

UNIVERSIDAD CEU - SAN PABLO
ESCUELA POLITECNICA

MEMORIA: Práctica 1

Aplicación para gestionar cultivos de bacterias

Ana Ventura-Traveset Cervera

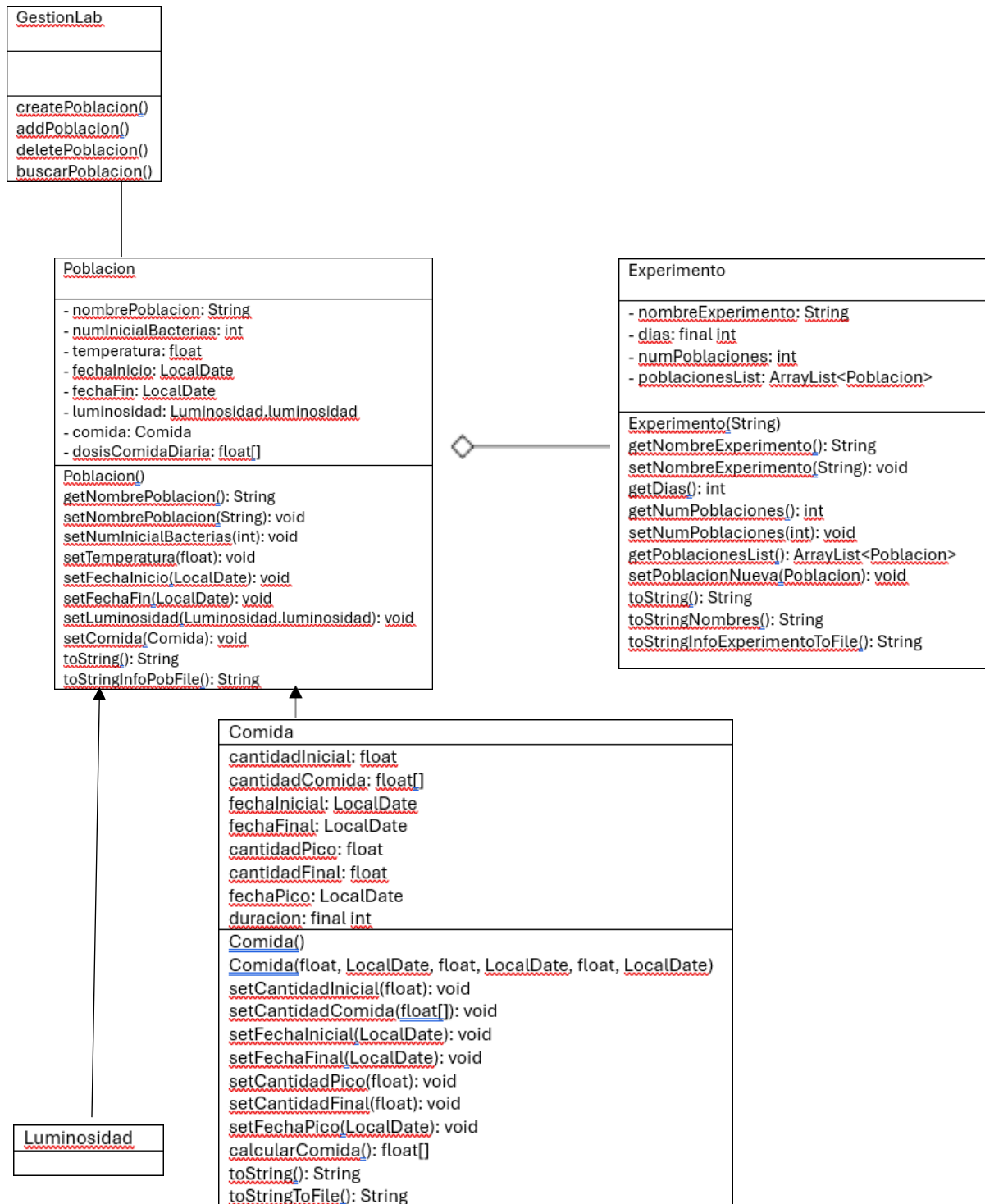
2023-2024

Grado Ingeniería Biomédica

Algoritmos y Estructuras de datos

- ANÁLISIS Y DESCRIPCIÓN DE LA APLICACIÓN. ESTE ANÁLISIS Y DESCRIPCIÓN DARÁ RESPUESTA A LAS SIGUIENTES PREGUNTAS:

- Diagramas de clases UML.



Las Poblaciones forman parte de los experimentos, pero pueden existir sin ellos.

Las poblaciones tienen clases Comida y Luminosidad. En ambos casos es una relación de 1-1. Cada población puede tener UNA Luminosidad y UNA Comida.

GestionLab, controla a las Poblaciones (las crea, añade a experimentos, borra de experimentos y busca y encuentra).

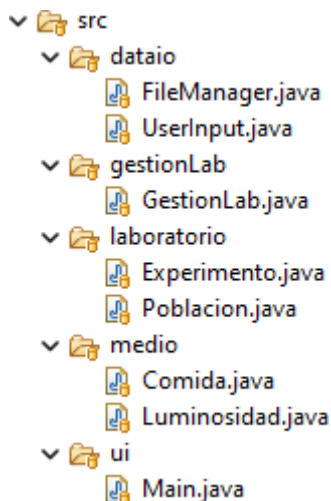
- Cómo se han organizado y estructurado las clases y cuál es la responsabilidad de cada una.
- Qué decisiones de diseño se han tomado.
- Qué comprobaciones de integridad (y excepciones) se han implementado.
- Qué técnicas de ordenación y búsqueda ha utilizado y por qué lo ha hecho.

Para conseguir mayor flexibilidad y a su vez claridad en el código, se ha intentado abordar la práctica siguiendo un modelo MVC (Modelo, Vista, Controlador), que es un patrón de arquitectura de software que separa la interfaz que ve y usa el usuario de la lógica de negocio en los tres componentes distintos.

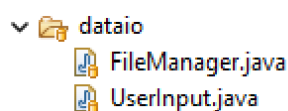
Descripción más detallada del modelo **MVC**:

- **VISTA:** Representa la interfaz de usuario. En este caso, de la vista se encarga el paquete ``ui``, cuyas siglas significan “User Interface”. En este paquete se encuentra el main con el menú de la aplicación.
- **MODELO:** Es dónde se encuentra la lógica de negocio. La representación del sistema. En este caso hay un paquete que se encarga de ello, el paquete ``laboratorio``, en el que están las clases Experimento y Poblacion.
- **CONTROLADOR:** Este se encarga de responder a la interacción del usuario en la interfaz y realiza las peticiones al modelo para después pasarlos a la vista. Como controladores, tenemos a dos paquetes responsables. Por un lado, el paquete ``dataio``, que significa “Data Input Output”, que se encarga de controlar el flujo de datos tanto de entrada como de salida por teclado o a un fichero. Y por otro lado el paquete ``gestionLab`` que se encarga de como su nombre indica, gestionar el laboratorio (los experimentos con sus poblaciones, crearlas, añadirlas, borrarlas, etc).

A continuación, se explicará más en detalle cada uno de estos cinco paquetes en los que se ha dividido la práctica y las clases que los componen. Pero antes un pequeño y simple diagrama, para visualizar bien los paquetes con sus correspondientes clases:



PAQUETE dataio:



Este paquete es uno de los encargados del control del programa. Es el que se encarga del manejo de datos, flujos de entrada y salida, lectura de datos de teclado y manejo de ficheros. Para ello se han creado dos clases.

- **Clase *FileManager*:**

Por un lado, está la clase *FileManager*, que se encarga de manejar todo lo que tenga que ver con ficheros (abrirlos, guardarlos, etc).

No tiene ningún atributo, pero sí un constructor vacío. Del que se hace uso e

Importa:

```
import gestionLab.GestionLab;
import laboratorio.Experimento
import laboratorio.Poblacion;
import medio.Comida;
import medio.Luminosidad;
import java.io.*;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
```

Se importa la clase *GestionLab* del paquete *gestionLab* porque contiene el método de añadir poblaciones en los experimentos, que se usa en el método de la clase *FileManager* `abrirArchivo`. Se importa el paquete *laboratorio*, con sus dos respectivas clases `Experimento` y `Poblacion` porque en el método `abrirArchivo`, es dónde se crean los experimentos y las poblaciones que éste contiene. Además del uso que se hace en la clase *FileManager* del método `toStringInfoExperimentoToFile()` de *Experimento* y `toStringInfoPobFile()` de *Poblacion*. Además se necesita importar el paquete *medio* con sus dos clases `Comida` y `Luminosidad`, ya que se van a usar para crear el experimento que haya en el archivo que se decida abrir, con sus respectivas poblaciones, que cada una de ella tiene un atributo `Comida` y otro `Luminosidad.luminosidad`. Cabe añadir que de la clase *Comida*, también se utilizan los métodos `calcularComida` y su `toStringToFile()` mediante el `toStringInfoPobFile()` de cada *Poblacion*. Otro atributo de población que es necesario importar es el `LocalDate`, por eso se importa `java.time.LocalDate`. El paquete *io*, se importa por completo ya que está constituido por una serie de interfaces y clases destinadas a definir y controlar los distintos tipos de flujos, el sistema de ficheros. Permite por ejemplo lanzar la excepción de `FileNotFoundException` en el método `abrirArchivo()`. Por último, se importa la clase *DateTimeFormatter* del paquete `java.time.format`, ya que esta clase se utiliza para formatear las fechas.

Métodos:

Ambos métodos `abrirArchivo` y `guardarArchivo`, son públicos y estáticos, para poder usarlos fuera de este paquete y sin necesidad de instanciar una clase de tipo *FileManager*.

`abrirArchivo(String nombreExperimento): Experimento`

Este método se encarga de abrir un archivo de texto que contiene información sobre un experimento y cargar esa información en la memoria del programa en forma de objetos de tipo *Experimento* y *Poblacion*. Lanza la excepción *FileNotFoundException*, que se utiliza en el main en try catch.

Como parámetro se le pasa un String, nombreExperimento que será a su vez el nombre del archivo que se va a abrir. Se espera que este nombre esté en formato de texto simple y se concatenará con ".txt" para formar el nombre completo del archivo.

Las variables utilizadas en este método son las siguientes: un objeto File, file que representa el archivo que se va a abrir; un Experimento experimento que se utilizará para almacenar la información del experimento que se carga desde el archivo; FileInputStream, fileInputStream, InputStreamReader inputStreamReader, BufferedReader bufferedReader que sirven para manejar el flujo de entrada de datos desde el archivo y por último un String, stringInfoTotal que se utiliza para acumular la información leída del archivo y que se imprimirá al final del proceso.

El proceso de lectura del archivo sigue los siguientes pasos:

Se crea un objeto File que representa el archivo a abrir, al que se le pone el nombre del experimento que se le pasa como parámetro.

Se inicializan los objetos FileInputStream, InputStreamReader y BufferedReader para leer el archivo a null.

Se declara un String stringInfoTotal para almacenar información del experimento y sus poblaciones, que se inicializa a "", para poder ir sumándole la información de experimento y luego de cada población mediante el uso de "+=", que lo que hace es coger stringInfoTotal y añadirle lo que esté a la derecha de "+=".

Se crea un try catch principal, en el que va básicamente todo el código. Éste sirve para capturar las excepciones en caso de que no se pueda abrir el archivo, porque no exista por ejemplo.

Después del catch, se abre una estructura finally, estructura finally, lo cual implica que se ejecutará independientemente de si ocurre una excepción o no. Su objetivo es cerrar los flujos de lectura de archivos (BufferedReader, InputStreamReader y FileInputStream) que se abrieron previamente para evitar fugas de recursos y liberar memoria. Dentro del finally, se hacen verificaciones para confirmar que no sean nulos mediante un if y se intentan cerrar. En caso de que ocurra una excepción al intentar cerrar alguno de estos flujos dentro del try, se captura la excepción en un catch y se imprime un mensaje de error. Esto asegura que los flujos de lectura de archivos se cierren adecuadamente, incluso si se produce un error durante el proceso de lectura del archivo.

Volviendo al try catch "principal", el código que va dentro de éste es el siguiente: se inicializa bufferedReader (que se encarga de leer el archivo de texto) envolviendo inputStreamReader, que a su vez envuelve a fileInputStream que representa el archivo especificado por file.

Se lee el archivo por líneas usando el método readLine().split("\n"); del BufferedReader y se almacena en un Array de String, todosArgs. Siendo "\n" un salto de línea.

La primera línea del archivo, que en este caso esta almacenada en todosArgs[0], contiene información general del experimento, el nombre y la duración en días separada por ";". Se almacena esta información en otro Array de Strings, infoExperimento, mediante todosArgs[0].split(";").

Se divide esta línea usando el carácter ; como delimitador y se almacenan las partes en un array de Strings llamado infoExperimento. El nombre del experimento se extrae de infoExperimento[0] y los días del experimento de infoExperimento[1]. Se crea un objeto Experimento utilizando el constructor al que se le pasa como parámetro el nombre extraído del archivo.

A continuación, se le añade a stringInfoTotal, la información del experimento, mediante experimento.toStringInfoExperimentoToFile(). Realmente, este stringInfoTotal, es simplemente

para que al abrir el archivo y guardarlo en memoria, se imprima por pantalla, para que sea más agradable para el usuario y tenga claro si el experimento que ha abierto, es el deseado.

Se declara un String line. Que se utiliza en un bucle while ((line = bufferedReader.readLine()) != null), que lo que hace es pedirle que siga en el bucle mientras haya más líneas por leer. Dentro del bucle se crea otro Array de String, infoPoblacion al que se le pasa `line.split(";")`. Ahí es dónde se almacena la información de cada población que se encuentra en el archivo separada por ";". Se crea una Poblacion mediante el constructor vacío de éste. Y se le va "seteando" cada atributo, que se guardan en el archivo en un orden muy específico que se ha mantenido en el array de Strings infoPoblacion. En cada posición, se encuentra un atributo distinto. Y así se va haciendo. Al recuperar la información para crear las Poblaciones, se hace desde un String, se hace el correspondiente parse para cada atributo. Al casi final de la creación de cada Poblacion, se crea un objeto de tipo Comida para poder pasárselo mediante un setComida a cada Poblacion (ya que es parte de sus atributos). Para acabar se llama al método addPoblacion de la clase GestionLab para ir añadiendo las poblaciones al experimento. Y a cada población creada, se le añade la información a stringInfoTotal. Al salir del bucle while, se imprime por pantalla un aviso de que se ha cargado el fichero en memoria y el stringInfoTotal, para ver lo que se ha cargado. Finalmente se cierra bufferedReader y se devuelve el experimento creado.

guardarArchivo(String nombreExperimento): boolean

Este método se encarga de guardar la información de un experimento en un archivo de texto. Este método se utiliza para "Guardar" tanto como para "Guardar Como". Al método se le pasan dos parámetros, el primero un String nombreExperimento y el segundo un Experimento. A la hora de "guardar como", en el main se le pedirá al usuario que introduzca por teclado el nombre con el que desea guardar su experimento y ese será el String que se le pasará al método como primer parámetro. Sino, si simplemente queremos "guardar", en ese caso se le pasa como primer parámetro el método getNombreExperimento() de la clase Experimento.

En cuanto al código del método: Se crea un objeto File que representa el archivo donde se guardará la información del experimento. El nombre del archivo se construye concatenando el nombre del experimento (que se ha pasado como primer parámetro) y la extensión ".txt". Se declara e inicializa a null un objeto PrintWriter que se utilizará para escribir en el archivo de texto. También se inicializa una variable "comprobación" de tipo boolean como false, que se utilizará para indicar si la operación de guardado fue exitosa o no.

Se abre un bloque try catch, que si ocurre alguna excepción durante la escritura en el archivo, se maneja imprimiendo la traza de la excepción y estableciendo comprobacion en false. A continuación del catch, se encuentra una estructura finally, que asegura que el objeto PrintWriter se cierre si no es null. Volviendo al bloque try catch, dentro del try, se abre el archivo para escritura utilizando el objeto PrintWriter y pasándole el file. Se crea un String experimentoInfoFile, en el que se le pasa la información del experimento (nombre y días del experimento) mediante los métodos get de Experimento, separada por ";". Acto seguido se le escribe esta información al archivo mediante el método de PrintWriter println(String experimentoInfoFile). A continuación para escribir la información de cada población del experimento, para ello se crea un bucle for con el que se itera sobre la lista de poblaciones del experimento y se escribe la información de cada población en líneas separadas en el archivo. La información se saca del método toStringInfoPobFile() (en el que la información está separada por ";") de cada población y se va añadiendo al archivo de nuevo mediante el método de PrintWriter print(String). Por último, se cambia el valor de "comprobacion" a true y se cierra printWriter.

Fuera del bloque try catch, se devuelve el boolean "comprobacion", para asegurarse de que se haya guardado o no correctamente el archivo.

- **Clase *UserInput*:**

Por otro lado, la clase *UserInput*, que se encarga de leer por teclado lo que el usuario inserte (enteros, reales, fechas, etc). Esta clase es de vital importancia ya que contiene todas las funciones que permiten leer de teclado y aseguran que lo que haya introducido el usuario sea correcto. En caso de no serlo, se le volverá a pedir el dato hasta que introduzca uno que sea correcto. De esta manera conseguimos evitar que el programa se ejecute con datos no válidos y, o sea en vano o acabe dando falsos resultados.

Importa:

```
import medio.Luminosidad;  
  
import java.text.ParseException;  
import java.text.SimpleDateFormat;  
import java.time.LocalDate;  
import java.time.ZoneId;  
import java.util.Date;  
import java.util.Scanner;
```

Se importa la clase *Luminosidad* del paquete *medio* para poder utilizar la enumeración *luminosidad* que se encuentra en la clase *Luminosidad*.

De la clase *Scanner* del paquete *java.util* es necesaria su importación porque se utiliza en todos los métodos de la clase *UserInput* para leer datos por teclado.

El resto de importaciones son necesario para el método *readDate*, para el manejo de fechas.

Métodos:

Todos los métodos de esta clase son públicos y estáticos, para poder usarlos fuera de este paquete y sin necesidad de instanciar una clase de tipo *UserInput*.

***readString(String peticion): String*:**

Se imprime la petición proporcionada como parámetro *peticion*, que es la indicación o mensaje que se muestra al usuario para solicitar la entrada de una cadena de texto.

Se declara una variable de tipo boolean *hecho*. Esta variable se utiliza para controlar el bucle *do-while*, que garantiza que se realice al menos un intento de lectura.

Se inicializa la cadena de texto *miString* como una cadena vacía. Esto se hace fuera del bloque *try-catch* para asegurarse de que esté disponible fuera de ese bloque y pueda ser devuelta al final del método.

Se inicia un bucle *do-while*, que se ejecutará hasta que se haya leído correctamente una cadena de texto (*hecho == true*).

Dentro del bucle *do-while*, se crea un objeto *Scanner* llamado *sc* para leer la entrada del usuario desde la entrada estándar (*System.in*). Lo que es lo mismo que decir la entrada por teclado.

Se utiliza el método *nextLine()* del objeto *Scanner* para leer la línea de texto ingresada por el usuario y se almacena en la variable *miString*.

Si la lectura es exitosa, se establece *hecho* en *true* para salir del bucle. Si se produce una excepción durante la lectura (por ejemplo, si el usuario ingresa un tipo de dato no válido), se captura la excepción en el bloque *catch*. En este caso, se imprime un mensaje de error y se utiliza *e.printStackTrace()* para imprimir información detallada sobre la excepción.

Finalmente, una vez que se ha leído correctamente una cadena de texto, esta se devuelve como resultado del método para ser utilizada por el código que lo llamó.

`readInt(String petición): int:`

Como parámetro se le pasa un String petición: Es una cadena de texto que se muestra al usuario para solicitarle que ingrese un número entero.

Posee dos variables. Una de tipo boolean, que le llama hecho que se utiliza como bandera para controlar si se ha completado con éxito la entrada de datos. Y una segunda de tipo int, milnt, donde se almacenará el número entero introducido por teclado por el usuario. En un inicio se inicializa a 0 fuera del bucle do while, para que pueda ser devuelta como parámetro por el método.

Se utiliza un bucle do-while para asegurarse de que se solicite al usuario que introduzca un número entero hasta que lo haga correctamente.

Dentro del bucle, se muestra el mensaje petición para solicitar al usuario que ingrese un número entero.

Se crea un objeto Scanner para leer la entrada del usuario por teclado.

Se utiliza el método nextInt() del objeto Scanner para leer un número entero ingresado por el usuario y asignarlo a la variable milnt.

Mediante un bucle if else, se verifica si el número ingresado es negativo. Si lo es, se muestra un mensaje de error, ya que en toda la aplicación no se hace uso de números negativos y que hablamos todo el rato de cantidades reales de poblaciones por ejemplo o del número de opción del menú.

Si la entrada es válida (es decir, no es negativa), se establece hecho en true, lo que indica que la entrada se ha completado con éxito.

Si ocurre un error durante la entrada de datos (por ejemplo, si el usuario ingresa una cadena en lugar de un número entero), se captura la excepción y se muestra un mensaje de error.

El bucle continuará ejecutándose mientras hecho sea false, es decir, mientras la entrada no se haya completado con éxito.

Una vez que se completa con éxito la entrada de datos, se devuelve el número entero introducido por teclado por el usuario almacenado en milnt.

`readFloat(String petición): float:`

Este método funciona exactamente de la misma manera que el método readInt aunque posee dos diferencias. La primera es que a pesar de que de igual manera se crea un objeto Scanner para leer la entrada del usuario por teclado.

Posteriormente se utiliza el método nextFloat() del objeto Scanner, en vez de nextInt() para leer un número real introducido por el usuario y asignarlo a la variable miFloat. En el caso de los float, no se realiza un bucle if else, para comprobar que no se introduzca un número negativo, ya que por ejemplo, la temperatura, que es un float puede ser bajo 0. En el caso de la cantidad de comida que también son floats pero no puede ser negativa, se regula en la clase GestionLab, al crear las poblaciones. Finalmente se devuelve el float introducido por el usuario, almacenado en miFloat.

`readLuminosidad(String petición): Luminosidad.luminosidad`

Como parámetro se le pasa un String petición que se muestra al usuario para solicitar el ingreso del nivel de luminosidad.

El método posee dos variables: la primera que es luminosidad: Una variable del tipo Luminosidad.luminosidad que almacenará el nivel de luminosidad seleccionado por el usuario, en un principio se inicializa a null para poder después de pedirla dentro del bucle, poder devolverla fuera de éste. La segunda variable es lum, de tipo String que almacena la entrada del usuario. Y por último la variable de tipo boolean, hecho que actúa como una bandera para indicar si la entrada de datos se ha completado con éxito.

Se emplea un bucle do-while para garantizar que el usuario ingrese un nivel de luminosidad válido antes de continuar.

Dentro del bucle, se muestra el mensaje petición para solicitar al usuario que introduzca el nivel de luminosidad.

Se crea un objeto Scanner para leer la entrada del usuario por teclado.

La entrada del usuario se almacena en la variable lum utilizando el método nextLine() del objeto Scanner.

Se comparan las opciones posibles de luminosidad (ALTA, MEDIA, BAJA) con la entrada del usuario (ignorando las diferencias de mayúsculas y minúsculas) utilizando el método equalsIgnoreCase() de la clase String.

Si la entrada del usuario coincide con alguna de las opciones válidas, se asigna el valor correspondiente al enum luminosidad y se establece hecho en true, indicando que la entrada se ha completado con éxito.

Si la entrada no coincide con ninguna de las opciones válidas, se muestra un mensaje de error y se establece hecho en false, indicando que la entrada no es válida.

El bucle continuará ejecutándose mientras hecho sea false, lo que significa que la entrada aún no se ha completado correctamente.

Una vez que se completa exitosamente la entrada de datos, el método devuelve el nivel de luminosidad seleccionado por el usuario, representado por el enum luminosidad de la clase Luminosidad.

readDate (String petición): LocalDate

Como parámetro se le pasa un String petición: que se muestra al usuario para solicitar el ingreso de la fecha.

Las variables utilizadas en este método son fecha que es un objeto de tipo Date que almacenará la fecha parseada desde la entrada del usuario.

También posee fechaADevolver: Un objeto LocalDate que representa la fecha convertida a formato LocalDate. Y por último una variable boolean que se llama hecho que actúa como una bandera para indicar si la entrada de datos se ha completado con éxito.

Se utiliza un bucle do-while para garantizar que el usuario ingrese una fecha válida antes de continuar.

Dentro del bucle, se muestra el mensaje petición para solicitar al usuario que ingrese la fecha.

Se muestra un mensaje adicional indicando el formato de fecha esperado (yyyy.MM.dd).

Se crea un objeto Scanner para leer la entrada del usuario desde System.in.

Se crea un objeto SimpleDateFormat con el formato esperado de la fecha.

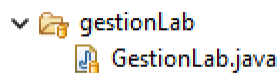
Se utiliza el método parse de SimpleDateFormat para convertir la cadena de entrada del usuario en un objeto Date.

Si la conversión tiene éxito, se utiliza el método toInstant() de Date para obtener una representación de fecha y hora, y luego se utiliza atZone() para obtener la zona horaria por defecto del sistema y finalmente toLocalDate() para obtener un LocalDate. La variable hecho se establece en true y se sale del bucle.

Si ocurre un error durante el proceso de análisis (ParseException), se captura y se muestra un mensaje de error. La bandera hecho se establece en false para repetir el bucle y solicitar al usuario que ingrese la fecha nuevamente.

Una vez que se completa exitosamente la entrada de datos y la conversión, el método devuelve el objeto LocalDate que representa la fecha ingresada por el usuario.

PAQUETE gestionLab:



Este paquete es el encargado de gestionar las poblaciones (crearlas, añadirlas a experimentos, borrarlas de experimentos y buscarlas).

- **GestionLab:**

Importa:

```
import dataio.UserInput;
import laboratorio.Experimento;
import laboratorio.Poblacion;
import medio.*;
import java.text.ParseException;
import java.time.LocalDate;
```

Se necesitan estos import ya que por un lado se va a hacer uso de los métodos de UserInput, a la hora de crear poblaciones por ejemplo, pidiendo los datos para crearla por teclado al usuario. De ahí también la necesidad de importar el paquete de ParseException.

Se importa el paquete laboratorio porque se hace uso de ambos objetos, tanto poblaciones que es todo lo que estamos gestionando como de los experimentos que son lo que contienen las poblaciones.

El paquete medio es también importante a la hora de crear una población para definir su comida y luminosidad.

Finalmente, el import de la clase LocalDate, que hacemos uso de ella para crear una población y definir las fechas de inicio, pico y fin de su alimentación durante el experimento.

Métodos:

Todos los métodos de esta clase son públicos y estáticos, para poder usarlos fuera de este paquete y sin necesidad de instanciar una clase de tipo GestionLab.

createPoblacion: Poblacion

Se le pasa como parámetro un Experimento y lanza las excepciones Exception y ParseException, ya que va a haber introducción de datos por teclado y varios parse al crear una población.

En este método se le pide al usuario toda la información de la población mediante los métodos que se encuentran en la clase UserInput del paquete dataio, como readInt() por ejemplo y se va creando la población haciendo el respectivo set() para cada atributo de población. En este método se controla que no se introduzca una comida negativa por ejemplo o que las fechas tengan concordancia (que no sea la fecha final antes de la inicial por ejemplo). Esto se hace mediante while true, que lo que hace es que va a pedir el dato hasta que se consiga y mediante bucles if else, dentro de éste para controlar este tipo de “errores” previamente mencionados. Finalmente se añade la población al experimento.

addPoblacion: void

Se le pasa como atributos una población y un experimento. Y se encarga de resetear el número de poblaciones del experimento y añadir la población al arrayList de Poblacion de experimento.

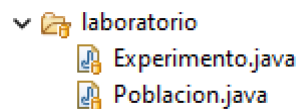
deletePoblacion: void

Se le pasa como atributos una poblacion y un experimento. Se recorre mediante un bucle for la lista de poblaciones del experimento y dentro de este for, se crea un if. Cuando el nombre de la población introducida sea igual que el nombre de alguna de las poblaciones del experimento, entonces ésta se elimina mediante el método remove(índice), de los ArrayList.

buscarPoblacion: Poblacion

Se le pasa como atributos una poblacion y un experimento. Se recorre mediante un bucle for la lista de poblaciones del experimento y dentro de este for, se crea un if. Cuando el nombre de la población introducida sea igual que el nombre de alguna de las poblaciones del experimento, entonces se devuelve esa población. En caso de no encontrarla se lanza una excepción.

PAQUETE laboratorio:



○ **Clase Experimento:**

Es la clase que representa un experimento en la aplicación.

Importa:

```
import java.util.ArrayList;
```

Se realiza este import porque los experimentos van a tener ArrayList de Poblacion.

Atributos:

nombreExperimento: String: Es el nombre del experimento. Es privado para encapsularlo y se accede a él mediante los métodos get y set.

dias: final int: Representa la duración del experimento en días. Está marcado como final para indicar que su valor no puede cambiar después de su inicialización, que de hecho se hace al principio de la clase. Se le atribuye el valor de 30.

numPoblaciones: int: Número de poblaciones en el experimento. Es privado para encapsularlo y se accede a él mediante los métodos get y set.

poblacionesList: ArrayList<Poblacion>: Una lista de objetos Poblacion que representan las poblaciones en el experimento. Es privado y se accede a él mediante los métodos get y set. La elección de que sea un ArrayList, es por la comodidad y flexibilidad que aporta, al no tener que fijar un número fijo de Poblaciones que tendría un array tradicional. Es importante ya que al crear un experimento con x poblaciones, luego también existe la opción de añadirle más.

Constructor:

Experimento(nombreNuevoExperimento: String): Constructor que inicializa el nombre del experimento.

Métodos get y set:

Se proporcionan métodos get para acceder a los atributos privados nombreExperimento, dias, numPoblaciones y poblacionesList, y métodos set para modificar los valores de nombreExperimento y numPoblaciones.

Métodos adicionales:

void setPoblacionNueva(p: Poblacion): Agrega una nueva población al experimento, mediante el método add(Poblacion P) del ArrayList poblacionesList.

String toString(): Se hace override del método toString() y devuelve una representación de cadena de la información completa del experimento, incluida la información de cada una de sus poblaciones. Esto último lo hace llamando al método toString() de la clase Poblacion, que a su vez llama al método toString de la clase Comida. Así obtenemos toda la información de cada una de las poblaciones, incluida la dosis de comida diaria. Para acceder al toString de todas las poblaciones del experimento en cuestión, se hace empleando un bucle for que recorre el ArrayList poblacionesList y dentro de éste se emplea el método get() de poblacionesList (por ser ArrayList) para acceder a cada una de las poblaciones (gracias a sus índices indicados por el iterador del bucle for), y de ahí llamar al método toString() de cada población.

String toStringNombres(): Devuelve una representación de cadena solo con los nombres de las poblaciones del experimento. De nuevo se emplea un bucle for que recorre el ArrayList poblacionesList y dentro de éste se vuelve a emplear el método get() de poblacionesList y de ahí llamar al método getNombrePoblacion() para cada poblacion del experimento.

String toStringInfoExperimentoToFile(): Devuelve una representación de cadena que puede ser utilizada para escribir información del experimento en un archivo. Cada dato está separado por un “,”.

- **Clase Poblacion:**

Importa:

```
import medio.Comida;  
import medio.Luminosidad;  
import java.time.LocalDate;
```

Esto es necesario ya que hay atributos de la clase Poblacion que son Comida, Luminosidad.luminosidad y LocalDate.

Atributos privados:

```
nombrePoblacion: String  
numInicialBacterias: int  
temperatura: float  
fechaInicio: LocalDate  
fechaFin: LocalDate  
luminosidad: Luminosidad.luminosidad  
comida: Comida  
dosisComidaDiaria: float []:
```

Todos atributos de la clase Poblacion son privados. Esto significa que solo pueden ser accedidos directamente dentro de la misma clase para hacerlo desde fuera, habría que usar los métodos get y set.

Constructor:

La clase Poblacion posee un constructor vacío.

En un inicio también se creó un constructor `public Poblacion(String nombrePoblacion, int numInicialBacterias, float temperatura, LocalDate fechaInicio, Luminosidad.luminosidad luminosidad)`: Es un constructor público que inicializa los atributos de la clase Poblacion. Pero al no ser utilizado en la aplicación, fue eliminado.

Métodos get y set:

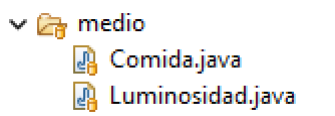
Todos los métodos set son públicos, lo que significa que pueden ser utilizados desde fuera de la clase para establecer los valores de los atributos. Hay método set para todos los atributos de Poblacion, excepto para `float[] dosisComidaDiaria`, ya que se hace directamente desde la clase Comida.

Todos los métodos get excepto `getNombrePoblacion()`, son omitidos intencionalmente, ya que no se proporcionan en esta implementación.

Métodos toString() y toStringInfoPobFile():

Ambos son públicos y permiten obtener representaciones de cadena de la información de la población, ya sea para visualización o para escribir en un archivo. Estos métodos son útiles para la interfaz de usuario y para la captura de datos. De nuevo como en `toStringInfoExperimentoToFile()`, en `toStringInfoPobFile()`, se separan los datos por “;”, para ser fácilmente guardado en un archivo que posteriormente se abrirá.

PAQUETE medio:



Clase Comida:

La clase `Comida` está dentro del paquete `medio`. Esta clase gestiona la comida de cada población de bacterias.

Importa:

```
import java.time.LocalDate;
import static java.time.temporal.ChronoUnit.DAYS;
import java.util.Arrays;
```

El primer import es porque la clase Comida va a manejar las fechas que se guardan como objetos `LocalDate`. El segundo porque en el propio método de Comida, `calcularComida`, se hace uso del método de `LocalDate`, `DAYS.between(LocalDate fecha1, LocalDate fecha2)`, que sirve para calcular los días entre dos fechas.

Atributos privados:

Todos los atributos de esta clase son privados. Lo cual quiere decir que para ser inicializados, ha de ser mediante métodos set.

cantidadInicial: float

cantidadComida: float[]

fechaInicial: LocalDate

fechaFinal: LocalDate

cantidadPico: float

cantidadFinal: float

fechaPico: LocalDate

final duracion: int = 30, en este caso lo hemos puesto final, para que el valor que siempre va a ser 30, no pueda ser cambiado.

Constructores:

Posee dos constructores, uno vacío y otro que inicializa todos los atributos de la clase Comida.

Todos los atributos excepto cantidadComida: float [], se inicializan con el valor que se pasa por el constructor. En el caso de cantidadComida, se inicializa mediante el método de Comida calcularComida, que devuelve un array de floats, con la dosis diaria.

Métodos get y set:

En este caso sólo se hace uso de los setters para todos los atributos de la clase.

Método float [] calcularComida();

Este método tiene como objetivo calcular la cantidad de comida diaria durante un período de tiempo determinado, que es de 30 días que dura el experimento. A este método no se le pasa nada y devuelve un array de float de longitud, la duración del experimento y en cada posición, la cantidad de comida del día correspondiente. Para ello se calcula la comida para cada día mediante una función que incrementa linealmente hasta el día de la fecha pico y de ahí hasta la fecha final, decrementa linealmente. La subida y la baja, se realiza mediante dos bucles for por separado. El primero va de 0 al último día de incremento y el segundo va del día de después del incremento al último día del experimento. Y para saber cuántos días hay entre el inicio y la fecha pico y la fecha pico y la fecha final, se hace uso de DAYS.between(día 1, día 2), que lo tenemos gracias a la importación de java.time.temporal.ChronoUnit.DAYS.

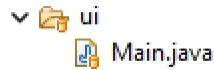
Métodos toString() y toStringToFile():

Ambos son públicos y permiten obtener representaciones de cadena de la información de la comida. De nuevo como en toStringInfoPobToFile(), en toStringToFile (), se separan los datos por “;”, para ser fácilmente guardado en un archivo que posteriormente se abrirá. El método toStringToFile(), es llamado desde toStringInfoPobFile(), mediante el objeto comida de la población en cuestión.

- **Clase Luminosidad:**

Esta clase consiste en un enumerado con las tres distintas posibilidades de luminosidad del medio: {ALTA, MEDIA, BAJA}. No posee ningún atributo ni realiza ningún import.

PAQUETE ui:



○ **Clase Main:**

Esta es la clase principal del programa, es la que posee: `public static void main(String[] args)`, por lo que es la clase que ejecuta el programa.

Importa:

```
import dataio.FileManager;
import static dataio.UserInput.readInt;
import static dataio.UserInput.readString;
import gestionLab.GestionLab;
import laboratorio.Experimento;
import laboratorio.Poblacion;
import java.io.FileNotFoundException;
```

Se importa la clase `FileManager` al completo ya que se hace uso de ambos métodos que esta posee. Lo mismo pasa para la clase `GestionLab`. En el caso del paquete `laboratorio`, se ha importado cada clase por separado, aunque podría haberse hecho de igual manera importando al paquete al completo ya que se usan ambas clases de éste. Y finalmente se importa `FileNotFoundException` que se utiliza en el caso de no encontrar el archivo.

En el caso de la clase `UserInput`, en el `main`, únicamente se hace uso de dos de sus métodos `readInt()` y `readString()`. El primero para elegir la opción del menú y en la opción 3, que es para crear un nuevo experimento, para elegir cuántas poblaciones tiene ese experimento (=> cuantas poblaciones ha de crear) y el segundo para escribir el nombre del experimento o población con el que se quiera “trabajar”. El resto de métodos de la clase `UserInput` no se usan directamente en el `main`. Por ello se hacen esos dos `import` por separado, de ambos métodos. De esta manera también, se “ahorra” el tener que precisar en el código del `main` que el método `readInt()` por ejemplo, pertenece a la clase `UserInput`, mediante la siguiente invocación a cada vez que se vaya a utilizar: `UserInput.readInt()`. En vez de eso, al haber importado el método y no toda la clase de golpe, se le puede invocar simplemente mediante `readInt()`.

Esta clase no lanza ninguna excepción, ya que el `main`, es el último en hacer uso de los métodos.

Se inicializan únicamente dos variables: una variable `int`, que en un principio es inicializada a 0, y esta representa el número de opción del menú y se declara un objeto de tipo `Experimento`, que es inicializado a `null`. Justo después se establece un bucle ``while`` con la condición ``opcion != 9``, lo que garantiza que el programa continúe ejecutándose y mostrando el menú al usuario hasta que seleccione la opción 9. Dentro de este bucle, se lleva a cabo la funcionalidad principal del programa. Se imprime el menú por pantalla. Para asegurar que el usuario selecciona una opción válida, se utiliza un bucle ``do while`` con la condición ``(opcion < 1 || opcion > 9)``, donde se utiliza el método ``readInt()`` para obtener la entrada del usuario y asignarla a la variable ``opcion``. Esta estructura asegura que el programa solicita al usuario que introduzca una opción válida. Si el número

introducido no está en el rango válido (de 1 a 9), se muestra el mensaje "¡Opción no válida!" después del `readInt()`. Esto se logra mediante un condicional `if` que verifica si el número es mayor que 9 o menor que 1, aprovechando que cualquier valor que alcance el `if` será un entero debido a las excepciones manejadas por `readInt()`.

Se sigue con bloque `switch case`, donde cada caso corresponde a una de las opciones del menú. A continuación, se detalla lo que ocurre en cada caso:

- *`case 1`: Abrir un archivo que contenga un experimento.*

Se establece un bloque `try-catch`. En el bloque `try`, se inicializa el nombre del experimento que se desea abrir, el cual es una cadena de texto (`String`), mediante el método `readString()`. Además, se inicializa la variable `experimento` utilizando un método de la clase `FileManager` llamado `abrirArchivo()`, al cual se le pasa como parámetro el nombre del experimento que se desea abrir. En caso de que ocurra algún error, se capturan dos excepciones: `FileNotFoundException`, en la cual se imprime por pantalla "No se ha encontrado el archivo.", y otra excepción del tipo `Exception`, en la cual se imprime por pantalla "ERROR.". Finalmente, se utiliza un `break` para salir del `case 1`.

- *`case 2`: "Crear un nuevo experimento"*

Se solicita al usuario que ingrese el nombre del nuevo experimento utilizando el método `readString()` para leer una cadena de texto desde la entrada estándar. A continuación, se solicita al usuario que ingrese el número de poblaciones que tendrá el nuevo experimento utilizando el método `readInt()` para leer un número entero desde la entrada estándar. Se utiliza un bucle `while` (`true`) para garantizar que el usuario ingrese un valor válido. Si el número de poblaciones es menor que 0, se muestra un mensaje de error, ya que el número de poblaciones no puede ser negativo y se vuelve a solicitar al usuario que ingrese un valor válido. Una vez que se ingresa un valor válido, el bucle se interrumpe con `break`. Se crea un nuevo objeto `Experimento` con el nombre proporcionado por el usuario. Se utiliza un bucle `for` para crear las poblaciones especificadas por el usuario. En cada iteración del bucle, se muestra un mensaje indicando el número de población que se está creando. Luego, se intenta crear la población utilizando el método `createPoblacion()` de la clase `GestionLab`, pasando como argumento el objeto `experimento` recién creado. Si ocurre algún error durante la creación de la población, se muestra un mensaje de error y se imprime la pila de llamadas (`stack trace`) para ayudar a depurar el problema. Una vez creadas todas las poblaciones, se muestra un mensaje indicando que el experimento ha sido creado correctamente, seguido de una representación en cadena del objeto `experimento` utilizando el método `toString()`. Y finalmente sale del `case` con un `break`.

- *`case 3`: Crear una población de bacterias y añadirla al experimento actual*

Se verifica mediante un bucle `if else` si hay un experimento cargado en memoria comprobando si el objeto `Experimento` `experimento` es `null`.

Si no hay ningún experimento cargado, se muestra un mensaje indicando que primero se debe abrir un archivo y luego se sale del `case` con `break`.

Si hay un experimento cargado en memoria, se intenta crear una nueva población de bacterias utilizando el método `createPoblacion()` de la clase `GestionLab`, pasando como argumento el `experimento` actual. El resultado de esta operación se asigna a un objeto `Poblacion` llamado `recienCreada`. Se muestra en consola la representación en cadena de la población recién creada utilizando el método `toString()`. Si ocurre algún error durante la creación de la población, se captura la excepción y se muestra un mensaje de error genérico "ERROR.". Y finalmente sale del `case` con un `break`.

- *`case 4`: Mostrar nombre poblaciones*

Se verifica mediante un bucle ``if else`` si hay un experimento cargado en memoria comprobando si el objeto Experimento `experimento` es null.

Si no hay ningún experimento cargado, se muestra un mensaje indicando que primero se debe abrir un archivo y luego se sale del case con `break`.

Si hay un experimento cargado, se llama al método `toStringNombres()` del Experimento `experimento`, que devuelve una cadena que contiene los nombres de todas las poblaciones asociadas al experimento. La cadena resultante se imprime en la consola. Y finalmente sale del case con un `break`.

- *`case 5`: Borrar una población de bacterias del experimento actual*

Se verifica mediante un bucle ``if else`` si hay un experimento cargado en memoria comprobando si el objeto Experimento `experimento` es null.

Si no hay ningún experimento cargado, se muestra un mensaje indicando que primero se debe abrir un archivo y luego se sale del case con `break`.

Si hay un experimento cargado, se solicita al usuario que ingrese el nombre de la población que desea eliminar utilizando el método `readString()` para leer una cadena de texto teclado. Se llama al método `deletePoblacion()` de la clase `GestionLab`, pasando como argumentos el nombre de la población a eliminar y el experimento actual. Este método se encarga de eliminar la población especificada del experimento. Se muestra un mensaje indicando que la población ha sido eliminada correctamente. Y finalmente sale del case con un `break`.

- *`case c`: Ver información detallada de una población de bacterias del experimento actual*

Se verifica mediante un bucle ``if else`` si hay un experimento cargado en memoria comprobando si el objeto Experimento `experimento` es null.

Si no hay ningún experimento cargado, se muestra un mensaje indicando que primero se debe abrir un archivo y luego se sale del case con `break`.

Si hay un experimento cargado, se solicita al usuario que ingrese el nombre de la población de la cual desea ver la información detallada utilizando el método `readString()` para leer una cadena de texto de teclado. Se intenta buscar la población especificada utilizando el método `buscarPoblacion()` de la clase `GestionLab`, pasando como argumentos el nombre de la población y el experimento actual. Este método devuelve un objeto `Poblacion` si la población es encontrada en el experimento.

Si se encuentra la población, se muestra en consola la representación en cadena de la población encontrada utilizando el método `toString()` de la clase `Poblacion`. Esta representación contiene la información detallada de la población de bacterias.

Si la población no es encontrada en el experimento, se captura la excepción y se muestra un mensaje indicando que la población no existe en el experimento. Finalmente sale del case con un `break`.

- *`case 7`: Guardar (se supone que para usar esta opción previamente hemos abierto un archivo)*

Se verifica mediante un bucle ``if else`` si hay un experimento cargado en memoria comprobando si el objeto Experimento `experimento` es null.

Si no hay ningún experimento cargado, se muestra un mensaje indicando que primero se debe abrir un archivo y luego se sale del case con break.

Si hay un experimento cargado, se llama al método guardarArchivo() de la clase FileManager, pasando como argumentos el nombre del experimento y el objeto experimento actual. Este método se encarga de guardar el experimento en un archivo. El método guardarArchivo() devuelve un valor booleano que indica si el guardado fue exitoso o no.

Si el guardado fue exitoso (guardado == true), se muestra un mensaje indicando que el experimento ha sido guardado correctamente.

Si el guardado no fue exitoso (guardado == false), se muestra un mensaje indicando que ha ocurrido un fallo al guardar el experimento.

Finalmente se sale del case con un break.

- *`case 8`: Guardar como*

Se verifica mediante un bucle `if else` si hay un experimento cargado en memoria comprobando si el objeto Experimento experimento es null.

Si no hay ningún experimento cargado, se muestra un mensaje indicando que primero se debe abrir un archivo y luego se sale del case con break.

Si hay un experimento cargado, se solicita al usuario que ingrese el nuevo nombre para el experimento utilizando el método readString() para leer una cadena de texto introducida por teclado.

Se actualiza el nombre del experimento actual utilizando el nuevo nombre proporcionado por el usuario mediante el método setNombreExperimento().

Se llama al método guardarArchivo() de la clase FileManager, pasando como argumentos el nuevo nombre del experimento y el objeto experimento actual. Este método se encarga de guardar el experimento con el nuevo nombre en un archivo. El método guardarArchivo() devuelve un valor booleano que indica si el guardado fue exitoso o no.

Si el guardado fue exitoso (guardadoComo == true), se muestra un mensaje indicando que el experimento ha sido guardado correctamente con el nuevo nombre.

Si el guardado no fue exitoso (guardadoComo == false), se muestra un mensaje indicando que ha ocurrido un fallo al guardar el experimento con el nuevo nombre.

Finalmente se sale del case con un break.

- *`case 5`: Salir del programa*

Simplemente se imprime por pantalla un aviso de que salió del programa y un "Adiós y muchas gracias."

Finalmente se sale del case con un break.

- LISTADO DE FALLOS CONOCIDOS Y FUNCIONALIDADES DEFINIDAS EN EL ENUNCIADO QUE NO SE HAN IMPLEMENTADO EN EL CÓDIGO ENTREGADO.

Seguramente haga falta una mejor implementación de los distintos niveles de acceso tanto de atributos como de métodos.

Para reducir mínimamente el código, podría por ejemplo en la clase `UserInput` dentro del paquete `dataio`, para cada método, haber inicializado boolean `hecho` a `false` y únicamente cambiarle el valor a `true` al conseguir el objetivo deseado, en vez de declararlo como `false` a cada vez que no sea así.

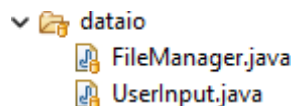
Se podría haber creado una clase excepciones y haber gestionado desde ahí, asuntos como por ejemplo la cantidad máxima de comida. Es un punto por mejorar de cara al futuro.

Además, se le podría haber implementado un sistema de ordenación por orden alfabético por ejemplo de las poblaciones.

En el caso de los `UnitTests` me he visto con bastantes dificultades y eso es algo a mejorar para la próxima. Por ejemplo en `GestionLabTest`: no he conseguido hacer el `unitTest` de `createPoblacion()` de `GestionLab` porque en el método le pido por pantalla varias variables y para ello se hace por ejemplo `int.nextInt()` entonces digamos que en la primera pedida por pantalla se "come" todo el `provided input`, a pesar de haberlo intentado separar por saltos de línea. He estado literal como CUATRO horas con esto y no he encontrado solución.

- LISTADO DE TODO EL CÓDIGO FUENTE DE LA APLICACIÓN ORGANIZADO POR PAQUETES (SI APLICA) Y CLASES.

PAQUETE `dataio`:



○ **Clase *FileManager*:**

```
package dataio;
import gestionLab.GestionLab;
import laboratorio.Experimento;
import laboratorio.Poblacion;
import medio.Comida;
import medio.Luminosidad;
import java.io.*;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
```

```
/**
 * Gestion de archivos
 * @author Ana Ventura-Traveset
 */
public class FileManager {

    /**
     * Abre archivo y lo carga en memoria
     * @param nombreExperimento
     * @return experimento
     * @throws FileNotFoundException
     */
```

```

public static Experimento abrirArchivo(String nombreExperimento) throws
FileNotFoundException {
    File file = new File("./" + nombreExperimento + ".txt");
    Experimento experimento = null;

    FileInputStream fileInputStream = null;
    InputStreamReader inputStreamReader = null;
    BufferedReader bufferedReader = null;
    String stringInfoTotal="";
    try {
        // lo leo
        bufferedReader = new BufferedReader(new InputStreamReader(new FileInputStream(file)));
        String [] todosArgs = bufferedReader.readLine().split("\n");
        String [] infoExperimento = todosArgs[0].split(";");
        String nombreExpFromFile = infoExperimento[0];
        int diasExpFromFile = Integer.parseInt(infoExperimento[1]);

        experimento = new Experimento(nombreExpFromFile);

        System.out.println(todosArgs.length+ "mi length");
        stringInfoTotal+=experimento.toStringInfoExperimentoToFile()+"\n";

        String line;
        while ((line = bufferedReader.readLine()) != null) {
            String [] infoPoblacion = line.split(";");

            Poblacion poblacion = new Poblacion();

            //Empiezo con la info de poblaciones
            String nombrePoblacionFromFile = infoPoblacion[0];
            poblacion.setNombrePoblacion(nombrePoblacionFromFile);

            int numBacteriasFromFile = Integer.parseInt(infoPoblacion[1]);
            poblacion.setNumInicialBacterias(numBacteriasFromFile);

            float temperaturaFromFile = Float.parseFloat(infoPoblacion[2]);
            poblacion.setTemperatura(temperaturaFromFile);

            Luminosidad.luminosidad luminosidadFromFile =
Luminosidad.luminosidad.valueOf(infoPoblacion[3]);
            poblacion.setLuminosidad(luminosidadFromFile);

            // Empiezo con la info de comida
            DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd");
            LocalDate fechaInicioFromFile = LocalDate.parse(infoPoblacion[4], dtf);
            poblacion.setFechaInicio(fechaInicioFromFile);
            float cantidadInicialFromFile = Float.parseFloat(infoPoblacion[5]);
            LocalDate fechaPicoFromFile = LocalDate.parse(infoPoblacion[6], dtf);
            float cantidadPicoFromFile = Float.parseFloat(infoPoblacion[7]);
            LocalDate fechaFinFromFile = LocalDate.parse(infoPoblacion[8], dtf);
            poblacion.setFechaFin(fechaFinFromFile);
            float cantidadFinalFromFile = Float.parseFloat(infoPoblacion[9]);

```

```

        Comida comida = new Comida(cantidadInicialFromFile, fechaInicioFromFile,
cantidadPicoFromFile, fechaPicoFromFile, cantidadFinalFromFile, fechaFinFromFile);
        poblacion.setComida(comida);
        GestionLab.addPoblacion(poblacion,experimento);
        stringInfoTotal+=poblacion.toStringInfoPobFile()+"\n";
    }
    System.out.println("\nFICHERO CARGADO EN MEMORIA\n");
    System.out.println(stringInfoTotal);
    bufferedReader.close();
}catch (Exception e){
    System.out.println("ERROR FileManager leyendo archivo. Puede que no exista ningún
archivo con ese nombre");
}
finally {
    if (bufferedReader != null){
        try{
            bufferedReader.close();
        }catch(IOException ioe){
            System.out.println(ioe.getMessage());
        }
    }
}

if(inputStreamReader != null){
    try{
        inputStreamReader.close();
    }
    catch(IOException ioException){
        System.out.println(ioException.getMessage());
    }
}
if (fileInputStream != null) {
    try {
        fileInputStream.close();
    } catch (IOException ex) {
        System.out.println(ex.getMessage());
    }
}
}
return experimento;
}

```

/**

* Guarda/guarda como experimento en archivo

* @param nombreExperimento

* @param experimento

* @return comprobacion

*/

```
public static boolean guardarArchivo(String nombreExperimento, Experimento experimento) {
```

```
    File file1 = new File( "/" + nombreExperimento + ".txt");
```

```
    PrintWriter printWriter = null;
```

```

        boolean comprobacion=false;
        try {
            printWriter = new PrintWriter(file1);
            String experimentoInfoFile = experimento.getNombreExperimento() + ';' +
experimento.getDias();
            printWriter.println(experimentoInfoFile);//escribe en el fichero primero la info del
experimento
            for (int i = 0; i < experimento.getPoblacionesList().size(); i++) {
                String infoPoblacionesFile = "";
                infoPoblacionesFile += experimento.getPoblacionesList().get(i).toStringInfoPobFile();
                printWriter.print(infoPoblacionesFile); //escribe en el fichero ahora la info de cada
población
                printWriter.println();
            }
            printWriter.close();
            comprobacion=true;
        } catch (IOException e) {
            e.printStackTrace();
            comprobacion=false;
        } finally {
            if (printWriter != null) {
                printWriter.close();
            }
        }
        return comprobacion;
    }
}

```

○ Clase UserInput:

```

package dataio;
import medio.Luminosidad;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.time.LocalDate;
import java.time.ZoneId;
import java.util.Date;
import java.util.Scanner;

/**
 * Clase para leer de teclado
 * @author Ana Ventura-Traveset
 */
public class UserInput {

    /**
     * Para leer por teclado Strings
     * @param petition
     * @return miString
     */
    public static String readString(String petition) {
        System.out.println(petition);
        boolean hecho;
        String miString = "";
    }
}

```

```

do {
    try {
        Scanner sc = new Scanner(System.in); //creo el scanner Object
        miString = sc.nextLine(); //leo el input del usuario
        hecho = true;
    } catch (Exception e) {
        System.out.println("ERROR al introducir por teclado.");
        e.printStackTrace();
        hecho = false;
    }
} while (hecho == false);
return miString;
}

/**
 * Para leer por teclado enteros, excluyendo los negativos que no se usan en esta práctica
 * @param peticion
 * @return miInt
 */
public static int readInt(String peticion) {
    boolean hecho;
    int miInt = 0;
    do {
        try {
            System.out.println(peticion);
            Scanner sc = new Scanner(System.in);
            miInt = sc.nextInt();
            if (miInt < 0) {
                System.out.println("ERROR. El número introducido es negativo.");
                hecho=false;
            } else {
                hecho = true;
            }
        } catch (Exception e) {
            System.out.println("ERROR al introducir por teclado. Debe introducir un número entero.");
            hecho = false;
        }
    } while (hecho == false);
    return miInt;
}

/**
 * Para leer por teclado reales
 * @param peticion
 * @return miFloat
 */
public static float readFloat(String peticion) {
    boolean hecho;
    float miFloat = 0;
    do {
        try {
            System.out.println(peticion);

```

```

        Scanner sc = new Scanner(System.in);
        miFloat = sc.nextFloat();
        hecho=true;
    } catch (Exception e) {
        System.out.println("ERROR al introducir por teclado. Debe introducir un número real.");
        hecho = false;
    }
} while (hecho == false);
return miFloat;
}

/**
 * Para leer por teclado un String y asociarlo con un elemento del enum luminosidad
 * @param peticion
 * @return lum
 */
public static Luminosidad.luminosidad readLuminosidad(String peticion) {
    Luminosidad.luminosidad luminosidad = null;
    String lum;
    boolean hecho;
    do{
        try {
            System.out.println(peticion);
            Scanner sc = new Scanner(System.in);
            lum = sc.nextLine();
            if (lum.equalsIgnoreCase("ALTA")) {
                luminosidad = Luminosidad.luminosidad.ALTA;
                hecho=true;
            } else if (lum.equalsIgnoreCase("MEDIA")) {
                luminosidad = Luminosidad.luminosidad.MEDIA;
                hecho=true;
            } else if (lum.equalsIgnoreCase("BAJA")) {
                luminosidad = Luminosidad.luminosidad.BAJA;
                hecho=true;
            } else {
                System.out.println("ERROR. Por favor introduzca una luminosidad correcta {ALTA,
MEDIA, BAJA}: ");
                hecho=false;
            }
        } catch (Exception e) {
            System.out.println("ERROR al introducir por teclado.");
            hecho=false;
        }
    }while (hecho == false) ;
    return luminosidad;
}

/**
 * Para leer por teclado fechas
 *
 * SimpleDateFormat tiene un método para convertir String
 * en el formato fecha que le hayamos dicho, y ese es el parse

```






```

* se utiliza el método toInstant de Date().atZone().toLocalDate() para pasar de Date a LocalDate
*
* @param petition
* @return fechaADevolver
*/
public static LocalDate readDate(String petition){
    Date fecha;
    LocalDate fechaADevolver=null;
    boolean hecho;

    do{
        try{
            System.out.println(petition);
            System.out.println("Introducir las fechas en este formato: yyyy.MM.dd");
            Scanner sc = new Scanner(System.in);
            SimpleDateFormat formato = new SimpleDateFormat("yyyy.MM.dd");
            fecha = formato.parse(sc.nextLine());
            fechaADevolver = fecha.toInstant().atZone(ZoneId.systemDefault()).toLocalDate();
            hecho=true;
        }catch(ParseException pe){
            hecho = false;
            System.out.println("ERROR. La fecha introducida no es correcta o no se ha parseado correctamente.");
        }
    }while(hecho==false);
    return fechaADevolver;
}
}

```

PAQUETE gestionLab:

  gestionLab
 GestionLab.java

○ **Clase GestionLab:**

```

package gestionLab;
import dataio.UserInput;
import laboratorio.Experimento;
import laboratorio.Poblacion;
import medio.*;
import java.text.ParseException;
import java.time.LocalDate;
import static java.time.temporal.ChronoUnit.DAYS;

/**
 * @author Ana Ventura-Traveset
 */

/**

```

```

* Gestion del laboratorio (crear, añadir, borrar y buscar poblaciones)
*/
public class GestionLab {

    /**
     *
     * CREAR población
     * @param e
     * @return
     * @throws Exception
     * @throws ParseException
     */
    public static Poblacion createPoblacion(Experimento e) throws Exception, ParseException {
        Poblacion p= new Poblacion();
        int dias=30;
        float comidaInicial;
        float comidaPico;
        float comidaFinal;
        Comida comida=new Comida();

        String nombreP = UserInput.readString("Escriba el nombre de su nueva población: ");
        p.setNombrePoblacion(nombreP);

        float temp = UserInput.readInt("Escriba la temperatura: ");
        p.setTemperatura(temp);

        Luminosidad.luminosidad lum = UserInput.readLuminosidad("Escriba el nivel de luminosidad
{ALTA, MEDIA, BAJA}: ");
        p.setLuminosidad(lum);

        while (true) {
            comidaInicial = UserInput.readFloat("Introduzca la cantidad de comida inicial: ");
            if (comidaInicial < 0) {
                System.out.println("La cantidad de comida no puede ser negativa.");
            } else if (comidaInicial > 300) {
                System.out.println("La cantidad de comida no puede ser superior a 300.");
            } else {
                comida.setCantidadInicial(comidaInicial);
                break;
            }
        }
        while (true) {
            comidaPico = UserInput.readFloat("Introduzca la cantidad de comida más alta: ");
            if (comidaPico < 0) {
                System.out.println("La cantidad de comida no puede ser negativa.");
            } else if (comidaPico > 300) {
                System.out.println("La cantidad de comida no puede ser superior a 300.");
            } else {
                comida.setCantidadPico(comidaPico);
                break;
            }
        }
    }
}

```

```

while (true) {
    comidaFinal = UserInput.readFloat("Introduzca la cantidad de comida final: ");
    if (comidaFinal < 0) {
        System.out.println("La cantidad de comida no puede ser negativa.");
    } else if (comidaFinal > 300) {
        System.out.println("La cantidad de comida no puede ser superior a 300.");
    } else {
        comida.setCantidadFinal(comidaFinal);
        break;
    }
}

//pido el resto de cosas para calcular la comida, que son las fechas
LocalDate fechaInicial;
LocalDate fechaMedia;
LocalDate fechaFinal;

//Para controlar que fecha media no sea antes que la de inicio ni después que la de fin
fechaInicial= UserInput.readDate("Introduzca la fecha dónde empieza su experimento: ");
while (true) {
    fechaMedia = UserInput.readDate("Introduzca la fecha dónde hay el pico en su experimento:
");
    fechaFinal=fechaInicial.plusDays(dias);
    int diasEntreInicioYPico= (int) DAYS.between(fechaInicial, fechaMedia);
    if (fechaMedia.isBefore(fechaInicial)) {
        System.out.println("La fecha introducida no es correcta. " +
            "\nNo puede ser la fecha pico antes de la fecha inicial del experimento." );
    }
    else if(fechaMedia.isAfter(fechaFinal)){
        System.out.println("La fecha introducida no es correcta. " +
            "\nLa fecha pico no puede ser después de la fecha final del experimento que dura 30
días." );
    }
    else if((diasEntreInicioYPico>29)){
        System.out.println("La fecha introducida no es correcta. " +
            "\nLa fecha pico no puede ser después de la fecha final del experimento que dura 30
días." );
    }
    else {
        comida.setFechaInicial(fechaInicial);
        p.setFechaInicio(fechaInicial);
        comida.setFechaPico(fechaMedia);
        comida.setFechaFinal(fechaFinal);
        p.setFechaFin(fechaFinal);

        comida.setCantidadComida(comida.calcularComida());
        p.setComida(comida);
        break;
    }
}
}

```

```

int numIniBact;
while (true) {
    numIniBact = UserInput.readInt("Escriba el número inicial de bacterias: ");
    if (numIniBact < 0) {
        System.out.println("El número inicial de bacterias no puede ser negativo.");
    } else {
        p.setNumInicialBacterias(numIniBact);
        break;
    }
}
System.out.println("Creada población: "+nombreP);

//añado la poblacion al experimento
e.setNumPoblaciones(e.getNumPoblaciones()+1);
e.setPoblacionNueva(p);
return p;
}

/**
 *
 * AÑADIR poblacion al experimento
 * @param pob
 * @param exp
 */
public static void addPoblacion (Poblacion pob, Experimento exp){
    exp.setNumPoblaciones(exp.getNumPoblaciones()+1);
    exp.setPoblacionNueva(pob);
}

/**
 *
 * ELIMINAR población del experimento
 * @param nombrePoblacion
 * @param e
 */
public static void deletePoblacion(String nombrePoblacion, Experimento e) {

    for (int i = 0; i < e.getPoblacionesList().size(); i++) {
        if (nombrePoblacion.equals(e.getPoblacionesList().get(i).getNombrePoblacion())) {
            e.getPoblacionesList().remove(i); // Si lo igualo a algo, me va a mandar la Poblacion que he
quitado
            // si no hago nada más, simplemente le arranca esa poblacion de mi lista
        }
    }
    e.setNumPoblaciones(e.getPoblacionesList().size());
}

public static Poblacion buscarPoblacion(String nombrePoblacion, Experimento e) {
    Poblacion miPoblacion=null;
    for (int i = 0; i < e.getPoblacionesList().size(); i++) {
        if (nombrePoblacion.equals(e.getPoblacionesList().get(i).getNombrePoblacion())) {

```

```

        miPoblacion=e.getPoblacionesList().get(i);
        break;
    }
}
if(miPoblacion!=null){
    return miPoblacion;
}
else {
    throw new RuntimeException();
}
}
}

```

PAQUETE laboratorio:

```

v laboratorio
  Experimento.java
  Poblacion.java

```

○ **Clase Experimento:**

```

package laboratorio;
import java.util.ArrayList;

/**
 * @author Ana Ventura-Traveset
 */
public class Experimento {
    /**
     * Atributos experimento
     *
     * nombreExperimento
     * dias
     * numPoblaciones
     * poblacionesList
     */
    private String nombreExperimento;
    private final int dias = 30;
    private int numPoblaciones;
    private ArrayList<Poblacion> poblacionesList;

    /**
     * Constructor de experimento
     * @param nombreNuevoExperimento
     */
    public Experimento(String nombreNuevoExperimento){
        this.nombreExperimento=nombreNuevoExperimento;
        this.poblacionesList = new ArrayList<Poblacion>();
    }
}

```

```

/**
 * Getters y setters de los atributos privados de Experimento
 *
 */
//Getters y setters

/**
 * getNombreExperimento
 * @return String nombreExperimento
 */
public String getNombreExperimento() {
    return nombreExperimento;
}

/**
 * setNombreExperimento
 * @param nombreExperimento
 */
public void setNombreExperimento(String nombreExperimento) {
    this.nombreExperimento = nombreExperimento;
}

/**
 * getDias
 * @return int dias
 */
public int getDias() {
    return dias;
}

/**
 * getNumPoblaciones
 * @return int numPoblaciones
 */
public int getNumPoblaciones() {
    return numPoblaciones;
}

/**
 * setNumPoblaciones
 * @param numPoblaciones
 */
public void setNumPoblaciones(int numPoblaciones) {
    this.numPoblaciones = numPoblaciones;
}

/**
 * getPoblacionesList()
 * @return ArrayList<Poblacion> this.poblacionesList: la lista de Poblaciones
 */
public ArrayList<Poblacion> getPoblacionesList() {

```

```

        return this.poblacionesList;
    }

    /**
     * setPoblacionNueva
     * Añade la población a la lista de poblaciones
     * @param p
     */
    public void setPoblacionNueva (Poblacion p){
        this.poblacionesList.add(p);
    }

    // Métodos para que al meter esta clase en un System.out.println() salga
    // algo legible

    /**
     * toString
     * Este nos enseña TODA la info del experimento
     * @return
     */
    @Override
    public String toString() {
        String stringToRepresentInfoPoblacionesExperimento = "Nombre Experimento: " +
this.nombreExperimento
            + "\nNumero de días: " + this.dias
            + "\n\nInformación de las poblaciones:\n";
        if(poblacionesList!=null) {
            for (int i = 0; i < poblacionesList.size(); i++) {
                stringToRepresentInfoPoblacionesExperimento += "\nPOBLACIÓN " + (i + 1) + ": " +
poblacionesList.get(i).toString()+"\n";
            }
        }
        return stringToRepresentInfoPoblacionesExperimento;
    }

    /**
     * Muestra el nombre de todas las poblaciones del experimento
     * @return toStringNombres(
     */
    public String toStringNombres() {
        String stringToRepresentNombrePoblacionesExperimento = "Nombre Experimento: " +
this.nombreExperimento
            + "\nNombre de poblaciones:\n";
        for (int i=0; i<this.poblacionesList.size() ; i++) {
            stringToRepresentNombrePoblacionesExperimento += "Nombre de la poblacion " + (i + 1) +
": " + poblacionesList.get(i).getNombrePoblacion()+"\n";
        }
        return stringToRepresentNombrePoblacionesExperimento;
    }

    /**

```

```

    * Muestra la información del experimento separada por ";" (modo fichero)
    * @return
    */
    public String toStringInfoExperimentoToFile(){
        String stringToRepresentInExperimentobFile = this.nombreExperimento;
        return stringToRepresentInExperimentobFile;
    }
}

```

- **Clase Poblacion:**

```

package laboratorio;
import medio.Comida;
import medio.Luminosidad;
import java.time.LocalDate;

/**
 * @author Ana Ventura-Traveset
 */
public class Poblacion {

    /**
     * Atributos poblacion
     *
     * nombrePoblacion
     * numInicialBacterias
     * temperatura
     * fechaInicio
     * fechaFin
     * luminosidad
     * comida
     * dosisComidaDiaria
     */
    private String nombrePoblacion;
    private int numInicialBacterias;
    private float temperatura;
    private LocalDate fechaInicio, fechaFin;
    private Luminosidad.luminosidad luminosidad;
    private Comida comida;
    private float [] dosisComidaDiaria;

    /**
     * Constructor vacío de poblacion
     */
    public Poblacion(){
    }

    /**
     * Getters y setters
     * @return
     */
}

```



```

/**
 * Muestra el nombre de la poblacion
 *
 * @return String
 */
public String getNombrePoblacion() {
    return nombrePoblacion;
}

/**
 * Permite modificar el nombre de la Población
 * @param nombrePoblacion
 */
public void setNombrePoblacion(String nombrePoblacion) {
    this.nombrePoblacion = nombrePoblacion;
}

/**
 * Permite modificar el número inicial de bacterias de la Población
 * @param numInicialBacterias
 */
public void setNumInicialBacterias(int numInicialBacterias) {
    this.numInicialBacterias = numInicialBacterias;
}

/**
 * Permite modificar la temperatura de la Población
 * @param temperatura
 */
public void setTemperatura(float temperatura) {
    this.temperatura = temperatura;
}

/**
 * Permite modificar la fecha de inicio del experimento de la Población
 * @param fechaInicio
 */
public void setFechaInicio(LocalDate fechaInicio) {
    this.fechaInicio = fechaInicio;
}

/**
 * Permite modificar la fecha de fin del experimento de la Población
 * @param fechaFin
 */
public void setFechaFin(LocalDate fechaFin) {
    this.fechaFin = fechaFin;
}

/**
 * Permite modificar la luminosidad de la Población
 * @param luminosidad

```

```

    */
    public void setLuminosidad(Luminosidad.luminosidad luminosidad) {
        this.luminosidad = luminosidad;
    }


    /**
     * Permite modificar la comida de la Población
     * @param comida
     */
    public void setComida(Comida comida) {
        this.comida = comida;
        this.dosisComidaDiaria=comida.calcularComida();
    }


    /**
     * toString para cuando se seleccione la opción 6, para visualizar la info de la población
     * @return stringToRepresentPoblacion
     */
    @Override
    public String toString(){
        String stringToRepresentPoblacion = "La población "
            + this.nombrePoblacion + ":"
            + "\nCantidad de bacterias inicialmente: " + this.numInicialBacterias
            + "\nTemperatura a la cual están sometidas las bacterias: " + this.temperatura
            + "\nLuminosidad: " + this.luminosidad
            + "\nDosis de comida diaria: " + this.comida.toString();
        return stringToRepresentPoblacion;
    }


    /**
     * Muestra la información de la población separada por ";" (modo fichero)
     * @return stringToRepresentInfoPobFile
     */
    public String toStringInfoPobFile() {
        String stringToRepresentInfoPobFile = this.nombrePoblacion
            + ";" + Integer.toString(this.numInicialBacterias)
            + ";" + this.temperatura
            + ";" + this.luminosidad
            + ";" + comida.toStringToFile();
        return stringToRepresentInfoPobFile;
    }
}

```

PAQUETE medio:

 medio

 Comida.java

 Luminosidad.java

- **Clase Comida:**

```
package medio;
import java.time.LocalDate;
import static java.time.temporal.ChronoUnit.DAYS;
import java.util.Arrays;

/**
 * @author Ana Ventura-Traveset
 */
public class Comida {

    /**
     * Atributos de Comida
     */
    * cantidadInicial
    * cantidadComida
    * fechaInicial
    * fechaFinal
    * cantidadPico
    * cantidadFinal
    * fechaPico
    * duracion
    */
    private float cantidadInicial;
    private float [] cantidadComida;
    private LocalDate fechaInicial;
    private LocalDate fechaFinal;
    private float cantidadPico;
    private float cantidadFinal;
    private LocalDate fechaPico;
    private final int duracion=30;

    /**
     * Constructor vacío de Comida
     */
    public Comida() {

    }

    /**
     * Constructor de Comida
     * @param cantidadInicial
     * @param fechaInicial
     * @param cantidadPico
     * @param fechaPico
     * @param cantidadFinal
     * @param fechaFinal
     */
}
```

```

    public Comida (float cantidadInicial, LocalDate fechaInicial, float cantidadPico, LocalDate
fechaPico, float cantidadFinal, LocalDate fechaFinal){
        this.cantidadInicial=cantidadInicial;
        this.fechaInicial=fechaInicial;
        this.cantidadPico=cantidadPico;
        this.fechaPico=fechaPico;
        this.cantidadFinal=cantidadFinal;
        this.fechaFinal=fechaFinal;
        this.cantidadComida = this.calcularComida();
    }

    /**
     * Permite modificar la cantidad inicial de comida
     * @param cantidadInicial
     */
    public void setCantidadInicial(float cantidadInicial) {
        this.cantidadInicial = cantidadInicial;
    }

    /**
     * Permite modificar la cantidad diaria de comida
     * @param cantidadComida
     */
    public void setCantidadComida(float[] cantidadComida){this.cantidadComida=cantidadComida;}

    /**
     * Permite modificar la fecha inicial cuando se empieza el experimento
     * @param fechaInicial
     */
    public void setFechaInicial(LocalDate fechaInicial) {
        this.fechaInicial = fechaInicial;
    }

    /**
     * Permite modificar la fecha final cuando se acaba el experimento
     * @param fechaFinal
     */
    public void setFechaFinal(LocalDate fechaFinal) {
        this.fechaFinal = fechaFinal;
    }

    /**
     * Permite modificar la cantidad pico de comida
     * @param cantidadPico
     */
    public void setCantidadPico(float cantidadPico) {
        this.cantidadPico = cantidadPico;
    }

```

```

/**
 * Permite modificar la cantidad final de comida
 * @param cantidadFinal
 */
public void setCantidadFinal(float cantidadFinal) {
    this.cantidadFinal = cantidadFinal;
}

/**
 * Permite modificar la fecha en la que se produce el pido de comida
 * @param fechaPico
 */
public void setFechaPico(LocalDate fechaPico) {
    this.fechaPico = fechaPico;
}

/**
 * Calcula la cantidad de comida diaria
 * @return
 */
public float[] calcularComida(){
    int diasIncremento = (int) DAYS.between(this.fechaInicial, this.fechaPico);
    int diasDecremento = (int) DAYS.between(this.fechaPico, this.fechaFinal);
    float interseccion= cantidadPico-((cantidadPico-this.cantidadFinal)/diasDecremento);
    float CantidadIncremento= this.cantidadPico - this.cantidadInicial;
    float CantidadDecremento= this.cantidadPico - this.cantidadFinal;
    float cantidadComida[]=new float [duracion];

    for (int i=0; i<diasIncremento; i++){
        cantidadComida[i]=((CantidadIncremento)/diasIncremento)*i+this.cantidadInicial;
    }
    for(int j=diasIncremento; j<duracion; j++){
        cantidadComida[j]= ((CantidadDecremento)/diasDecremento)*j+interseccion;
    }
    return cantidadComida;
}

/**
 * Muestra la información de la comida, incluida la cantidad de comida de cada día
 * @return stringToRepresentComida
 */
@Override
public String toString(){
    String stringToRepresentComida =
        "En la fecha de inicio: "+this.fechaInicial +", cantidad de comida inicial: " +
this.cantidadInicial
        + "\nEn la fecha de pico: "+this.fechaPico +", cantidad de comida pico: " +
this.cantidadPico
        + "\nEn la fecha de fin: "+this.fechaFinal +", cantidad de comida final: " + this.cantidadFinal
        + "\nCantidad de dosis de comida diaria: " + Arrays.toString(cantidadComida);
    return stringToRepresentComida;
}

```

```

    }

    /**
     * Muestra la información de la comida separada por ";" (modo fichero)
     * @return stringToRepresentDosisComidaToFile
     */
    public String toStringToFile(){
        String stringToRepresentDosisComidaToFile = this.fechaInicial
        + ";" + this.cantidadInicial
        + ";" + this.fechaPico
        + ";" + this.cantidadPico
        + ";" + this.fechaFinal
        + ";" + this.cantidadFinal;
        return stringToRepresentDosisComidaToFile;
    }
}

```

- **Clase Luminosidad:**

package medio;



```


/**
 * @author Ana Ventura-Traveset
 */
public class Luminosidad {

    /**
     * enumerado de luminosidad
     */
    public enum luminosidad {
        ALTA, MEDIA, BAJA
    };
}

```

PAQUETE ui:


 ui

 Main.java

- **Clase Main:**

```

package ui;
import dataio.FileManager;
import static dataio.UserInput.readInt;
import static dataio.UserInput.readString;
import gestionLab.GestionLab;
import laboratorio.Experimento;
import laboratorio.Poblacion;
import java.io.FileNotFoundException;

```

```

/**
 * @author Ana Ventura-Traveset

```

*

* Esta es la clase principal que contiene el método main.

* Se utiliza como punto de entrada para la aplicación.

*/

```
public class Main {
    public static void main(String[] args) {

        int opcion = 0;
        Experimento experimento=null;

        while (opcion != 9) {
            System.out.println("\nSeleccione una opción:" +
                "\n1. Abrir un archivo que contenga un experimento" +
                "\n2. Crear un nuevo experimento" +
                "\n3. Crear una población de bacterias y añadirla al experimento actual" +
                "\n4. Visualizar los nombres de todas las poblaciones de bacterias del experimento actual" +
                "\n5. Borrar una población de bacterias del experimento actual" +
                "\n6. Ver información detallada de una población de bacterias del experimento actual" +
                "\n7. Guardar" +
                "\n8. Guardar como" +
                "\n9. Salir");

            System.out.println("\nIntroduzca las fechas siempre en este formato \"yyyy.MM.dd\"");
            do {
                opcion = readInt("Seleccione una opción: ");
                if (opcion < 1 || opcion > 9) {
                    System.out.println("¡ Opción no valida !");
                }
            } while (opcion < 1 || opcion > 9);

            switch (opcion) {
                case 1: //Abrir un archivo que contenga un experimento
                    try {
                        String nombreExperimentoAbrir = readString("Escriba el nombre de su experimento: ");
                        experimento = FileManager.abrirArchivo(nombreExperimentoAbrir);
                    } catch (FileNotFoundException fnf) {
                        System.out.println("No se ha encontrado el archivo. ");
                    } catch (Exception ex) {
                        System.out.println("ERROR.");
                    }
                    break;

                case 2: //Crear un nuevo experimento
                    String nombreNuevoExperimento = readString("Escriba el nombre de su nuevo experimento: ");
                    int numPoblaciones;

                    while (true) {
                        numPoblaciones = readInt("Escriba el número de poblaciones que tiene su nuevo experimento: ");
                    }
                }
            }
        }
    }
}
```

```

        if (numPoblaciones < 0) {
            System.out.println("El número de poblaciones no puede ser negativo.");
        } else {
            break;
        }
    }
    experimento = new Experimento(nombreNuevoExperimento);

    for (int i = 0; i < numPoblaciones; i++) {
        System.out.println("Poblacion " + (i+1) + ":\n");
        try {
            GestionLab.createPoblacion(experimento);
        } catch (Exception ex) {
            System.out.println("Ha ocurrido un error.");
            ex.printStackTrace();
        }
    }

    System.out.println("\nExperimento " + experimento.getNombreExperimento() + " ha
    sido creado correctamente.");
    System.out.println(experimento.toString());
    break;

    case 3: //Crear una población de bacterias y añadirla al experimento actual
        if (experimento == null) {
            System.out.println("No tiene ningún experimento cargado en memoria. Primero debe
    abrir un archivo.");
            break;
        }
        try {
            Poblacion reciénCreada=GestionLab.createPoblacion(experimento);
            System.out.println(reciénCreada.toString());
        } catch (Exception ex) {
            System.out.println("ERROR.");
        }
        break;

    case 4: //Mostrar nombre poblaciones
        if (experimento== null) {
            System.out.println("No tiene ningún experimento cargado en memoria. Primero debe
    abrir un archivo.");
        }
        else {
            String nombres = experimento.toStringNombres();
            System.out.println(nombres);
        }
        break;

    case 5: //Borrar una población de bacterias del experimento actual
        if (experimento== null) {
            System.out.println("No tiene ningún experimento cargado en memoria. Primero debe
    abrir un archivo.");

```



```

    }
    else {
        String pobDeletear = (readString("Escriba el nombre de la población que desea
eliminar: "));
        GestionLab.deletePoblacion(pobDeletear, experimento);
        System.out.println("Su población se ha borrado correctamente.");
    }
    break;

case 6: //Ver información detallada de una población de bacterias del experimento actual
    if (experimento== null) {
        System.out.println("No tiene ningún experimento cargado en memoria. Primero debe
abrir un archivo.");
    }
    else {
        String pobVerInfo = (readString("Escriba el nombre de la población que desea ver la
info: "));
        try {
            Poblacion poblacionEncontrada = GestionLab.buscarPoblacion(pobVerInfo,
experimento);
            System.out.println(poblacionEncontrada.toString());
        } catch (Exception e) {
            System.out.println("La población no existe en este experimento.");
        }
    }
    break;

case 7: //Guardar (se supone que para usar esta opción previamente hemos abierto un
archivo)

    if (experimento== null) {
        System.out.println("No tiene ningún experimento cargado en memoria. Primero debe
abrir un archivo.");
    }
    else{
        boolean guardado =
FileManager.guardarArchivo(experimento.getNombreExperimento(), experimento);
        if (guardado) {
            System.out.println("Experimento guardado correctamente.");
        } else {
            System.out.println("Fallo al guardar el experimento.");
        }
    }
    break;

case 8: //Guardar como
    if (experimento== null) {
        System.out.println("No tiene ningún experimento cargado en memoria. Primero debe
abrir un archivo.");
    }
    else {
        String nombreExperimento = readString("Introduzca el nombre del experimento que
desea guardar: ");
        experimento.setNombreExperimento(nombreExperimento);
    }

```

```

        boolean guardadoComo = FileManager.guardarArchivo(nombreExperimento,
experimento);
        if (guardadoComo) {
            System.out.println("Experimento guardado correctamente.");
        } else {
            System.out.println("Fallo al guardar el experimento.");
        }
    }
    break;

    case 9: //Salir del programa
        System.out.println("Salió del programa.");
        System.out.println("Adiós y muchas gracias.");
        break;
    }
}
}
}
}

```

➤ LISTADO DE PRUEBAS UNITARIAS REALIZADAS.

Paquete tests:

Paquete dataioTest:

○ **Clase FileManagerTest**

```
package test.dataioTest;
```

```

import dataio.FileManager;
import laboratorio.Experimento;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.when;

```

```

public class FileManagerTest {
    @Mock
    private BufferedReader bufferedReader;

    private FileManager fileManager;

    @BeforeEach
    void setUp() {
        MockitoAnnotations.initMocks(this);
        fileManager = new FileManager();
    }
}

```

```

    }
    @Test
    void testAbrirArchivo() throws IOException {
        String input = "newE;30\np1;3;3.0;BAJA;2023-09-09;2.0;2023-09-12;6.0;2023-10-09;4.0\n";
        when(bufferedReader.readLine()).thenReturn(input, null);

        Experimento experimento = null;
        try {
            experimento = fileManager.abrirArchivo("testFile");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }

        assertEquals("newE", experimento.getNombreExperimento());
        assertEquals(30, experimento.getDias());
    }
    @Test
    void testGuardarArchivo() {
        Experimento experimento = new Experimento("newE");
        boolean result = FileManager.guardarArchivo("testFile", experimento);

        assertEquals(true, result);
    }
}

```

- ***Clase UserInputTest***

```

package test.dataioTest;

import dataio.UserInput;
import medio.Luminosidad;
import org.junit.jupiter.api.Test;
import java.io.ByteArrayInputStream;
import java.time.LocalDate;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class UserInputTest {
    void inputDeUsuario(String data){
        ByteArrayInputStream in = new ByteArrayInputStream(data.getBytes());
        System.setIn(in);
    }
    @Test
    void testReadFloat() {
        inputDeUsuario("10,5");
        float result = UserInput.readFloat("Enter a float:");
        assertEquals(10.5, result, 0.001);
    }
    @Test
    void testReadString() {
        inputDeUsuario("test");
        String result = UserInput.readString("Enter a string:");
    }
}

```

```

        assertEquals("test", result);
    }
    @Test
    void testReadInt() {
        inputDeUsuario("10");
        int result = UserInput.readInt("Enter an integer:");
        assertEquals(10, result);
    }
    @Test
    void testReadLuminosidad() {
        inputDeUsuario("ALTA");
        Luminosidad.luminosidad result = UserInput.readLuminosidad("Enter luminosidad:");
        assertEquals(Luminosidad.luminosidad.ALTA, result);
    }
    @Test
    void testReadDate() {
        inputDeUsuario("2023.09.09");
        LocalDate result = UserInput.readDate("Enter a date:");
        assertEquals(LocalDate.of(2023, 9, 9), result);
    }
}

```

Paquete GestionLabTest:

- ***Clase GestionLabTest***

```
package test.GestionLabTest;
```

```

import gestionLab.GestionLab;
import laboratorio.Experimento;
import laboratorio.Poblacion;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import static gestionLab.GestionLab.buscarPoblacion;

```

```

public class GestionLabTest {
    @Test
    public void testCreatePoblacion() throws Exception {
        // Arrange
        Poblacion poblacion = new Poblacion();
        Experimento experimento = new Experimento("Experiment");
        // Act
        GestionLab.createPoblacion( experimento);
        // Assert
        assertEquals(1, experimento.getNumPoblaciones()); // Verify that numPoblaciones is
incremented
        assertEquals(poblacion, experimento.getPoblacionNueva()); // Verify that poblacionNueva is set
correctly
    }
    @Test

```

```

public void testAddPoblacion() {
    // Arrange
    Poblacion poblacion = new Poblacion();
    Experimento experimento = new Experimento("Experiment");
    // Act
    GestionLab.addPoblacion(poblacion, experimento);
    // Assert
    assertEquals(1, experimento.getNumPoblaciones()); // Verify that numPoblaciones is
incremented
    assertEquals(poblacion, experimento.getPoblacionNueva()); // Verify that poblacionNueva is set
correctly
}
@Test
public void testDeletePoblacion() {
    // Arrange
    Poblacion poblacion = new Poblacion();
    poblacion.setNombrePoblacion("NombrePoblacionToRemove");
    Experimento experimento = new Experimento("Experiment");
    experimento.getPoblacionesList().add(poblacion); // Add the poblacion to the list
    // Act
    GestionLab.deletePoblacion("NombrePoblacionToRemove", experimento); // Replace with actual
poblacion name
    // Assert
    assertEquals(0, experimento.getNumPoblaciones()); // Verify that numPoblaciones is updated
    assertEquals(0, experimento.getPoblacionesList().size()); // Verify that poblacion is removed
}
@Test
void testBuscarPoblacion_ExistingPopulation() {
    // Arrange
    Experimento e = new Experimento("Experimento"); // Create an instance of Experimento
    Poblacion existingPoblacion = new Poblacion(); // Create an existing population
    existingPoblacion.setNombrePoblacion("PoblacionAEncontrar");
    e.getPoblacionesList().add(existingPoblacion);
    // Act
    Poblacion result = buscarPoblacion("PoblacionAEncontrar", e);
    // Assert
    assertNotNull(result);
    assertEquals(existingPoblacion, result);
}
@Test
void testBuscarPoblacion_NonExistingPopulation() {
    // Arrange
    Experimento e = new Experimento("Experimento"); // Create an instance of Experimento
    // Act and Assert
    assertThrows(RuntimeException.class, () -> buscarPoblacion("NonExistingCity", e));
}
}

```

Paquete LaboratorioTest:

- **Clase ExperimentoTest:**

```
package test.laboratorioTest;
import laboratorio.Experimento;
import laboratorio.Poblacion;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class ExperimentoTest {
    private Experimento experimento;
    @BeforeEach
    void setUp() {
        experimento = new Experimento("Mi Experimento");
    }
    @Test
    void testGetNombreExperimento() {
        assertEquals("Mi Experimento", experimento.getNombreExperimento());
    }
    @Test
    void testSetNombreExperimento() {
        experimento.setNombreExperimento("Nuevo Nombre");
        assertEquals("Nuevo Nombre", experimento.getNombreExperimento());
    }
    @Test
    void testGetDias() {
        assertEquals(30, experimento.getDias());
    }
    @Test
    void testGetNumPoblaciones() {
        experimento.setNumPoblaciones(5);
        assertEquals(5, experimento.getNumPoblaciones());
    }
    @Test
    void testGetPoblacionesList() {
        assertNotNull(experimento.getPoblacionesList());
    }
    @Test
    void testSetPoblacionNueva() {
        Poblacion poblacion = new Poblacion();
        experimento.setPoblacionNueva(poblacion);
        assertTrue(experimento.getPoblacionesList().contains(poblacion));
    }
}
```

- **Clase PoblacionTest:**

```
package test.laboratorioTest;
import laboratorio.Poblacion;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import java.time.LocalDate;
```

```

import static org.junit.jupiter.api.Assertions.*;

class PoblacionTest {
    private Poblacion poblacion;
    @BeforeEach
    void setUp() {
        poblacion = new Poblacion();
    }
    @Test
    void testGetNombrePoblacion() {
        poblacion.setNombrePoblacion("Población A");
        assertEquals("Población A", poblacion.getNombrePoblacion());
    }
    @Test
    void testSetNumInicialBacterias() {
        poblacion.setNumInicialBacterias(100);
        assertEquals(100, poblacion.getNumInicialBacterias());
    }
    @Test
    void testSetTemperatura() {
        poblacion.setTemperatura(25.5f);
        assertEquals(25.5f, poblacion.getTemperatura());
    }
    @Test
    void testSetFechaInicio() {
        LocalDate fechaInicio = LocalDate.of(2024, 4, 1);
        poblacion.setFechaInicio(fechaInicio);
        assertEquals(fechaInicio, poblacion.getFechaInicio());
    }
    @Test
    void testSetFechaFin() {
        LocalDate fechaFin = LocalDate.of(2024, 4, 30);
        poblacion.setFechaFin(fechaFin);
        assertEquals(fechaFin, poblacion.getFechaFin());
    }
}

```

Paquete medioTest:

○ **ComidaTest:**

```

package test.medioTest;

import medio.Comida;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import java.time.LocalDate;
import static org.junit.jupiter.api.Assertions.*;

public class ComidaTest {
    private Comida comida;
    @BeforeEach

```

```

void setUp() {
    comida = new Comida(2.0f, LocalDate.of(2000, 9, 9), 9.0f, LocalDate.of(2000, 9, 12), 3.0f,
LocalDate.of(2000, 10, 9));
}
@Test
void testConstructor() {
    assertEquals(2.0f, comida.getCantidadInicial());
    assertEquals(9.0f, comida.getCantidadPico());
    assertNotNull(comida.calcularComida());
}
@Test
public void testCalcularComida() {
    float[] result = comida.calcularComida();
    float[] expected = {2.0f, 4.333333f, 6.6666665f, 9.444445f, 9.666667f, 9.888889f, 10.111111f,
10.333333f, 10.555555f, 10.777778f, 11.0f, 11.222222f, 11.444445f, 11.666666f, 11.888889f,
12.111111f, 12.333333f, 12.555555f, 12.777778f, 13.0f, 13.222222f, 13.444445f, 13.666666f,
13.888889f, 14.111111f, 14.333334f, 14.555555f, 14.777778f, 15.0f, 15.222222f};
    assertEquals (expected, result, 0.001f);
}
}

```

- **LuminosidadTest:**

```
package test.medioTest;
```

```
import medio.Luminosidad;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
```

```

public class LuminosidadTest {
    @Test
    void testLuminosidadEnum() {
        // Test the count of enum values
        assertEquals(3, Luminosidad.luminosidad.values().length);
        // Test each enum value
        assertEquals(Luminosidad.luminosidad.ALTA, Luminosidad.luminosidad.valueOf("ALTA"));
        assertEquals(Luminosidad.luminosidad.MEDIA, Luminosidad.luminosidad.valueOf("MEDIA"));
        assertEquals(Luminosidad.luminosidad.BAJA, Luminosidad.luminosidad.valueOf("BAJA"));
    }
}

```

➤ CONCLUSIONES

Concluyendo esta práctica, considero que este ejercicio ha sido extremadamente valioso para mi desarrollo profesional futuro. Ha sido un proyecto extenso que ha requerido una dedicación considerable y me ha ayudado a mejorar mi capacidad de organización y gestión del tiempo. Trabajar en un proyecto de esta magnitud me ha permitido adquirir una comprensión más profunda de los conceptos de programación y de Java en particular.

Aunque el proceso ha sido desafiante y he invertido una cantidad significativa de tiempo y esfuerzo en él, puedo decir que ha valido la pena. A pesar de los momentos de dificultad, he logrado superar los obstáculos y, al final, he aprendido mucho más de lo que esperaba. Siento que ahora tengo un

dominio mucho más sólido de Java y estoy más preparada para enfrentar proyectos futuros.

En cuanto a la documentación, además del javadoc, he descubierto que me ha sido útil dejar algún comentario estratégico en el código. Esto me permitió proporcionar información, lo que facilitó mi comprensión.