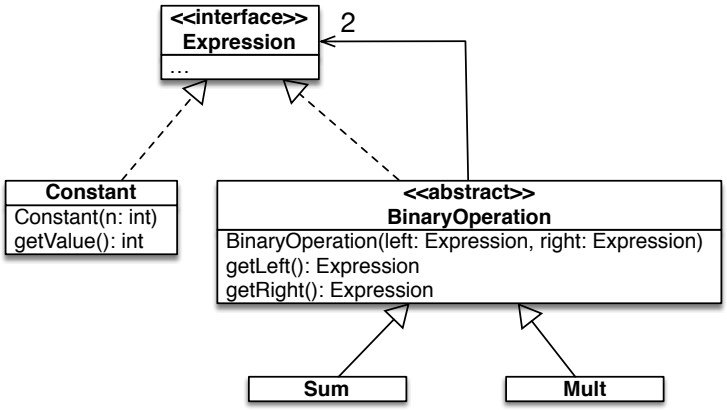
Problema 1. (1 punto)

¿Por qué gran parte de las novedades de Java están dirigidas a simplificar la programación concurrente?

Problema 2. (5 puntos)

Dado el siguiente diseño:



Este diseño permite construir expresiones aritméticas enteras formadas per constantes, sumas y productos. Per ejemplo, la expresión **2 + (4 * 8)** se construiría como:

```
Expression exp = new Sum(new Constant(2),
                         new Mult(new Constant(4),
                                  new Constant(8)))
```

Como se quieren implementar bastantes operaciones sobre expresiones, se ha pensado en aplicar el **patrón visitor** para tener una infraestructura que las permita implementar.

Uno de los visitantes concretos que se quiere construir es el que permite calcular el valor de una expresión y que se utilizaría como:

```
Evaluator evaluator = new Evaluator();
exp.accept(evaluator);
int evaluator.getEvaluation(); // Hauria de tornar 34
```

Se pide que añadáis la infraestructura que se necesita para implementar el patrón visitor y que implementéis el visitante `Evaluator` mostrado.

Problema 3. (4 puntos)

En una aplicación de calendario se dispone de la clase inmutable `CalendarItem`:

```
public class CalendarItem {

    private final String name;
    private final Place place;
    private final DayOfYear date;
    private final HoursMinutes begin;
    private final HoursMinutes end;
    private final List<Contact> others;

    public CalendarItem(String name, Place place, DayOfYear date,
                        HoursMinutes begin, HoursMinutes end,
                        List<Contact> others) {

        // Copies all parameters to fields

    }
}
```

Esta clase se apoya en otras que también son inmutables:

- `Place`: representa un lugar y se crea con un `String`. Su constructor controla que sea correcto (ni vacío ni `null`).
- `DayOfYear`: representa una fecha (día, mes, año) y se crea con tres enteros. Su constructor controla que sea una data correcta.
- `HoursMinutes`: representa una hora (horas, minutos) y se crea con dos enteros. Su constructor controla que sea una hora correcta. Esta clase implementa `Comparable<HoursMinutes>`.
- `Contact`: representa un contacto y se crea con un `String`. Su constructor controla que sea correcto (ni vacío ni `null`).

Se desea posibilitar la creación de instancias de la clase de una manera más fluida, como por ejemplo:

```
CalendarItem ci = CalendarItem.create("Segon Parcial AmpBBDDiEP")
                    .in(new Place("P2.01+P2.09+P2.11"))
                    .at(new DayOfWear(29, 5, 2018))
                    .from(new HourMinutes(12, 0))
                    .to(new HourMinutes(14, 0))
                    .with(new Contact("Marta Oliva"))
                    .with(new Contact("Toni Granollers"))
                    .build();
```

Además, la construcción, que deberá permitir invocar los métodos en cualquier orden, deberá comprobar las siguientes propiedades:

- Los métodos `in`, `at`, `from` i `to` se han de invocar una sola vez.
- El método `with` puede no invocarse o bien invocarse tantas veces como se quiera
- Ningún método puede invocarse con un parámetro `null`
- La hora indicada en `from` ha de ser anterior que la indicada en `to`

En caso de que alguna de estas propiedades no se cumpla, se lanzará la excepción no comprobada `IllegalStateException`. Para simplificar no es necesario que la excepción contenga un mensaje con la causa del problema. **Implementad esta forma de crear instancias**.

Si se quisiera tener diferentes formas de crear instancias de forma flioda que permitieran deducir los valores de algunos parámetros (p.e. considerando que, por defecto, les duraciones son de una hora y, por tanto, se puede deducir la hora de final a partir de la hora inicial), ¿cómo estructuraríais las clases de la vuestra solución? Mostrad el diagrama de clases con los métodos relevantes. **No habéis de implementar nada en este apartado.**