Guía Práctica de Laboratorio
Sesión 4: INTRODUCCIÓN A LA PROGRAMACIÓN
MLBC - VACJ - CJFV

En esta sesión,

- debes analizar cuales de las caracteristicas que describen a un tipo de objeto son ocultas y cuales no
- implementar los métodos REQUERIDOS de las clases que se describen en esta práctica

Ejemplo

Considera el siguiente contexto:
Contexto El binomio de Newton es un algoritmo que permite calcular una potencia cualquiera de un binomio, para ello se emplean los coeficientes binomiales, que a su vez utilizan los datos básicos de un polinomio. Para ello recordemos la forma de un binomio de Newton: $(a+b)^n$, que son los datos necesarios mínimos para aplicar el algoritmo. Cada binomio:

1. debería poderse mostrar en formato reducido: $(a+b)^n$
2. debería poderse mostrar de forma extendida: $a^n + a^{n-1} * b..b^n$

Modelo El modelo se observa en la figura 1

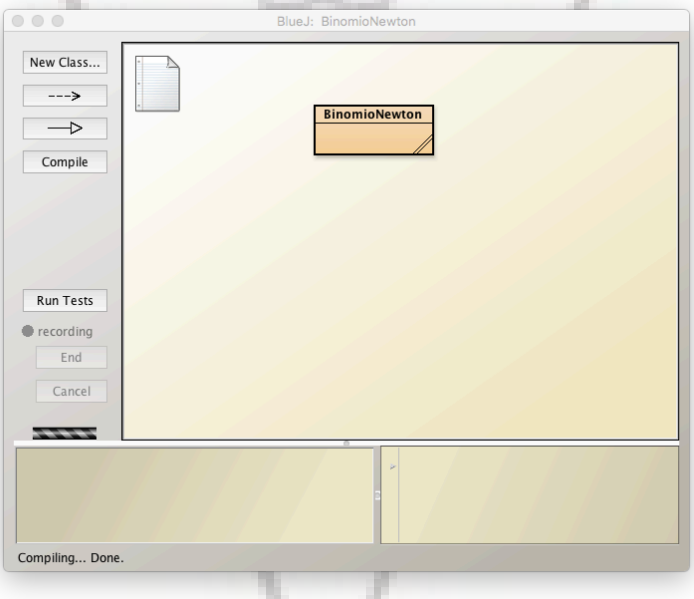

Figura 1: Ejemplo del modelo de clases

Qué características son ocultas? en este caso lo que se necesita proteger de accesos externos son los atributos del objeto, por lo que se pone *private* a los mismo.
Cómo lo hace? se pide definir el conjunto de instrucciones necesarias para indicar cómo un Binomio de Newton se muestra en formato reducido: esquematizando lo que se pide es practicamente mostrar así: $(a+b)^n$ o lo más parecido que en caso de la computadora seria así: `(a+b)^n`

```
1  /**
2   * Modelo que identifica los datos minimos para
3   * aplicar el algoritmo del Binomio de Newton
4   *
5   * @author MLBC
6   * @version 06.03.2019
7   */
8  public class BinomioNewton{
```

```
    private int a;
    private int b;
    private int n;
    // Constructor
    public BinomioNewton(int a, int b, int n){
        this.a = a;
        this.b = b;
        this.n = n;
    }
    // firma de metodo mostrar en formato reducido
    public String mostrar(){
        String reporte;
        reporte = "(" + a + " + " + b + ")^" + n;
        return reporte;
    }
    // firma de metodo mostrar en formato extendido
    public String mostrarExt(){
        return "";
    }
}
```

Debes asegurarte que el modelo compile, cada vez que haces una modificación. Recomendación es que valides cada método que implementas.

NOTA: Antes de programar piensa, esquematiza tu solución y recién pasa a código, esto te ayudara a organizar tu cerebro (desarrollar lógica) sin preocuparte por los detalles del lenguaje de programación.

TAREA

1. **Complejo** Los números complejos incluyen todas las raíces de los polinomios, a diferencia de los reales. Todo número complejo puede representarse como la suma de un número real y un número imaginario (que es un múltiplo real de la unidad imaginaria, que se indica con la letra i). Por ejemplo: 5 + 4i es un número complejo.

   Considera el modelo para poder

   a) sumar,

   b) restar,

   c) multiplicar y

   d) mostrar números complejos.

   El siguiente código tiene lo mínimo necesario para representar un objeto de tipo Complejo:

```
class Complejo{
    private double real;
    private double imaginario;
    public Complejo(double real, double imaginario){
      this.real      = real;
      this.imaginario = imaginario;
    }
    public Complejo sumar(Complejo otro){
      return null; // metodo OPA
    }
    public Complejo restar(Complejo otro){
      return null; // metodo OPA
    }
    public Complejo multiplicar(Complejo otro){
      return null; // metodo OPA
    }
    public String mostrar(){
      return null; // metodo OPA
    }
}
```

El siguiente pedazo de código muestra a objetos de la clase Complejo respondiendo a mensajes acordes al comportamiento ofrecido:

```
Complejo num1, num2;
Complejo num3, num4, num5;
String num;
num1 = new Complejo(3.0, 5.5);
num2 = new Complejo(4.0, 7.0);
num3 = num1.sumar(num2);
num4 = num1.multiplicar(num3);
num5 = num1.restar(num4);
num  = num3.mostrar();
```

Se han declarado cinco objetos de clase **Complejo**: *num1, num2, num3, num4 y num5*. *num1* representa al complejo *3.0 + 5.5i*, *num2* representa a *4.0 + 7.0i*. Al finalizar este pedazo de código el:

- *num3* representaría al complejo *7.0 + 12.5i*.
- *num4* representaría al complejo *-47.75 + 76i*.
- *num5* representaría al complejo *50.75 - 81.5i*.
- *num* representa a la cadena *"7.0 + 12.5i"*

Implementa los métodos: sumar, restar, multiplicar y mostrar de la clase Complejo.

2. **Vector** Un vector puede utilizarse para representar una magnitud física, quedando definido por un módulo y una dirección u orientación. Su expresin geométrica consiste en segmentos de recta dirigidos hacia un cierto lado, asemejándose a una flecha.

   Considera el modelo para que se pueda:

   a) sumar con otro vector

   b) multiplicar con otro vector

   c) calcular los grados del vector considerando su dirección

   d) decir a que cuadrante del eje de coordenadas el vector apunta

   e) calcular la magnitud del vector

   El modelo a continuación representa al vector $\overrightarrow{AB}$

```
/** clase que representa a un vector entre el punto A y B, con direccion
*   de A a B, tambien representado AB con direccion ->
**/
class Vector{
      private int ptoAX;
      private int ptoAY;
      private int ptoBX;
      private int ptoBY;
      public Vector(int ptoAX, int ptoAY, int ptoBX, int ptoBY){
        this.ptoAX = ptoAX;
        this.ptoAY = ptoAY;
        this.ptoBX = ptoBX;
        this.ptoBY = ptoBY;
      }
      public Vector sumar(Vector otro){
        return null;
      }
      public Vector multiplicar(Vector otro){
        return null;
      }
      public double calcularGrados(){
        return 0.0;
      }
```

```
24        public String ubicarCuadrante(){
25          return null;
26        }
27        public double calcularMagnitud(){ // esto es equivalente al modulo
28          return 0.0;
29        }
30  }
```

Para comprender mejor el modelo y cómo éste se usa, te presentamos el siguiente pedazo de código que muestra un ejemplo particular:

```
1   Vector vec1, vec2, vec3;
2   double grados, magnitud;
3   vec1 = new Vector(1, 1, 3, 6);
4   vec2 = new Vector(2, 2, 5, 6);
5   vec3 = vec1.sumar(vec2);
6   grados = vec3.calcularGrados();
7   magnitud = vec2.calcularMagnitud();
```

Se han declarado tres objetos de clase **Vector**: *vec1, vec2 y vec3. vec1* representa al vector $\overrightarrow{AB}$ donde A es el punto *(1,1)* y B es el punto *3,6*; *vec2* representa al vector $\overrightarrow{CD}$ donde C es el punto *(2,2)* y D es el punto *5,6*

Al finalizar este pedazo de código el:

- *vec3* representaría al vector $\overrightarrow{XY}$ donde X es el punto *(1,1)* e Y es el punto *6,10*
- *grados* representaría al ángulo en grados *29.054604099077146*.
- *magnitud* representaría al número real *5.0*.

Implementa los métodos: sumar, calcularGrados y calcularMagnitud de la clase Vector.

AYUDA: para implementar estos métodos, revisa la clase *Math* de la biblioteca de *java.lang*, ingresa a *https://docs.oracle.com/javase/7/docs/api/*

3. **Cuentas, Cuentas** El dinero es algo que mueve al mundo, y las instituciones que regulan y manejan este concepto son los bancos, es así que ellos permiten a las personas guardar de manera segura el dinero que tiene a través de Cuentas Bancarias, de las cuales se tiene: el numero de cuenta que generalmente es un numero grande, el saldo que tiene, el cliente y la moneda de la cuenta (Bs., $us). Refleja estos datos mínimos en un modelo Orientado a Objetos.

   En un futuro, es posible:

   a) depositar un monto en la cuenta

   b) retirar un monto de la cuenta

   c) transferir un monto a otra cuenta

   d) verificar si el numero de la cuenta es igual a *nroCta*

   e) consultar el saldo de la cuenta

   Con este contexto, se ha modelado una Cuenta Bancaria como sigue:

```
1
2  class CuentaBancaria{
3    private String numeroCuenta;
4    private double saldo;
5    private String cliente;
6    private String moneda;
7
8    public CuentaBancaria(String numeroCuenta, String cliente, String moneda){
9       this.numeroCuenta = numeroCuenta;
```

```
10        this.cliente       = cliente;
11        this.moneda        = moneda;
12        saldo              = 0.0;
13     }
14
15     public void depositar(double monto){
16     }
17     public boolean retirar(double monto){
18        return false;
19     }
20     public boolean transferir(double monto, String otroNroCuenta){
21        return false;
22     }
23     public boolean verificarNroCuenta(String numeroCuenta){
24        return false;
25     }
26     public double consultarSaldo(){
27        return 0.0;
28     }
29  }
```

Para comprender mejor el modelo y cómo éste se usa, te presentamos el siguiente pedazo de código que muestra un ejemplo particular:

```
1   CuentaBancaria cta1, cta2;
2   double saldo;
3   cta1 = new CuentaBancaria("1000785436", "Luis Choque", "Bs");
4   cta2 = new CuentaBancaria("1030565886", "Clark kent", "$us");
5   saldo  = cta1.consultarSaldo();
6   cta1.depositar(100.0);
7   saldo  = cta1.consultarSaldo();
8   saldo  = cta2.consultarSaldo();
```

Se han declarado dos objetos de clase **CuentaBancaria**: *cta1 y cta2*.

cta1 representa a la cuenta bancaria cuyo numero de cuenta es *1000785436*, le pertenece al cliente *Luis Choque* y la cuenta esta en *Bs.*.

cta2 representa a la cuenta bancaria cuyo numero de cuenta es *1030565886*, le pertenece al cliente *Clark kent* y la cuenta esta en *$us.*.

Al finalizar este pedazo de código el saldo es consultado tres veces a distintos objetos, en el tiempo el saldo tomaría valores de:

- primera consulta *saldo* es *0.0*
- segunda consulta *saldo* es *100.0*
- tercera consulta *saldo* es *0.0*

Implementa los métodos: depositar y consultarSaldo de la Cuenta Bancaria.