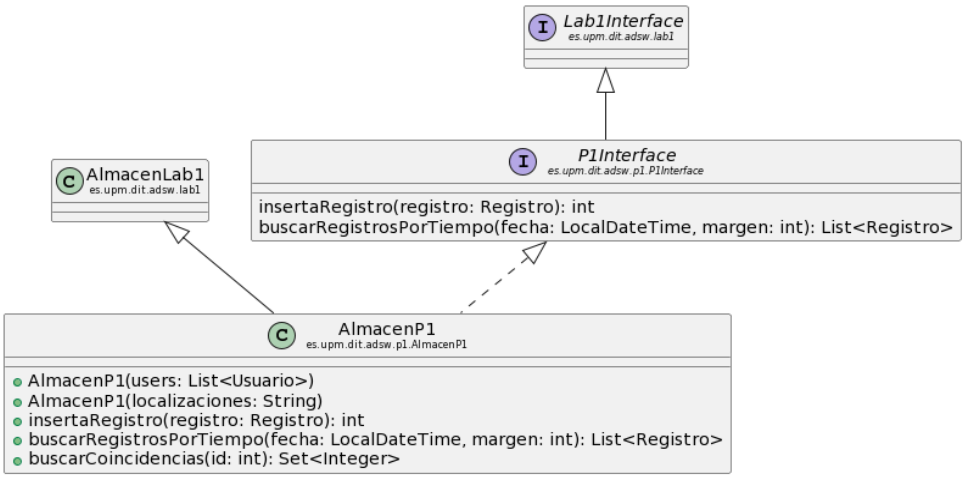
Utilizaremos conjuntos de datos de diferente tamaño, de la forma `data/locations<N>.tsv`, que contienen registros de `<N>` usuarios (desde 10 hasta 10.000).

De momento, nos centraremos en los ficheros más pequeños (entre 10 y 100 usuarios). En siguientes prácticas tendremos que utilizar ficheros mayores.

Diagrama de clases

Los elementos principales de esta práctica son los siguientes:

Sólo se debe implementar la clase `es.upm.dit.adsw.p1.AlmacenP1`. El resto de clases e interfaces no deben modificarse.



1 Diagrama de clases

Actividades

Los ejercicios consisten en **implementar la clase** `AlmacenP1` especificada en el diagrama de clases, método a método. Esta clase funcionará igual que la clase `es.upm.dit.adsw.lab1.AlmacenLab1`, con la diferencia de que también implementará los métodos definidos en el interfaz `es.upm.dit.adsw.p1.P1Interface`.

No se debe modificar ningún fichero fuera del paquete `es.upm.dit.adsw.p1`**, ni la interfaz** `P1Interface`. Se pueden ampliar los tests con pruebas nuevas, pero no se aconseja modificar las ya existentes.

Ejercicio 0: Descargar el proyecto y añadir los resultados del laboratorio

Se debe descargar el fichero `ADSW-p1.zip` del repositorio en GitHub. El fichero debe importarse en eclipse mediante la opción `File -> Import -> Existing projects into workspace`, y después seleccionando el fichero `ADSW-lab1.zip`.

Si todo ha ido bien. veremos un proyecto con dos paquetes: `es.upm.dit.adsw.p1` y `es.upm.dit.adsw.geosocial`.

A este código, deberemos añadir el código del laboratorio (paquete `es.upm.dit.adsw.lab1`), al completo.

Cada uno de los siguientes ejercicios tiene una serie de pruebas asociadas. ### Ejercicio 1: Implementar la clase `AlmacenP1`

Antes de nada, implementaremos la clase `AlmacenP1`, al igual que hicimos en el laboratorio. En este caso, queremos que la clase cumpla la interfaz `P1Interface` y que herede de la clase `AlmacenLab1`.

La diferencia entre el constructor de `AlmacenP1` y el de `AlmacenLab1` será que `AlmacenP1` ordenará la lista de registros en el propio constructor (utilizando los métodos ya vistos).

```
public class AlmacenP1 extends ... implements ... {

    // Implementar los constructores a partir de una lista de usuarios y de
una cadena de localizaciones
```

Para probar que el constructor funciona, se pueden utilizar los tests unitarios de `TestsFuncionales.java`. En particular, los tests `test1_Constructor` y `test2_ConstructorFichero`.

Para lanzar los tests, la clase debe compilar correctamente, por lo que deberemos añadir todos los métodos necesarios a la clase. Esto se puede hacer manualmente (comprobando la interfaz), o utilizando la función `add unimplemented methods` de eclipse:

Ejercicio 2: Implementar el método `insertarRegistro`

En la clase `AlmacenLab1` no era posible añadir registros una vez creados. En este ejercicio vamos a solucionar este problema añadiendo el método `insertarRegistro`.

Este método debe añadir el registro dado a la lista de registros del almacén, **y a la lista de registros del usuario correspondiente**. Una vez añadido el registro, la lista de registros debe seguir estando ordenada. Para ello, este método debe calcular en qué posición se debe introducir el registro nuevo.

```
    /**
     * Añade un registro a los guardados, manteniendo el orden.
     *
     * El registro se añade tanto a la lista dentro del almacén como al
usuario al que corresponda el registro.
     *
     * Este método inserta el nuevo registro de forma eficiente. No requiere
re-ordenar toda la lista.
     * Para ello, primero busca la posición en la que se debe insertar el
nuevo registro.
     *
     * @param registro Registro a insertar en el almacén
     * @return índice del registro en la lista de registros del almacén
     */
    public int insertaRegistro(Registro registro) {}
```

Para probar esta funcionalidad, se proporcionan un test que crea un almacén con unos usuarios específicos (`test3_Insertar`). Se recomienda leer los tests para entender el proceso y realizar pruebas adicionales si se considerase necesario.

Ejercicio 3: Implementar el método `buscarRegistrosPorTiempo`

El método `buscarRegistrosPorTiempo` funciona de forma similar a `buscarUsuariosPorTiempo`, salvo que en lugar de devolver la lista de usuarios, este método devuelve la lista completa de registros encontrados.

```
/**
* Busca registros cercanos a una fecha dada. Este método funciona de forma
```

```
análoga al método buscarUsuariosPorTiempo
*
* @param fecha
* @return
*/
public List<Registro> buscarRegistrosPorTiempo(LocalDateTime fecha, int
margen);
```

Para probar este funcionalidad, se proporcionan el test `test4_BusquedaTiempoManual`.

Ejercicio 4: Optimizar el método `buscarCoincidencias`

Por último, se pide hacer una nueva implementación más eficiente del método `buscarCoincidencias`. El método `buscarCoincidencias` del lmaboratorio 1 comprobaba todas las combinaciones de los registros del usuario pedido con el resto de usuarios. Si el número de registros totales es muy alto, esta búsqueda es muy costosa.

Dos registros coinciden si han sido realizados en un margen corto de tiempo y en la misma localización. En el nuevo método `buscarCoincidencias` nos aprovecharemos del hecho de que la lista de registros **está ordenada en función de la fecha**. En lugar de comprobar cada registro del usuario con **todos** los registros, podemos comparar sólamente con los registros que estén cercanos en el tiempo (utilizando la función `buscarRegistrosPorTiempo`).

Se puede usar este código para la nueva implementación:

```
@Override
public Set<Integer> buscarCoincidencias(int id) {
    return null;
}
```

Para comprobar esta funcionalidad, se proporcionan tres tests: `test5_CoincidentesManual`, `test6_Coincidentes` y `test7_CoincidentesComplejidad`.

Anexos:

Anexo 1: Cargar un proyecto en Eclipse

- Seleccione el menú File > Import > Existing Projects into Workspace > Next
- Navegue hasta el archivo zip que contiene el proyecto tal como se ha bajado de Moodle.
- Compruebe que el proyecto está marcado, y seleccione Finish

Anexo 2: Generar y acceder a la documentación con Javadoc en Eclipse

La documentación existente se encuentra en la carpeta *doc* del proyecto. Para consultarla, abra el fichero *index.html* en un navegador (botón derecho > Open with > Web browser).

También puede consultar la documentación desde una ventana del editor de código Java. Si posiciona el ratón sobre el nombre de una clase o un método aparece una ventana auxiliar con un resumen de la documentación.

Si tiene activada la vista *Javadoc* (con Window > Show View > Javadoc), al hacer clic sobre el

nombre de un elemento se mostrará la documentación correspondiente en la ventana correspondiente a esta vista.

Para generar o actualizar la documentación *javadoc* vaya al menú Project > Generate Javadoc. Si aparecen errores de codificación de caracteres asegúrese de poner las opciones -encoding utf8 -docencoding utf8 -charset utf8 en el cuadro *VM options* de la tercera ventana que aparece (después de hacer Next dos veces).

Anexo 3: Crear una clase de *JUnit*

Sitúese sobre la ventana del editor correspondiente a la clase que quiere probar y vaya al menú File > New > Junit Test Case. Asegúrese de que está seleccionada la opción "New JUnit Jupiter test", y conteste afirmativamente si el entorno le pide añadir la biblioteca Junit 5 al proyecto.

Anexo 4: Configurar el registrador (*logger*) java.util.logging

Opción 1

Los pasos básicos para configurar y usar en un programa son:

- Importar el paquete

```
import java.util.logging.*;
```

- Crear el configurador, que debe ser un atributo de la clase:

```
static final Logger LOGGER = Logger.getLogger(ListaTrazas.class.getName());
```

- Configurar el registrador. Ejecutar las siguientes instrucciones en el constructor en el que se van a poner los registros:

```
LOGGER.setUseParentHandlers(false);
handler = new ConsoleHandler();
handler.setLevel(Level.FINEST);
LOGGER.addHandler(handler);
LOGGER.setLevel(Level.FINEST);
```

- Incluir las trazas donde se considere:

```
LOGGER.info("Comentario");
LOGGER.fine("La lista: " + this.toString());
```

Opción 2

En algunas implementaciones, la opción previa duplica las trazas. A continuación se proporciona otra opción:

- Importar el paquete

```
import java.util.logging.*;
```

- Copiar el fichero logging.properties por defecto en un diretorio del usuario. Este fichero se encuentra en el paquete de Java en cada computador. Para facilitar, se adjunta una versión de este fichero en la distribución de este laboratorio.
- Asignar el nivel por defecto de la consosla a: FINEST

```
java.util.logging.ConsoleHandler.level = FINEST
java.util.logging.ConsoleHandler.formatter =
java.util.logging.SimpleFormatter
```

- En el código definir el fichero de configuración que el usuario ha cambiado y crear el gestor de trazas:

```
System.setProperty("java.util.logging.config.file",
"/MiDirectorio/logging.properties");
LOGGER = Logger.getLogger(ConfigurarSuma.class.getName());
```

- Finalmente, es posible elegir el nivel preferido:

```
LOGGER.setLevel(Level.FINER);
```

Anexo 5: Uso del depurador

El depurador permite identificar y eliminar errores de un programa que compila y ejecuta pero que no produce resultados correctos. El depurador ejecuta el programa de forma interactiva, permitiendo observar una a una las instrucciones que se ejecutarán, las variables activas en memoria y sus valores. Para iniciar el depurador sobre la clase que contiene el método main marque -Menú: Run->Debug

Las herramientas disponibles para el control de la ejecución son varias:

- **Puntos de parada "*breakpoints*".** Paran la ejecución del programa en instrucciones determinadas.
- **Perspectiva de depuración** con las siguientes vistas:
  - Vista de visualización y modificación de valores de variables.
  - Vista de consola que muestra la salida del programa
  - Vista de editor del código fuente con una línea verde en la instrucción que va a ser ejecutada
  - Vista de depuración indicando la línea de código que va a ser ejecutada
  - Vista de vigilancia de expresiones

Pantallazo de eclipse

- **Control de la ejecución** paso a paso, entrando en los métodos (*Step in)* o ejecutando los métodos completos y parando al terminar de ejecutarlos (*Step over*).

Para buscar un error en un programa pondremos un punto de parada en la primera sentencia ejecutable del método main. Iremos ejecutando controladamente el código entrando en los métodos suma y multiplicación de la clase Operaciones reales saltando las instrucciones que ejecutan constructores o llamadas a métodos de clases de la API de Java.

Se observa que las variables *op, uno* y *dos* sólo aparecen en la vista de variables en memoria tras su declaración. Al entrar en la ejecución del método *suma* se modifican varias vistas. En la del editor se ve la clase *OperacionesReales*. En la vista de variables se observan las variables disponibles por el método suma: la referencia a la propia instancia, *this*, y los parámetros *dos* y *uno*. Estas variables contienen los valores pasados al invocar el método. Aunque las variables tienen los mismos nombres que en la sentencia que invoca al método, están cambiadas de orden de modo que la variable *dos* contiene el valor almacenado en la variable *uno* de main. Se puede seguir ejecutando y salir del método. Se puede ver que los valores de las variables *uno* y *dos* no han cambiado.

Respecto al ejercicio 2, utilice el depurador para ver qué ocurre y verificar si se ha producido o

no el intercambio de valores deseado. Para ello se pone otro punto de parada en la sentencia que imprime el mensaje de cambio de sección. Al ejecutar el programa en modo debug, éste se parará en el primer punto de parada. Siga la ejecución hasta al siguiente punto de parada para ya ir instrucción a instrucción. Continúe con la depuración hasta entender por qué un método funciona y otro no.

En la tercera sección del programa se ejecuta la suma y multiplicación de dos números complejos. El programa funciona sin problemas, pero no da el valor adecuado. Debe identificar y corregir los errores usando el depurador.

Anexo 6: tratamiento de fechas en Java

En esta práctica, será necesario utilizar métodos para modificar fechas (añadiendo un cierto desfase en minutos) y comprobar el orden temporal de dos fechas. Además de la documentación enlazada desde el laboratorio 1, se proporciona el siguiente código de ejemplo:

```
// obtener la fecha actual
LocalDateTime ahora = LocalDateTime.now();
// obtener fecha y hora hace 5 minutos:
LocalDateTime hace5minutos = ahora.minusMinutes(5);
// obtener fecha y hora dentro de 5 minutos
LocalDateTime en5minutos = ahora.plusMinutes(5);
// comprobar que la fecha hace 5 minutos es ANTES que la fecha actual
assert(hace5minutos.isBefore(ahora));
// comprobar que la fecha en 5 minutos es posterior a la fecha actual
assert(en5minutos.isAfter(ahora))
Análisis y Diseño de Software, 2023

Grado en Ingeniería de Tecnologías y Servicios de
Telecomunicación

ETSI de Telecomunicación

Universidad Politécnica de Madrid
```