Nombre y Apellido: ..... N° Legajo: .....

Segundo Parcial de Programación Orientada a Objetos (72.33)

19/06/2019

Ejercicio 1	*Ejercicio 2*	*Ejercicio 3*	*Nota*	*Firma Docente*
/3	/3	/4		

- **<u>Condición mínima de aprobación: SUMAR 5 PUNTOS</u>.**
- **Las soluciones que no se ajusten al paradigma OO, no serán aceptadas.**
- **Las soluciones que no se ajusten estrictamente al enunciado, no serán aceptadas.**
- **Puede entregarse en lápiz.**
- **No es necesario escribir las sentencias `import`.**
- **Además de las clases solicitadas se pueden agregar las que consideren necesarias.**
- **Escribir en cada hoja Nombre, Apellido, Legajo, Número de Hoja y Total Hojas entregadas.**

Ejercicio 1

La clase **`CensureIterator`** modela un iterador que permite censurar una serie de elementos de la misma, reemplazando un elemento censurado por un sustituto que recibe en su constructor. Los elementos a censurar son seteados a través del método `censure`. El iterador deberá retornar en cada invocación a `next` un sustituto del elemento de la colección (si corresponde censurarlo) o al elemento original de la colección recibida (si no corresponde censurarlo) .

Implementar todo lo necesario para que, con el siguiente programa

```
public class CensureIteratorTester {

   public static void main(String[] args) {
       List<String> strings = new ArrayList<>();
       strings.add("One");
       strings.add("Two");
       strings.add("Three");
       CensureIterator<String> stringCensureIterator = new CensureIterator<>(strings, "XXXX");
       stringCensureIterator.censure("Two").censure("Four");
       iteratorPrinter(stringCensureIterator);

       Map<String, Integer> stringIntegerMap = new HashMap<>();
       stringIntegerMap.put("One", 1);
       stringIntegerMap.put("Two", 2);
       stringIntegerMap.put("Three", 3);
       stringCensureIterator = new CensureIterator<>(stringIntegerMap.keySet(), "XXXX");
       stringCensureIterator.censure("Two").censure("Four");
       iteratorPrinter(stringCensureIterator);

       Integer[] integers = new Integer[]{1, 2, 3};
       CensureIterator<Integer> integerCensureIterator = new CensureIterator<>(integers, -1);
       integerCensureIterator.censure(2).censure(4);
       iteratorPrinter(integerCensureIterator);
   }

   private static <T> void iteratorPrinter(Iterator<T> iterator) {
       while (iterator.hasNext()) {
           System.out.println(iterator.next());
       }
       System.out.println();
   }

}
```

se obtenga la siguiente salida

```
One
XXXX
Three

One
XXXX
Three

1
-1
3
```

Ejercicio 2

Un **multimapa** es un mapa que admite más de un valor para una misma clave. Se cuenta con la siguiente interfaz `MultiMap`.

```
public interface MultiMap<K, V> {

   /**
    * Agrega un par key,value al multimapa si el par no existe.
    */
   void put(K key, V value);

   /**
    * Cantidad de valores del multimapa.
    */
   int size();

   /**
    * Cantidad de valores del multimapa para la clave key.
    */
   int size(K key);

   /**
    * Elimina la clave del multimapa (con todos sus valores) si existe.
    */
   void remove(K key);

   /**
    * Elimina el valor value de la clave key si existe.
    */
   void remove(K key, V value);

   /**
    * Colección de valores de la clave key con el orden establecido.
    * Si la clave no existe, retorna una colección vacía.
    */
   Iterable<V> get(K key);

   /**
    * Colección de valores de la clave key con el orden inverso al establecido.
    * Si la clave no existe, retorna una colección vacía.
    */
   Iterable<V> getReverseOrder(K key);

}
```

Implementar todo lo necesario (sin modificar la interfaz) y completar los … para que, con el siguiente programa

```
public class MultiMapTester {

   public static void main(String[] args) {
       MultiMap<String,Integer> m = new MultiMapImpl<>(.....................................);
       m.put("hola", 4);
       m.put("hola", 2);
       m.put("hola", 3);
       m.put("chau", 4);
       m.put("chau", 5);
       m.put("adios", 6);
       System.out.println(m.size());
       System.out.println(m.get("hola"));
       System.out.println(m.getReverseOrder("hola"));
       m.remove("adios");
       m.remove("hola", 2);
       System.out.println(m.get("hola"));
       System.out.println(m.getReverseOrder("hola"));
       System.out.println(m.get("adios"));
       System.out.println(m.getReverseOrder("adios"));
       System.out.println(m.size());
   }

}
```

se obtenga la siguiente salida

```
6
[2, 3, 4]
[4, 3, 2]
[3, 4]
[4, 3]
[]
[]
4
```

Ejercicio 3

Se modela una **central de reservas para eventos** en base a las clases `BookingCentral`, `Event`, y `Reservation` que se presentan a continuación:

```
public class BookingCentral {

   protected Map<String, Event> events = new HashMap<>();
   protected List<Reservation> reservations = new ArrayList<>();

   public void book(String eventName, String person, int seats) {
       Event event = getEvent(eventName);
       Reservation reservation = getReservation(eventName, person);
       if (reservation != null) {
           throw new IllegalArgumentException("Person has an unconfirmed reservation for the
event.");
       }
       event.book(seats);
       reservations.add(new Reservation(person, event, seats));
   }

   public void confirm(String eventName, String person) {
       Reservation reservation = getReservation(eventName, person);
       if (reservation == null) {
           throw new IllegalArgumentException("Unknown reservation.");
       }
       reservation.confirm();
   }
```

```
    public void cancel(String eventName, String person) {
        Reservation reservation = getReservation(eventName, person);
        if (reservation == null) {
            throw new IllegalArgumentException("Invalid reservation.");
        }
        reservation.getEvent().cancel(reservation.getSeats());
        reservations.remove(reservation);
    }

    public void buy(String eventName, String person, int seats) {
        Event event = getEvent(eventName);
        event.book(seats);
        Reservation reservation = new Reservation(person, event, seats);
        reservation.confirm();
        reservations.add(reservation);
    }


    public void printConfirmations() {
        System.out.println("Tickets sold: ");
        for (Reservation reservation : reservations) {
            if (reservation.isConfirmed()) {
                reservation.print();
            }
        }
    }

    public void addEvent(Event event) {
        events.put(event.getName(), event);
    }

    private Event getEvent(String eventName) {
        Event event = events.get(eventName);
        if (event == null) {
            throw new IllegalArgumentException("Invalid event.");
        }
        return event;
    }

    private Reservation getReservation(String eventName, String person) {
        Event event = getEvent(eventName);
        for (Reservation reservation : reservations) {
            if (!reservation.isConfirmed() && reservation.getEvent().equals(event)
                    && reservation.getPerson().equals(person)) {
                return reservation;
            }
        }
        return null;
    }

}
```

```
public class Event {

    private String name;
    private int emptySeats;

    public Event(String name, int emptySeats) {
        this.name = name;
        this.emptySeats = emptySeats;
    }

    public String getName() {
        return name;
    }

    public void book(int seats) {
        if (seats > emptySeats) {
            throw new IllegalArgumentException("Event has not enoug empty places");
        }
        this.emptySeats -= seats;
    }
```

```
    public void cancel(int seats) {
        this.emptySeats += seats;
    }

    @Override
    public boolean equals(Object o) {
        // Implementación correcta del equals
    }

    @Override
    public int hashCode() {
        // Implementación correcta del hashCode
    }

}
```

```
public class Reservation {

    private String person;
    private Event event;
    private boolean confirmed;
    private int seats;

    public Reservation(String person, Event event, int seats) {
        this.person = person;
        this.event = event;
        this.confirmed = false;
        this.seats = seats;
    }

    public void confirm() {
        if (confirmed) {
            throw new IllegalStateException("Reservation already confirmed.");
        }
        confirmed = true;
    }

    public boolean isConfirmed() {
        return confirmed;
    }

    public Event getEvent() {
        return event;
    }
    public String getPerson() {
        return person;
    }

    public int getSeats() {
        return seats;
    }

    public void print() {
        System.out.println(event.getName() + " - " + person);
    }

}
```

Se pide implementar la clase **`EnhancedBookingCentral`** cuyo comportamiento evita que un cliente realice una reserva teniendo más de N reservas sin confirmar o más de M cancelaciones realizadas. Cada vez que un cliente hace una compra directa (invocación al método `buy`), se le eliminan las cancelaciones registradas (eventualmente esto lo vuelve a habilitar para hacer reservas, siempre que no supere la cantidad de reservas pendientes).

Implementar todo lo necesario (sin cambiar el código de las clases ya provistas) para que, con el siguiente programa de prueba:

```
public class EnhancedBookingCentralTest {

   public static void main(String[] args) {
       BookingCentral central = new EnhancedBookingCentral(2,1);  // N = 2, M = 1

       central.addEvent(new Event("Evento 1", 10));
       central.addEvent(new Event("Evento 2", 10));
       central.addEvent(new Event("Evento 3", 10));
       central.addEvent(new Event("Evento 4", 10));
       central.addEvent(new Event("Evento 5", 10));
       central.addEvent(new Event("Evento 6", 10));

       central.book("Evento 1", "Mariano", 1);
       central.book("Evento 2", "Mariano", 1);
       try {
           central.book("Evento 3", "Mariano", 1);
       } catch (BlacklistedCustomerException e) {
           System.out.println(e.getMessage());
       }
       central.buy("Evento 3", "Mariano", 1);

       try {
           central.book("Evento 4", "Mariano", 1);
       } catch (BlacklistedCustomerException e) {
           System.out.println(e.getMessage());
       }
       central.confirm("Evento 1", "Mariano");
       central.book("Evento 4", "Mariano", 1);
       central.confirm("Evento 4", "Mariano");
       central.cancel("Evento 2", "Mariano");
       try {
           central.book("Evento 5", "Mariano", 1);
       } catch (BlacklistedCustomerException e) {
           System.out.println(e.getMessage());
       }
       central.buy("Evento 5", "Mariano", 1);
       central.book("Evento 6", "Mariano", 1);
       central.printConfirmations();
   }

}
```

se obtenga la siguiente salida:

```
Can't book: Person cannot have more than 2 pending reservation/s.
Can't book: Person cannot have more than 2 pending reservation/s.
Can't book: Person cannot have more than 1 cancellation/s.
Tickets sold:
Evento 1 - Mariano
Evento 3 - Mariano
Evento 4 - Mariano
Evento 5 - Mariano
```