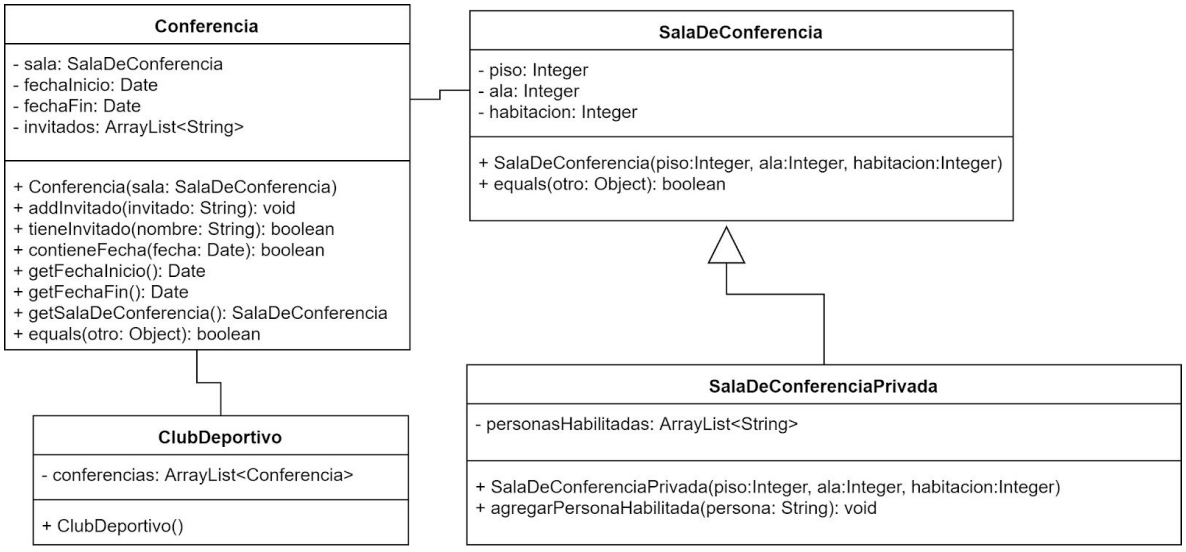
Examen 111Mil – Club Deportivo

En el Club Deportivo "111mil" varias personas (socios, futbolistas, entrenadores, dirigentes), utilizan las salas de conferencia para dar conferencias a periodistas y aficionados. Julieta, una de las dirigentes, realizó el curso de 111mil y se le ocurrió informatizar el sistema de reserva de salas de conferencia. Las salas se alquilan y pueden ser públicas (cualquier socio del club puede utilizarlas) o privadas (solo pueden ser utilizados por ciertas personalidades del club).

Ejercicio 1. Implementar desde el diagrama de clases

Julieta realizó un diagrama UML preliminar del sistema y nos pidió que implementemos la clase *SalaDeConferenciaPrivada* según el diagrama. Recuerde inicializar la lista "personasHabilitadas" en el constructor.



Posible solución

```
package edu.cientoonce.club.saladeconferencia;

import java.util.ArrayList;

public class SalaDeConferenciaPrivada extends SalaDeConferencia {
    private ArrayList<String> personasHabilitadas;

    public SalaDeConferenciaPrivada(Integer piso, Integer ala,
Integer habitacion) {
        super(piso, ala, habitacion);
        this.personasHabilitadas = new ArrayList<String>();
    }
```

```
    public void agregarPersonaHabilitada(String persona) {
        this.personasHabilitadas.add(persona);
    }
}
```

Ejercicio 2. Implementar un método a partir de un enunciado

Julieta instaló en cada *SalaDeConferencia* una cerradura electrónica conectada a Internet. La única forma de abrir la cerradura es teniendo acceso a una *Conferencia* válida. Por este motivo, agregó un método "abrir" en la clase *SalaDeConferencia*, que recibe una *Conferencia* y verifica que dicha *Conferencia* sea en la sala que se desea abrir. Además, verifica que la fecha y hora coincidan con la fecha actual (esto último lo hace la clase *Conferencia*):

```
public boolean abrir(Conferencia conferencia) {
  Date fechaActual = new Date();
  return conferencia.getSalaDeConferencia().equals(this)
         && conferencia.contieneFecha(fechaActual);
}
```

La *SalaDeConferenciaPrivada* es un poco más compleja. Además de los chequeos anteriores, se debe verificar que alguno de los invitados esté habilitado para abrir la puerta.

Sobreescriba el método *abrir* en la clase *SalaDeConferenciaPrivada* y, mediante el uso de **super**, reutilice el chequeo que ya se encuentra en la clase *SalaDeConferencia*.

Posible solución

```
@Override
public boolean abrir(Conferencia conferencia) {
  for (String persona : this.personasHabilitadas)
    if (conferencia.tieneInvitado(persona))
      return super.abrir(conferencia);
  return false;
}
```

Ejercicio 3. Implementar un método a partir de un enunciado

Debido a un error de diseño, actualmente en el sistema dos conferencias pueden tener asignada la misma sala y sus fechas solapadas, generando un gran malestar a los usuarios. Peor aún, dos personas pueden estar invitadas a conferencias solapadas en distintas salas, haciendo imposible que dichas personas estén en dos lugares al mismo tiempo.

Con el tiempo, Julieta fue limpiando manualmente el sistema de conferencias inválidas, pero necesita una funcionalidad para verificar si una conferencia "colisiona" con otra, es decir, tiene que chequear:

- Si alguna de las fechas de una conferencia está en el rango de la otra conferencia o viceversa y:
  - Tienen asignada la misma *SalaDeConferencia*
  - Tienen asignada distinta *SalaDeConferencia*, pero comparten algún invitado

Recuerde que el método *contieneFecha* de Conferencia chequea si una fecha dada está en el rango de tiempo de la Conferencia. Es decir, retorna `true` solo si la fecha dada por parámetro es mayor a la fecha de inicio de la conferencia y menor a la fecha de fin.

Implemente el método **`public boolean colisiona(Conferencia otra)`** en la clase *Conferencia*, el cual verifica si un objeto *Conferencia* colisiona con otro objeto *Conferencia* pasado por parámetro.

Posible solución
Clase Conferencia

```
public boolean colisiona(Conferencia otra) {
  boolean colisionaFecha = otra.contieneFecha(this.getFechaInicio()) ||
                           otra.contieneFecha(this.getFechaFin()) ||
                           this.contieneFecha(otra.getFechaInicio()) ||
                           this.contieneFecha(otra.getFechaFin());
  if (!colisionaFecha)
    return false;
  if (sala.equals(otra.getSalaDeConferencia()))
    return true;
  for (String invitado : this.invitados)
    if (otra.tieneInvitado(invitado))
      return true;
  return false;
}
```

Ejercicio 4. Seguimiento de código

Julieta agregó a la *SalaDeConferencia* un listado de dispositivos disponibles de tipo **`ArrayList<String>`**, que se inicializa en el constructor de *SalaDeConferencia*. Por ejemplo, dicha lista puede contener "proyector", "presentador", "televisor", etc. Escribió rápidamente 2 métodos asociados a la lista de dispositivos, pero dejó los métodos con nombres poco representativos, al igual que algunas variables.

Recordar que:

- El método *isEmpty* de la clase String verifica si el objeto String es una cadena de caracteres vacía.

- El método *compareTo* de String compara dos strings de forma alfabética. Por ej: si a y b son dos Strings y se hace a.compareTo(b), el método retorna cero si son iguales, un número mayor a cero si **a** es mayor que **b** y un número menor a cero si **a** es menor que **b**.
- El método *add(pos, objeto)* de ArrayList inserta "objeto" en la posición "pos" corriendo el elemento existente en dicha posición "pos" y todos los subsiguientes hacia la derecha. Por ej: hacer add(1, 9) sobre la lista [1, 2, 3, 4] resulta en la lista modificada [1, 9, 2, 3, 4]

Clase SalaDeConferencia

```
private boolean metodoMisterioso2(String variable1) {
  return variable1 != null && !variable1.isEmpty();
}

public boolean metodoMisterioso1(String variable1) {
  if (!metodoMisterioso2(variable1))
    return false;

  for (int i = 0; i < this.dispositivos.size(); i++) {
    String current = this.dispositivos.get(i);

    Integer comparacion = current.compareTo(variable1);

    if (comparacion == 0)
      return false;

    if (comparacion > 0) {
      this.dispositivos.add(i, variable1);
      return true;
    }
  }
  this.dispositivos.add(variable1);
  return true;
}
```

Sabiendo que *getDispositivos* de la clase *SalaDeConferencia* retorna la lista de dispositivos en la *SalaDeConferencia*. ¿Qué imprimirá el programa al ejecutar el siguiente código?

```
 public static void main(String[] args) {
        SalaDeConferencia sala1 = new SalaDeConferencia(0, 0, 0);
        sala1.metodoMisterioso1(null);
        sala1.metodoMisterioso1("");
        System.out.println(sala1.getDispositivos());

        SalaDeConferencia sala2 = new SalaDeConferencia(1, 1, 1);
```

```
        sala2.metodoMisterioso1("proyector-1");
        sala2.metodoMisterioso1("proyector-2");
        sala2.metodoMisterioso1(null);
        sala2.metodoMisterioso1("tv");
        sala2.metodoMisterioso1("appletv");
        sala2.metodoMisterioso1("microfono");
        System.out.println(sala2.getDispositivos());

        SalaDeConferencia sala3 = new SalaDeConferencia(2, 2, 2);
        sala3.metodoMisterioso1("appletv");
        sala3.metodoMisterioso1("proyector");
        sala3.metodoMisterioso1("");
        sala3.metodoMisterioso1("tv");
        sala3.metodoMisterioso1("appletv");
        sala3.metodoMisterioso1("microfono");
        sala3.metodoMisterioso1("proyector");
        System.out.println(sala3.getDispositivos());
    }
```

Solución

```
[]
[appletv, microfono, proyector-1, proyector-2, tv]
[appletv, microfono, proyector, tv]
```

Ejercicio 5. Consulta SQL

El encargado del club tiene un problema grave de inventario de proyectores en el piso 1, ala 2. Por ello, desea encontrar todas las *Conferencias* cuyas *SalaDeConferencias* de dicho piso y ala posean 2 o más proyectores (el encargado considera que debería haber uno solo por sala). Para no molestar, requiere ver la fecha de inicio de las conferencias, y así poder recuperar los proyectores antes de que comiencen las mismas.

Dado el diagrama de entidad-relación parcial, **escriba la consulta SQL que liste la fecha de inicio de la conferencia, el número de habitación de la sala y la cantidad de proyectores**, filtrando sólo aquellas salas del piso 1 y ala 2 que tienen más de 2 proyectores. Los proyectores se identifican simplemente verificando que el nombre del dispositivo empiece con “proyector” (puede utilizar la notación LIKE ‘proyector%’ de SQL para verificar si el campo comienza con “proyector”).

Además, dadas las siguientes tuplas de ejemplo, **determinar el resultado de la consulta**.

Conferencia			
1	2019-01-01 15:00:00	2019-01-01 16:00:00	1
2	2019-03-26 08:00:00	2019-03-26 12:00:00	3
3	2019-10-09 10:00:00	2019-10-10 10:30:00	2

SalaDeConferencia			
1	1	2	101
2	0	2	207
3	1	2	198

Dispositivo		
1	proyector-1	1
2	proyector-9	1
3	tv	1
4	proyector-2	3
5	proyector-5	3
6	proyector-7	3

Posible solución

```
SELECT  conferencia.fechaInicio,
        sala.habitacion,
        count(*) as proyectores
FROM  Conferencia conferencia
INNER JOIN SalaDeConferencia sala ON (conferencia.idSalaDeConferencia
= sala.idSalaDeConferencia)
INNER JOIN Dispositivo dispositivo ON (dispositivo.idSalaDeConferencia
= sala.idSalaDeConferencia)
WHERE sala.piso = 1
  AND sala.ala = 2
  AND dispositivo.nombre LIKE 'proyector%'
```

```
GROUP BY conferencia.fechaInicio, sala.habitacion
HAVING proyectores >= 2
```

Resultado de la consulta:

"2019-01-01 15:00:00" "101" "2"
"2019-03-26 08:00:00" "198" "3"