Nombre:..... ***Legajo:.....***

Segundo Parcial de Programación Orientada a Objetos

19 de Noviembre de 2010

Ej. 1	*Ej. 2*	*Ej. 3*	*Nota*

- **Condición de aprobación: Tener dos ejercicios calificados como B o B-.**
- **Los ejercicios que no se ajusten estrictamente al enunciado, no serán aceptados.**
- **No es necesario agregar las sentencias import.**
- **Además de las clases pedidas se pueden agregar las que se consideren necesarias.**

Ejercicio 1

Dado un conjunto de elementos se quiere obtener un mapa. Dicho mapa debe contener como valores los elementos del conjunto y como claves el resultado de aplicar cierta función a cada uno de esos elementos. Se piden dos soluciones:

a) Extender la interface **Set** (e implementarla) agregando un método que genere dicho mapa.

b) Implementar la clase **Sets** (similar a **Collections**) que ofrece un método de clase que genere dicho mapa.

En ambos casos se debe lanzar una excepción si hay una clave duplicada.

En cada caso defina las clases, interfaces y parametrización de métodos que considere necesaria.

Ejercicio 2

La clase **Expression** representa una expresión lógica, que puede ser evaluada en **true** o **false**:

```
public abstract class Expression {
	public abstract boolean evaluate();
}
```

Se cuenta con la subclase **SimpleExpression** que modela un operando:

```
public class SimpleExpression extends Expression{

	private boolean value;

	public SimpleExpression(boolean value) {
		this.value = value;
	}

	@Override
	public boolean evaluate() {
		return value;
	}

	public void setValue(boolean value) {
		this.value = value;
	}
}
```

Se desea agregar la posibilidad de construir expresiones que realicen operaciones lógicas (**and**, **or** y **not**). Para esto se pueden agregar métodos y/o propiedades a las clases **Expression** y/o **SimpleExpression** o bien crear clases nuevas, de forma tal que el siguiente ejemplo compile e imprima lo que se indica en los comentarios (NO SE PUEDE MODIFICAR EL CODIGO EXISTENTE, SOLO AGREGAR):

```
public class EvaluatorTest {
   public static void main(String[] args) {
      SimpleExpression exp1 = new SimpleExpression(true);
      SimpleExpression exp2 = new SimpleExpression(false);

      Expression exp3 = exp1.not();
      Expression exp4 = exp1.or(exp2);
      Expression exp5 = exp3.and(exp4);
      Expression exp6 = exp3.or(exp4);
      Expression exp7 = exp4.not();

      System.out.println(exp1.evaluate());  // true
      System.out.println(exp3.evaluate());  // false
```

```
      System.out.println(exp4.evaluate());  // true
      System.out.println(exp5.evaluate());  // false

      exp1.setValue(false);

      System.out.println(exp3.evaluate());  // true
      System.out.println(exp4.evaluate());  // false
      System.out.println(exp5.evaluate());  // false

      exp2.setValue(true);

      System.out.println(exp5.evaluate());  // true
   }
}
```

Ejercicio 3

Se modela una central de reservas para eventos en base a las clases **BookingCentral**, **Event**, y **Reservation** que siguen a continuación.

```
public class BookingCentral {

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

   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);
   }

   protected Event getEvent(String eventName) {
      Event event = events.get(eventName);
      if (event == null) {
```

```
            throw new IllegalArgumentException("Invalid event.");
         }

         return event;
      }

      protected 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;
   }

   public int hashCode() { // Código de hashcode correcto
   }

   public boolean equals(Object obj) { // Código de equals correcto
   }
}
```

```
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 **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).

```
public class EnhancedBookingCentralTest {
   public static void main(String[] args) {
      BookingCentral central = new EnhancedBookingCentral(2,1);

      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("A- Person cannot have more than 2 pending reservations.");
      }

      central.buy("Evento 3", "Mariano", 1);

      try {
         central.book("Evento 4", "Mariano", 1);
      } catch (BlacklistedCustomerException e) {
         System.out.println("B- Person cannot have more than 2 pending reservations.");
      }

      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("C- Person cannot make reservation: too many previous cancellations.");
      }

      central.buy("Evento 5", "Mariano", 1);
      central.book("Evento 6", "Mariano", 1);
      central.printConfirmations();
   }
}
```

La salida de la ejecución del programa es:

```
A- Person cannot have more than 2 pending reservations.
B- Person cannot have more than 2 pending reservations.
C- Person cannot make reservation: too many previous cancellations.
Tickets sold:
Evento 1 - Mariano
Evento 3 - Mariano
Evento 4 - Mariano
Evento 5 - Mariano
```