Programación Orientada a Objetos

Curso 2023/2024

Convocatoria Julio

Alumno/a: _____GRUPO:_____

Preparación del examen

- Crea la carpeta examen" en el Escritorio.
- Arranca Eclipse y utiliza la carpeta "examen" como espacio de trabajo.
- Crea un proyecto Java que incluya tu grupo, apellidos y nombre, siguiendo la plantilla: **GXX-*Apellidos-Nombre-Julio***. Por ejemplo "G11-MartinezLopez-Juan-Julio".
- Al acabar el examen, comprime la carpeta con el proyecto (no el espacio de trabajo). El nombre del fichero comprimido debe coincidir con el nombre del proyecto con extensión .rar, .zip o la correspondiente al compresor utilizado.
- Sube el fichero a la tarea del **Aula Virtual** "Entrega examen convocatoria julio".

Previo. Gestión de fechas.

- Para representar las fechas utilizaremos la clase **`java.time.LocalDateTime`**, que implementa una fecha (día y hora) sin información de la zona horaria de acuerdo al sistema de calendario ISO-8601.
- El formato (por defecto) es `año-mes-día`**`T`**`hora:min:segundos`, `2024-06-16T10:00:00`.
- La clase `LocalDateTime` no dispone de constructores. Para crear objetos se utiliza el método de clase *`of`*, que recibe como parámetro el año, mes (como un entero o un valor del enumerado `Month`), día, hora y minutos. Por ejemplo:

```
LocalDateTime fechaHora = LocalDateTime.of(2024, 06, 16, 10, 0);

LocalDateTime fechaHora = LocalDateTime.of(2024, Month.JUNE, 16, 10, 0);
```

- La clase `LocalDateTime` dispone del método de clase `now()` que devuelve la fecha y hora actual.
- Las fechas (`LocalDateTime`) son **comparables**, aun así, también están disponibles los métodos `isAfter`, `isBefore` e `isEqual`.
- Los objetos `LocalDateTime` son **inmutables**. Por ejemplo, la clase ofrece el método `plusMinutes (int numMinutes)` que devuelve una nueva fecha con el resultado de la suma del número de minutos que se pasa como parámetro al objeto receptor de la llamada.

El objetivo del examen es desarrollar una aplicación que permita a los usuarios alquilar bicicletas depositadas en estaciones de aparcamiento.

Bloque 0: Ejercicios comunes a los bloques 1 y 2

Nota: Este bloque será valorado junto al bloque 2.

a) Implementa la **clase `Reserva`** de acuerdo con la siguiente especificación:

Una reserva representa la solicitud de un usuario para mantener bloqueada una bicicleta durante un tiempo determinado. Esta clase implementa **objetos inmutables** con las siguientes propiedades:

- idBici: número entero que representa el identificador de la bicicleta.
- usuario: cadena de texto con el identificador del usuario que realiza la reserva.
- caducidad: instante de tiempo (`LocalDateTime`) en el que la reserva deja de estar activa.
- activa: una reserva se considera activa si el instante actual es anterior a la caducidad.

- caducada: una reserva está caducada si no está activa.

La clase ofrece un constructor que recibe como parámetros los identificadores de la bicicleta y el usuario. La caducidad se calcula a partir del instante actual sumando 30 minutos.

b) Crea una clase **`Programa`** con la siguiente funcionalidad:

- Declara una lista vacía de reservas.
- Crea una reserva para la bici 1 y usuario "juan". Añade la reserva a la lista.
- Igual que el anterior con bici 2 y usuario "pedro", bici 3 y usuario "enrique".
- Muestra en la consola el contenido de la lista para comprobar que todo ha ido bien.

Bloque 1 (4,5 ptos): Bloque compensable por parciales

NOTA: Este apartado no es obligatorio para el alumnado que haya aprobado los parciales. En cualquier caso, se mantiene la nota sacada por parciales si se implementa y se obtiene menos nota en este apartado.

1. (0,1 ptos) Extensión de la clase `Reserva`

Redefine los métodos `equals/hashCode` en la clase Reserva. Dos reservas son iguales si tienen el mismo valor para todas sus propiedades.

2. (0,35 ptos) Eventos de alquiler

Implementa la clase `EventoBici` que representa las acciones del proceso de alquiler de una bicicleta mediante objetos inmutables. Las propiedades son:

- usuario: cadena de texto con el identificador del usuario.
- tiempo: instante de tiempo (`LocalDateTime`) en el que se ha realizado la acción.
- tipo: identifica la acción de alquiler, que solo puede ser "inicio alquiler" o "fin alquiler".

La clase ofrece un constructor que recibe como argumentos las tres propiedades. Adicionalmente, declara un segundo constructor que omite el tiempo e inicializa esta propiedad con el instante actual.

3. (1,6 ptos) Estación

La clase `Estacion` define una estación de aparcamiento de bicicletas que se ofrecen para ser alquiladas. La clase tiene como propiedades:

- nombre: cadena de texto con el nombre de la estación. Esta propiedad no puede variar una vez establecida.
- bicis: conjunto con los identificadores de las bicicletas gestionadas por la estación (cada bicicleta está identificada por un número).
- número de bicis: cantidad de bicis que gestiona la estación.

En relación con la propiedad `bicis`, la implementación de la clase almacena la propiedad en un mapa en el que las claves son los identificadores de las bicicletas (enteros) y el valor asociado es una lista de eventos de alquiler (`EventoBici`). Este mapa es un atributo de implementación denominado `eventos`.

El constructor de la clase recibe como argumentos el nombre y el número inicial de bicicletas. En el constructor se debe inicializar el mapa registrando una entrada para cada una de las bicicletas comenzando por el número 1. A cada bicicleta se le asocia una lista vacía. Por tanto, si el número inicial de bicicletas es 3, se crearán las entradas del mapa con identificadores 1, 2 y 3.

La funcionalidad que ofrece la clase `Estacion` es la siguiente:

- Añadir una bici. La tarea es añadir una nueva bici a la estación. Tendrá como identificador el número actual de bicis de la estación más uno. La operación retorna el identificador de la nueva bici.
- Consultar si una bici es alquilable (parámetro: identificador de la bicicleta). Una bici es alquilable si en el mapa de eventos la bicicleta está asociada a una lista vacía o el último evento de la bici es "fin alquiler". Además, si la bici no existe retornará falso porque no es alquilable.
- Alquilar (parámetros: identificador bici, identificador usuario, instante inicial). El objetivo de la operación es crear un alquiler si la *bici es alquilable*. Por tanto, la operación retorna un valor booleano informando si ha tenido éxito. Si se puede alquilar, se crea un objeto `EventoBici` de tipo "inicio alquiler" y se almacena en último lugar de la lista de eventos asociados a la bici.
- Versión sobrecargada del método alquilar en el que se omite el instante inicial en el que se hace el alquiler y se entiende que estamos haciendo referencia al instante actual (fecha y hora actual).
- Consultar si una bici está alquilada (parámetro: identificador de la bicicleta). Una bici cumple esta condición si el último evento que tiene asociado es de tipo "inicio alquiler".
- Estacionar (parámetros: identificador bici, identificador usuario, instante final). Esta operación da por finalizado un alquiler entregando la bicicleta. Retorna un número real con el coste del alquiler concluido. En primer lugar, crea un `EventoBici` de tipo "fin alquiler" y lo almacena al final de la lista de eventos de la bici. A continuación, calcula el número de minutos entre el instante inicial y el instante final del alquiler. El coste del alquiler se calcula como el número de minutos multiplicado por 0,01 euros. Por último, si la bici no estaba alquilada, se retornará el valor -1.

  Nota: la llamada `ChronoUnit.MINUTES.between(fin, inicio)` calcula el número de minutos transcurridos entre dos instantes de tiempo (`LocalDateTime`).
- Consultar las bicicletas alquiladas. Retorna un conjunto con los identificadores de todas las bicicletas que están alquiladas.

4. (1,5 ptos) Estación con reservas

Una estación con reservas es un tipo de estación que permite a los usuarios reservar una bici durante un tiempo determinado para poder alquilarla.

La clase `EstacionReservas` no declara propiedades nuevas, sino que utiliza un atributo de implementación, `reservas`, que mantiene una lista con todas las reservas realizadas.

La funcionalidad que añade la clase es la siguiente:

- Consultar la reserva activa de un usuario. Esta operación retorna el valor nulo si no la tiene. Nótese que un usuario solo puede tener una reserva activa.
- Consultar si un usuario tiene reserva activa. Esta operación retorna un valor booleano.
- Consultar el número de reservas caducadas de un usuario.
- Consultar si un usuario está bloqueado. Un usuario está bloqueado (quiere que decir que no podrá hacer reservas ni alquilar una bici) si tiene 3 reservas caducadas.

- Consultar si una bici está reservada. Esta condición se cumple si tiene una reserva que esté activa.
- Reservar una bici por parte de un usuario. No está permitido hacer una reserva si el usuario está bloqueado o tiene una reserva activa. Además, tampoco se puede reservar una bicicleta que ya esté reservada o que no sea alquilable. Si se permite realizar la reserva, se crea un objeto `Reserva` y se añade a la lista de reservas. La operación retorna un valor booleano informando si se ha efectuado la reserva.
- Desbloquear usuario. Esta operación elimina todas las reservas caducadas de un usuario. **Requisito**: implementar la operación haciendo uso de un iterador explícito.

En una estación con reservas se puede alquilar una bici reservada siempre que el usuario que solicita el alquiler coincida con la reserva. En este caso, antes de efectuar el alquiler se elimina la reserva. En el caso de que la bici no esté reservada, la política de alquiler coincide con el de cualquier estación. Por último, recuerda que un usuario bloqueado no puede realizar alquileres.

Finalmente, los usuarios con alguna reserva caducada tienen una penalización sobre el precio del alquiler, el cual se duplica.

5. (0,65 ptos) Redefinición de métodos de `Object`

- Implementa `toString` en todas las clases siguiendo las recomendaciones de la asignatura.
- Implementa `clone` en la jerarquía de estaciones siguiendo las recomendaciones de la asignatura. Se implementará una **copia profunda**.

6. (0,3 ptos) Programa

En el programa del bloque 0 añade la siguiente funcionalidad:

- Crea una estación con nombre "Glorieta" y 4 bicicletas:
  - Da de alta una nueva bici y muestra el identificador asignado por la consola.
  - El usuario "juan" alquila la bici 1 en este momento.
- Crea una estación con reservas con nombre "Plaza Circular" y 3 bicicletas:
  - El usuario "juan" reserva la bici 1.
- Construye una lista de estaciones y añade las dos estaciones anteriores.
- Construye una lista vacía de estaciones llamada "copias".
- Recorre la lista de estaciones:
  - Muestra el conjunto de bicis.
  - Crea una copia y añádela a la lista "copias".
- Añade la lista copias a la lista de estaciones.
- Recorre de nuevo la lista de estaciones:
  - Si la estación permite reservas, comprueba si el usuario "juan" tiene una reserva activa y en ese caso, efectúa el alquiler de la bici reservada.
  - Muestra el nombre de la estación y las bicis alquiladas.

Bloque 2 (2 ptos)

7. (0,4 ptos) Bloque 0, ejercicio a)

8. (0,1 ptos) Bloque 0, ejercicio b)

9. (0,45 ptos) Procesamiento utilizando el modelo *stream*

En el programa del bloque 0, utilizando la lista de reservas, implementa las siguientes consultas utilizando el *procesamiento basado en streams*:

- Consulta si la bici 1 ha sido reservada. Respuesta esperada: `true`.
- Cuenta el número de reservas caducadas de "juan". Respuesta esperada: 0
- Obtén una lista con las reservas **activas** ordenadas alfabéticamente por el identificador del usuario. Muestra el resultado en la consola. Respuesta esperada: una nueva lista con todas las reservas ordenadas.

10. (0,6 ptos) Excepciones

El propósito de este ejercicio es escribir un método que genere un número aleatorio si se cumple una condición. Para ello:

a. Define una excepción comprobada con nombre `GeneradorException` que se utilizará para notificar que no ha podido cumplir su tarea. La excepción debe permitir establecer un mensaje de error.
b. Crea una clase `Utilidades` y declara un método de clase `getNumero`. Esta operación tiene dos parámetros que especifican una horquilla de valores enteros, por ejemplo, 10 y 20 definen la horquilla de valores entre 10 y 20, ambos incluidos. La horquilla debe definirse en el rango de 0 a 100. Utilizando un generador de números aleatorios se obtendrá un número al azar en el rango 0 a 100. Si el número está comprendido en la horquilla establecida, será retornado por la operación. En caso contrario, se intentará hasta 3 veces y si no se consigue se notifica la excepción `GeneradorException`.

   Nota: utiliza la clase `java.util.Random` para obtener un generador aleatorio. El método `nextInt(cota)` genera un número al azar entre 0 y cota - 1.

c. Incluye el **control de *precondiciones*** en la operación.
d. Introduce una prueba en el programa.

11. (0,45 ptos) Genericidad

Implementa en la clase `Utilidades` un método de clase genérico denominado `fill` cuya tarea es introducir en una colección *n* elementos con valor `o`. Por ejemplo, la llamada `fill(lista, 5, "examen")` deberá introducir 5 veces la cadena "examen" en la colección `lista`.

En el programa principal realiza una prueba del método.