Semana 14: Clase de repaso

Ejercicios de examen del curso 2021/22

Los siguientes ejercicios han sido extraídos de los exámenes del curso 2021-2022. Estos ejercicios se centran en cuestiones relacionadas con los últimos temas de la asignatura: *colecciones*, *genericidad*, *programación funcional* y *excepciones*.

1. Clase tramo horario

Un **tramo horario** se define como un tipo de datos inmutable que se caracteriza por las siguientes propiedades:

- Día de la semana (valor del enumerado `java.time.DayOfWeek`).
- Hora de inicio (entero).
- Hora de finalización (entero).

La clase ofrece un método de consulta que comprueba si una fecha (`LocalDateTime`) pasada como parámetro se encuentra dentro del tramo horario. Una fecha está dentro del tramo horario si la fecha se corresponde con el mismo día de la semana y la hora de la fecha está comprendida entre la hora de inicio y la hora de finalización (excluida). En ese caso devolverá `true`, en caso contrario devolverá el valor `false`.

Ejercicios:

a) Implementa la clase `TramoHorario` que se ajuste a la especificación anterior.

b) Implementa el ***orden natural*** (interfaz `Comparable`) de la clase `TramoHorario` teniendo en cuenta que un tramo horario `th1` es menor que otro `th2` si el día de la semana de `th1` es anterior al día de la semana de `th2`. Si los dos tramos horarios se refieren al mismo día se ordenan atendiendo a la hora de inicio.

c) Implementa el ***control de precondiciones*** en el constructor de la clase.

d) Implementa el método `toString` siguiendo las recomendaciones de la asignatura.

e) Crea la clase `Programa1` que incluya la siguiente funcionalidad:

- Crea una lista con 4 tramos horarios: dos el lunes, uno el viernes y otro el sábado por la mañana.
- Aplicando **procesamiento basado en *streams***:
  - Cuenta el número tramos horarios que existen en la lista correspondientes al lunes (`MONDAY`).
  - Construye un conjunto con los tramos horarios en los que esté dentro la fecha actual.
  - Muestra por la consola la respuesta a la consulta "¿Existe algún tramo correspondiente al fin de semana?".

2. Construcción de tramos

Implementa un método de clase en la clase `TramoHorario` denominado `valueOf` que reciba como parámetro tres cadenas de texto y devuelva un objeto `TramoHorario` inicializado con los valores establecidos en las cadenas. En el caso de que alguna de las cadenas de texto no contenga un valor válido el método lanzará la ***excepción*** no comprobada o *runtime* `java.time.DateTimeException`.

- Para obtener un `DayOfWeek` a partir de una cadena se hará uso del método de clase `valueOf` que ofrece el enumerado. Este método recibe como parámetro una cadena de texto y devuelve la instancia de `DayOfWeek` que tenga como nombre dicha cadena de texto. El método notifica la excepción no comprobada o *runtime* `IllegalArgumentException` si la cadena de texto no coincide con ninguna de las etiquetas del enumerado. También lanza la excepción `NullPointerException` si la cadena de texto que se pasa como parámetro es igual a `null`.
- Para obtener un valor entero a partir de una cadena de texto se hará uso del método de clase `parseInt` de la clase `Integer`. Este método recibe como parámetro una cadena de texto y devuelve un `int`, si el formato de la cadena es correcto. El método notifica la excepción no comprobada o *runtime* `NumberFormatException` si la cadena de texto no representa un valor entero.

3. Clase Servicio

Un **servicio** representa el turno en el que un restaurante sirve a sus clientes. Se define como un tipo de datos inmutable que se caracteriza por las siguientes propiedades:

- tipo, definido en un enumerado (desayuno, comida, cena).
- fecha en la que se realiza el servicio (`LocalDate`).

En la construcción puede omitirse la fecha. En tal caso, se establece la fecha como la fecha actual.

Ejercicios:

a) Implementa la clase `Servicio` que se ajuste a la especificación anterior.

b) Implementa los métodos `equals/hashCode`. Se entiende que dos servicios son iguales si coinciden los valores de los dos atributos.

c) Implementa el ***orden natural*** de comparación (interfaz `Comparable`). Los servicios se ordenan, primero por orden ascendente de fecha, y en caso de empate, por tipo (se entiende que el desayuno va antes que la comida y ésta antes que la cena).

   NOTA: Los enumerados son comparables, su orden natural se corresponde con el orden de la declaración de los valores.

4. Clase Cita

Una *cita* representa la solicitud de asistencia al centro médico para ser atendido por un médico. Se corresponde con un tipo de datos inmutable que se caracteriza por las siguientes propiedades:

- Paciente que solicita la cita (cadena de texto que lo identifica).
- Fecha y hora en la que se quiere asistir al centro médico (`LocalDateTime`).
- Activa: propiedad booleana que toma el valor verdadero si el momento actual es anterior a la fecha y hora de la cita.

La clase ofrece dos constructores, uno de ellos recibe como parámetro el paciente y un objeto de tipo `LocalDateTime` que representa la fecha y la hora, y el otro constructor recibe como parámetro el paciente, un objeto `LocalDate` que representa la fecha y un entero para la hora (se entiende que los minutos de la hora es igual a 0).

Ejercicios:

a) Implementa la clase `Cita` que se ajuste a la especificación anterior.

b) Implementa un ***criterio de ordenación*** (interfaz `Comparator`) para la clase `Cita` en base a:

- Primero, por orden natural de la fecha y hora.
- Segundo, por orden natural del nombre del paciente (orden alfabético).

c) Redefine los métodos de la clase `Object`:

- `equals`/`hashCode` de manera consistente con el criterio de comparación del apartado anterior.
- `toString` siguiendo las recomendaciones de la asignatura.

d) Crea la clase `Programa2` que incluya la siguiente funcionalidad:

- Crea una lista de citas e introduce 4 citas: fecha actual (`LocalDate.now()`), una con fecha de ayer y dos con fecha de mañana. **Nota:** utiliza el método `plusDays` para calcular las fechas anteriores y posteriores a la actual.
- Aplicando procesamiento basado en *streams*:
  - Cuenta las citas del día de mañana.
  - Construye una lista con las citas de un paciente concreto.

  Muestra en ambos casos el resultado en la consola.

e) En la clase `Utilidades`, implementa un método de clase que reciba como parámetro una colección de citas y elimine todas las que no estén activas. Es requisito del ejercicio utilizar un ***iterador explícito***. Haz una prueba en `Programa2`.

f) En la clase `Utilidades`, implementa otro método de clase que construya un mapa ordenado a partir de una colección de citas recibida como parámetro. Las claves del mapa son los días (`LocalDate`), que estarán ordenadas cronológicamente (orden natural de las fechas). Cada día tendrá asociado una lista con las citas cuya fecha corresponda a ese día.

5. Clase Valoración

Una *valoración* se define como un tipo de datos inmutable que se caracteriza por las siguientes propiedades:

- usuario que realiza la valoración (cadena de texto).
- fecha de creación (`LocalDate`).
- nota (entero): puede tomar valores entre 1 y 5.
- comentario (cadena de texto). Esta propiedad es opcional.

La fecha no se establece como parámetro en la construcción. Se inicializa con el valor de la fecha actual.

Ejercicios:

a) Implementa la clase `Valoracion` que se ajuste a la especificación anterior.

b) Implementa el ***control de precondiciones*** en los constructores.

c) Implementa el método `toString` siguiendo las recomendaciones de la asignatura.

d) Crea la clase `Programa3` que incluya la siguiente funcionalidad:

- Crea una lista con 4 valoraciones.
- Utilizando una ***expresión lambda***, ordena la lista por nombre de usuario (orden alfabético). Muestra el resultado en la consola.
- Aplicando ***procesamiento basado en streams***:
  - Construye un conjunto con los usuarios que han realizado alguna valoración. Muestra el resultado por la consola.
  - Muestra por la consola la respuesta a la consulta "¿Todas las valoraciones tienen nota 5?".

6. Lista con filtro

Implementa un nuevo tipo de lista que tenga la capacidad de rechazar inserciones. Para realizar esta tarea recibe como parámetro del constructor un *predicado* (***programación funcional***) que se utilizará para controlar las inserciones. Solo los elementos que cumplan el predicado serán insertados en la lista. **Requisito**: la implementación debe apoyarse en una alguna de las colecciones que implementan listas estudiadas en la asignatura.

7. Clase Utilidades

a) Implementa un ***método de clase genérico*** que devuelva la intersección de dos conjuntos que se pasan como parámetro. Realiza una prueba de este método pasando como parámetro dos conjuntos de números.

b) Implementa un método de clase que construya una lista de fechas (`LocalDate`) a partir de una lista variable de cadenas (*argumento variable*). Para obtener una fecha a partir de una cadena se hará uso del método de clase `parse` que ofrece la clase `LocalDate`. Este método recibe como parámetro una cadena de texto y devuelve una instancia de `LocalDate`, si el formato de la cadena es correcto. El método, notifica la ***excepción*** no comprobada o *runtime* `DateTimeParseException` si no puede hacer la conversión. En ese caso, en la lista se introducirá un valor nulo que informa que la cadena correspondiente no ha podido ser transformada. El formato textual de las fechas debe ser "2022-05-18", esto es, "año-mes-día". Realiza una prueba de este método en `Programa4` pasando como parámetro la lista con las cadenas “2022-04-23”, “22-05-18”, “2022-30-05”, “2022-05-10”.

c) Implementa un ***método de clase genérico*** denominado `sonDisjuntos` que reciba como parámetro dos colecciones y devuelva un valor `boolean` indicando si no tienen elementos en común. Realiza una prueba de este método en `Programa4` pasando como parámetro dos conjuntos de números.

d) Implementa un ***método genérico*** que almacene en un fichero de texto los objetos de una lista. Cada objeto será representado en una línea de texto utilizando su representación `toString`. Si al escribir un objeto se produce un error, se volverá a reintentar la escritura hasta 5 veces. En caso de no conseguirlo, se mostrará por la salida de error el mensaje “No ha podido ser escrito el objeto en la posición X” y el procesamiento continuará con el siguiente elemento. Añade en la clase `Programa2` una prueba del método anterior pasando como parámetro la lista de citas del ejercicio 4.

Nota: es requisito utilizar la clase `java.io.FileWriter` para implementar la funcionalidad. Esta clase ofrece un constructor en el que se establece la cadena con el nombre del fichero que va a ser creado. El constructor puede lanzar la excepción `java.io.IOException`, si no se puede crear el fichero.

Para escribir cadenas de texto la clase `FileWriter` ofrece el método `write`. La ejecución del método `write` puede lanzar la excepción `java.io.IOException` si en la operación de escritura se produjo un error de entrada/salida. Una vez finalizada la escritura del fichero se utiliza el método `close()` para cerrar el manejador del fichero. La ejecución de este método también puede lanzar la excepción `java.io.IOException` si se produce un error de entrada/salida.