ProdRent - Gestor de alquileres (v2)

Introducción 1

La clase Product (1 punto) 2

La clase Client (2 punts) 3

La clase ProductFile (1 punt) 5

La clase ClientFile (1 punt) 6

La classe LogFile 8

La classe ProdRent (3 puntos) 8

- El método main 9
- El método run 9
- El método processMovements 9
- El fichero de movimientos 10
  - Alta de un producto 10
  - Alta de un cliente 11
  - Petición de información sobre un producto 11
  - Petición de información sobre un cliente 11
  - Alquiler de un producto a un cliente 11
  - Devolución de un producto de un cliente 12

Ejemplos de ejecución 12

Formato de la entrega 13

Introducción

Los objetivos de la práctica son

- tratamiento de ficheros secuenciales de texto organizados por líneas
- tratamiento de ficheros de acceso directo binarios organizados por registros
- descomposición descendente de acciones y funciones
- descomposición en clases auxiliares

El contexto de la práctica es gestionar la información sobre clientes y productos de una empresa que alquila sus productos a sus clientes. Durante el enunciado iremos presentando las reglas de funcionamiento de dicha empresa, que tendrán incidencia en las implementación de las clases de nuestra solución.

Las operaciones que consideraremos serán las siguientes:

- dar de alta un nuevo producto
- dar de alta un nuevo cliente
- pedir información sobre un producto
- pedir información sobre un cliente
- alquilar un producto a un cliente
- devolver un producto de un cliente

Para guardar de forma permanente la información tanto de clientes como de productos, se dispondrá de sendos ficheros binarios de acceso directo. Además, las acciones a realizar vendrán expresadas como líneas en un fichero de texto de movimientos y, al realizar cada operación, tanto de forma exitosa como no, dejaremos constancia en un fichero de bitácora.

Como es costumbre, en la descripción del enunciado detallaremos los pasos a realizar y el orden en que hay que realizarlos. Además, el proyecto que os proporcionaremos contendrá implementaciones parciales de alguno de ellos y clases de prueba de las clases auxiliares.

La clase `Product` (1 punto)

Es la clase que representa la información que se guarda de un producto.

Define las constantes

- `static final int DESCRIPTION_LIMIT = 20`
  - limita el número de caracteres que se considerarán del atributo `description` en las operaciones `toBytes/fromBytes`
- `static final int SIZE = ???`
  - tamaño en bytes de los registros de producto

Los atributos privados que tiene son:

- `long id`: identificador del producto (comienzan a partir de 1L)
- `String description`: descripción del producto
- `int price`: precio del producto
- `int stock`: número de unidades disponibles del producto

En cuanto operaciones públicas, tendremos un constructor:

- `Product(long id, String description, int price, int stock)`
  - Precondición: `id > 0L && price > 0 && stock > 0`
    - no hace ninguna comprobación sobre los mismos
    - serán las operaciones que invocan el constructor las que deberán asegurar que tanto `id` como `price` como `stock` sean positivos
  - inicializa el producto con los valores indicados

Dispondremos también de getters para todos los atributos:

- `int getId()`
- `String getDescription()`

- `int getPrice()`
- `int getStock()`

De métodos para modificar el stock:

- `void incrementStock()`
  - aumenta en una unidad el stock del producto
- `void decrementStock()`
  - Precondición: `this.stock > 0`
    - recordad, son la operaciones que la llaman las que se aseguran de que antes de llamar al método, se cumple la precondición
  - decrementa en una unidad el stock del producto
    - fijaos en que, debido a la precondición, el valor del stock jamás será negativo

Para convertir desde / hacia un array de bytes, disponemos de los métodos, con el significado habitual:

- `byte[] toBytes()`
- `static Product fromBytes(byte[] record)`

Como métodos adicionales que os damos implementados, básicamente para los tests y para la escritura de productos en el fichero de bitácora, disponemos de:

- `boolean isEqualTo(Product other)`
  - devuelve si el producto receptor es igual (tiene los mismos valores en todos los atributos) que el que se pasa como parámetro
- `String toString()`
  - crea una cadena con los datos que se tienen de un producto

De cara a comprobar mínimamente su funcionamiento se ha creado la clase `ProductTest`.

La clase `Client` (2 punts)

Es la clase que representa la información sobre los clientes, que incluye los productos y cantidades de los mismos que tienen alquilados.
Esta clase define las constantes:

- `static final int MAX_PRODUCTS = 3`
  - este número limita el número máximo de productos diferentes que se pueden tener alquilados
  - no limita las unidades, sino la cantidad de productos con `id` diferente
    - es decir, podemos tener alquiladas 20 unidades del producto 1L, 2 del producto 12L y 45 del producto 17L
    - pero ya no podríamos alquilar una unidad del producto 7L
    - aunque sí podríamos alquilar más unidades de los productos 1L, 12L ó 17L
- `static final int NAME_LIMIT = 20`

    - limita el número de caracteres que se considerarán del atributo `name` en las operaciones `toBytes/fromBytes`
- `static final int SIZE = ???`
    - tamaño en bytes de los registros de cliente

Esta clase tiene los atributos:

- `long id`
- `String name`
- `int balance`

En esta lista de atributos no aparecen los que gestionan los productos que tiene un cliente. **Estos atributos los decidís vosotros** y os han de permitir implementar las operaciones de la clase Cliente.

Pista: repasad la solución que presentamos del primer parcial y veréis cómo podéis aplicar la misma idea a este problema.

En cuanto operaciones públicas, tendremos un constructor:

- `Client(long id, String name, int balance)`
    - Precondición: `id >= 0L && balance > 0`
        - no hace ninguna comprobación sobre los mismos
        - serán las operaciones que invocan el constructor que deberán asegurar que tanto `id` como `balance` sean positivos

Tiene los getters:

- `long getId()`
- `String getName()`
- `int getBalance()`

Y las siguientes operaciones para gestionar el saldo

- `void addBalance(int amount)`
    - Precondición: `amount > 0`
    - incrementa el saldo en la cantidad dada
- `void subBalance(int amount)`
    - Precondición: `this.balance >= amount > 0`
    - decrementa el saldo en la cantidad dada
        - gracias a la precondición, el saldo resultante jamás será negativo

Las operaciones que gestionan los productos que tiene alquilado un cliente, y para las que necesitaréis añadir más atributos a la clase, son:

- `boolean canAddProduct(long idProduct)`
    - comprueba si el cliente puede alquilar una nueva unidad del producto con identificador `idProduct`
    - recordad que como máximo solamente se pueden tener 3 productos alquilados con identificadores diferentes
- `boolean hasProduct(long idProduct)`

    - comprueba si el cliente tiene alquilado algún producto con identificador `idProduct`
- `void addProduct(long idProduct)`
    - añade, si es posible, una unidad del producto `idProduct` a los productos que el cliente tiene alquilados
    - en caso de que no fuera posible, la operación no hace nada[1]
- `void removeProduct(long idProduct)`
    - elimina una unidad del producto con identificador `idProduct`
- `int getProductStock(long idProduct)`
    - devuelve el número de unidades del producto con identificador `idProduct` que tiene alquiladas el cliente
    - si no tiene ninguna, obviamente, devuelve 0
- long[] getProductIds()
    - devuelve un array, **creado por la propia operación**, con los valores de los identificadores de los productos que tiene alquilado el cliente
    - en caso de no tener producto alguno, devuelve el **array vacío**, es decir, de longitud cero.

También tendremos que implementar las operaciones que convierten desde y hacia array de bytes, que por supuesto, también tendrán que guardar y restaurar los valores de los atributos que habéis añadido a la clase.

- `byte[] toBytes()`
- `Client fromBytes(byte[] record)`

Y las operaciones, que os damos ya implementadas, básicamente para los tests y para la escritura de productos en el fichero de bitácora, disponemos de:

- `boolean isEqualTo(Client other)`
    - devuelve si el cliente receptor es igual (tiene los mismos valores en todos los atributos) que el que se pasa como parámetro
- `String toString()`
    - crea una cadena con los datos que se tienen de un producto
    - para representar la información de los productos que tiene un cliente, se muestra un vector con los `idProducto`, ordenados crecientemente, y otro con las cantidades de cada uno de ellos

De cara a comprobar mínimamente su funcionamiento se ha creado la clase `ClientTest`.

La clase `ProductFile` (1 punt)

Esta clase servirá para encapsular las operaciones que tienen que acceder al fichero binario de acceso aleatorio que guarda información sobre los productos.

[1] Como veréis el curso que viene, lo razonable en este caso sería lanzar una excepción. Como el tratamiento de errores en Java no es parte del temario de la asignatura, simplemente no haremos nada en caso de que la operación no pueda llevarse a cabo.

Este fichero estará organizado por registros de productos (todos ellos de la misma longitud Product.SIZE) de manera que el primer registro del fichero se corresponde con el producto con identificador 1L, el segundo con el de identificador 2L, etc.
Todos los métodos de la clase, constructor incluído, podrán lanzar la excepción IOException

El constructor de la clase:

- `ProductsFile(String fileName) throws IOException`
    - crea la instancia de RandomAccessFile correspondiente al fichero de nombre fileName para realizar tanto operaciones de escritura como de lectura

Operaciones sobre el contenido del fichero y, tal y como hemos hecho en los ejemplos de clase, no se preocuparán de si existen realmente las posiciones de los ficheros a las que intentan acceder.

- `void write(Product product) throws IOException`
    - escribe los datos del producto en la posición que le corresponde en el fichero según el valor de su identificador (el primer registro del fichero para el producto con identificador 1L, el segundo para el de identificador 2L, etc.)
- `Product read(long idProduct) throws IOException`
    - devuelve el producto que se encuentra en la posición correspondiente al identificador que se pasa como parámetro

Operaciones sobre identificadores

- `long nextId() throws IOException`
    - devuelve el identificador que le corresponderá al siguiente producto a añadir el fichero
    - su valor será el número de registros existentes actualmente más uno, para no dejar huecos en el mismo
- `boolean isValid(long idProducto) throws IOException`
    - indica si el identificador dado es válido para el fichero, es decir, su valor está entre 1L y el número de registros del fichero

Operaciones de gestión del fichero de acceso aleatorio

- `void reset() throws IOException`
    - pone a cero la longitud del fichero de acceso aleatorio correspondiente a los productos (eliminando así su contenido)
- `void close() throws IOException`
    - cierra el fichero de acceso aleatorio correspondiente a los productos

De cara a comprobar posibles errores en la implementación, se dispone de la clase ProductsFileTest.

La clase `ClientFile` (1 punt)

Esta clase servirá para encapsular las operaciones que tienen que acceder al fichero binario de acceso aleatorio que guarda información sobre los clientes.

Este fichero estará organizado por registros de clientes (todos ellos de la misma longitud Client.SIZE) de manera que el primer registro del fichero se corresponde con el cliente con identificador 1L, el segundo con el de identificador 2L, etc.
Todos los métodos de la clase, constructor incluído, podrán lanzar la excepción IOException

El constructor de la clase:

- `ClientsFile(String fileName) throws IOException`
    - crea la instancia de RandomAccessFile correspondiente al fichero de nombre fileName para realizar tanto operaciones de escritura como de lectura

Operaciones sobre el contenido del fichero y, tal y como hemos hecho en los ejemplos de clase, no se preocuparán de si existen realmente las posiciones de los ficheros a las que intentan acceder.

- `void write(Client product) throws IOException`
    - escribe los datos del cliente en la posición que le corresponde en el fichero según el valor de su identificador (el primer registro del fichero para el cliente con identificador 1L, el segundo para el de identificador 2L, etc.)
- `Client read(long idClient) throws IOException`
    - devuelve el cliente que se encuentra en la posición correspondiente al identificador que se pasa como parámetro

Operaciones sobre identificadores

- `long nextId() throws IOException`
    - devuelve el identificador que le corresponderá al siguiente cliente a añadir el fichero
    - su valor será el número de registros existentes actualmente más uno, para no dejar huecos en el mismo
- `boolean isValid(long idClient) throws IOException`
    - indica si el identificador dado es válido para el fichero, es decir, su valor está entre 1L y el número de registros del fichero

Operaciones de gestión del fichero de acceso aleatorio

- `void reset() throws IOException`
    - pone a cero la longitud del fichero de acceso aleatorio correspondiente a los clientes (eliminando así su contenido)
- `void close() throws IOException`
    - cierra el fichero de acceso aleatorio correspondiente a los productos

De cara a comprobar posibles errores en la implementación, se dispone de la clase `ClientsFileTest`.

NOTA: Como veréis, salvo algunos valores para los tipos de las variables, el código es el mismo para esta clase y la anterior. El año que viene veremos técnicas para evitar esta duplicación de código.

La classe `LogFile`

Esta clase os la proporcionamos nosotros y contiene los métodos que escriben, en el fichero de bitácora, tanto los resultados de las operaciones exitosas como los errores que se encuentran en las mismas.

La classe `ProdRent` (3 puntos)

Esta clase representa el programa principal y utilizará todas las clases descritas anteriormente.

```
public class ProdRent extends CommandLineProgram {

    private static final String PRODUCTS = "productsDB.dat";
    private static final String CLIENTS = "clientsDB.dat";
    private String movements;
    private String logger;

    private BufferedReader movementsFile;
    private LogFile logFile;
    private ProductFile productsDB;
    private ClientFile clientsDB;

    public static void main(String[] args) {
        new ProdRent().start(args);
    }

    public void run() {
        try {
            askFileNames();
            openFiles();
            resetFiles();
            processMovements();
            closeFiles();
        } catch (IOException ex) {
            println("ERROR");
            ex.printStackTrace();
        } finally {
            try {
                closeFiles();
            } catch (IOException ex) {
                println("ERROR Closing");
                ex.printStackTrace();
            }
```

```
            }
        }

        ...
    }
```

Las variables de instancia del programa principal son las siguientes:

- `movements`: nombre del fichero de texto que tiene los movimientos
- `logger`: nombre del fichero de bitácora
- `movementsFile`: fichero de texto de lectura organizado por líneas que contiene las operaciones a realizar
- `logFile`: instancia de la clase `LogFile` para realizar las operaciones de escritura sobre el fichero de bitácora
- `productsDB`: instancia de la clase `ProductsDB` que contiene la información sobre productos
- `clientsDB`: instancia de la clase `ClientsDB` que contiene la información sobre libros

El método `main`

Lo hemos añadido para que el hecho de tener la clase que representa el programa principal fuera del paquete por defecto no presente problemas en algunas plataformas.

El método `run`

Más o menos se corresponde con el método run que hemos visto en multitud de ejemplos. Lo único que hemos añadido es el bloque finally, para asegurar que, aunque se produzca un error, se intenten cerrar los ficheros. Esto es especialmente importante para el fichero de bitácora pues, si no se hace así, es posible que lo que se ha escrito en último momento no quede registrado en el fichero (lo que puede complicar saber qué está ocurriendo realmente).

Las llamadas a printStackTrace() escriben información sobre qué línea del programa ha llamado a la operación que ha lanzado la excepción.

El método `processMovements`

La parte más importante de vuestra solución será descomponer en acciones y funciones pequeñas, intentando que cada función realice directamente solamente una tarea y que, por tanto, sea sencilla de programar y de entender.
En las siguientes secciones del enunciado comentaremos los aspectos que debéis de conocer sobre:

- formato del fichero de movimientos
- tipos de movimientos que pueden realizarse
    - casos de error que debéis controlar
    - casos que no se han de controlar y que podéis suponer que son correctos

- llamadas que se hacen a los métodos de la clase Logger

El fichero de movimientos

Tal y como se ha comentado anteriormente este fichero contiene las operaciones a realizar sobre los ficheros de datos de productos y de clientes. Como se ha mencionado anteriormente, estas operaciones posibles son:

- dar de alta un nuevo producto
- dar de alta un nuevo cliente
- pedir información sobre un producto
- pedir información sobre un cliente
- alquilar un producto a un cliente
- devolver un producto de un cliente

Como en todos los ejemplos que hemos visto tanto en los apuntes como en los problemas, supondremos que **el fichero está bien formado** y, por tanto, para cada movimiento estarán todos los parámetros y serán de los tipos adecuados (es decir, si han de corresponderse con un número, serán un número).

Eso sí, si detectamos que la operación indicada no es una de las existentes, se anotará tal hecho en el fichero de bitácora utilizando el método errorUnknownOperation de la clase Logger.

En todas las operaciones, se intentarán comprobar el máximo de situaciones de error y las comprobaciones se se realizarán en el **orden en que aparecen en el enunciado**. Asimismo, si la operación se ejecuta con **éxito**, también se indicará este hecho en el fichero de bitácora.

Alta de un producto

El formato de la línea será

```
ALTA_PRODUCTO,descripción,precio,stock
```

Ejecución:

- se calcula un id para el producto, de manera que los productos tengan identificadores consecutivos, que el primer producto tenga identificado 1L, y que no se dejen huecos en el fichero
- se añade el producto al fichero de productos
- indicar en el fichero de bitácora que se ha dado de alta el producto con el método `okNewProduct` de la clase `LogFile`

Casos de error:

- el precio no puede ser negativo o cero (se indica el error con el método `errorPriceCannotBeNegativeOrZero`)
- el stock no puede ser negativo o cero (se indica el error con el método `errorStockCannotBeNegativeOrZero`)

Alta de un cliente

El formato de la línea será

```
ALTA_CLIENTE,nombre,saldo
```

Ejecución:

- se calcula un id para el cliente, de manera que los clientes tengan identificadores consecutivos, que el primer cliente tenga identificado 1L, y que no se dejen huecos en el fichero
- se añade el cliente al fichero de clientes
- indicar en el fichero de bitácora que se ha dado de alta el producto con el método `okNewClient` de la clase `LogFile`.

Casos de error:

- el saldo no puede ser negativo o cero (se indica el error con el método `errorBalanceCannotBeNegativeOrZero`)

Petición de información sobre un producto

El formato de la línea será

```
INFO_PRODUCT,idProduct
```

Ejecución:

- se obtienen los datos del producto correspondiente a `idProduct`
- se añade la información del producto al fichero de bitácora con el método `infoProduct`

Casos de error:

- `idProduct` no es un identificador válido en el fichero de productos (`errorInvalidProductId`)

Petición de información sobre un cliente

El formato de la línea será

```
INFO_CLIENT,idClient
```

Ejecución:

- se obtienen los datos del cliente correspondiente a `idClient`
- se obtienen los datos de los productos que el cliente tiene alquilado
    - en este caso no hace falta comprobar que los identificadores son correctos, ya que son los que hemos guardado nosotros y no una información que viene del exterior
- se añade esa información al fichero de bitácora con el método `infoClient`

Casos de error:

- `idClient` no es un identificador válido en el fichero de clientes (`errorInvalidClientId`)

Alquiler de un producto a un cliente

El formato de la línea será

```
ALQUILAR,idClient,idProduct
```

Ejecución:

- se obtienen de los ficheros las instancias de cliente y producto según sus identificadores
- se añade una unidad de `idProduct` a los productos que tiene el cliente
- se actualiza el saldo del cliente en función del precio del producto
- se decrementa el stock del producto en una unidad
- se guardan los resultados
- se indica la finalización exitosa con el método `okRent`

Casos de error:

- `idClient` no es un identificador válido en el fichero de clientes (`errorInvalidClientId`)
- `idProduct` no es un identificador válido en el fichero de productos (`errorInvalidProductId`)
- no hay alguna unidad del producto en stock para ser alquilada (`errorCannotRentProductWithNoStock`)
- el cliente no dispone de saldo para pagar el precio del producto (`errorClientHasNotEnoughFundsForRentProduct`)
- el cliente no puede añadir una unidad del producto con idProduct (que es válido) a los que ya tiene alquilados (`errorClientCannotAddProduct`)

Devolución de un producto de un cliente

El formato de la línea será

```
DEVOLVER,idClient,idProduct
```

Ejecución:

- se obtienen de los ficheros las instancias de cliente y producto según sus identificadores
- se elimina una unidad de `idProduct` a los productos que tiene el cliente
- se incrementa el stock del producto en una unidad
- se guardan los resultados
- se indica la finalización exitosa con el método `okReturn`

Casos de error:

- `idClient` no es un identificador válido en el fichero de clientes (`errorInvalidClientId`)
- `idProduct` no es un identificador válido en el fichero de productos (`errorInvalidProductId`)
- el cliente realmente no tiene alquilada alguna unidad del producto con `idProduct`, que es válido (`errorClientHasNotProduct`)

Ejemplos de ejecución

En el directorio `examples`, encontraréis ejemplos de ejecución de la práctica, formada por parejas de ficheros:

- `xxxx.txt` es el fichero que contiene las operaciones
- `xxxx.out` es el fichero de bitácora obtenido en la ejecución

Es conveniente que, antes de realizar las implementaciones de las operaciones, entendáis el porqué de los resultados que muestran los ejemplos.

Formato de la entrega

Un fichero comprimido con **ZIP** que contenga

- directorio del proyecto IntelliJ
- documentación del proyecto (**2 puntos**)
    - un informe en **PDF** (máximo 5 páginas páginas, contando la portada) con
        - portada con el nombre de la práctica, nombre del alumno, dni (o equivalente) y grupo al que pertenece
        - diseño de la gestión de los productos en la clase `Client`
            - variables de instancia que habéis añadido
            - diseño de las operaciones que involucran a la gestión de productos
        - los diagramas correspondientes a los registros de autores y libros indicando los tamaños y posiciones de cada una de las partes del registro
        - comentad cómo habéis enfocado la descomposición descendente del método `processMovements`. Una forma de explicarlo es, para cada método de la descomposición, explicar de qué parte del problema se ocupa directamente y qué parte delega en otras operaciones.
            - Por ejemplo, si hacemos la descomposición habitual de tratamiento de las líneas de un fichero de texto, el método `processMovements` lee una a una las líneas del fichero de movimientos y llama a `processMovement` para tratar el movimiento correspondiente a esa línea, ...
    - javadoc para todos lo métodos públicos que definái de las clases `Product`, `Client`, `ProductFile` y `ClientFile`.

Recordad, antes de entregar el ZIP aseguraos de que, cuando se descomprime, contiene tanto el informe en PDF como el directorio del proyecto.