

CUVL – Parcial de Programación III

Alumno: Montes, Lucas

Fecha: 02/05/23

Ejercicio 1

1. Defina polimorfismo y sus principales ventajas.
2. Utilizando únicamente dos clases, dar un ejemplo de sobrecarga (overload) y de sobreescripción (overwrite).
3. Marcar la respuesta correcta: "la diferencia entre una interfaz y una clase abstracta es que..."
 - a. La interfaz es exclusiva para objetos que comparten funcionalidad pero son de distinta jerarquía.
 - b. En la interfaz únicamente se declaran las firmas y en la clase abstracta también es posible implementar funciones.
 - c. La interfaz no puede heredar de otra interfaz pero una clase abstracta si puede heredar de otra clase abstracta.
 - ☒ d. Las respuestas a y b son correctas.
 - e. Ninguna de las respuestas anteriores es correcta.
4. ¿Qué es una excepción? ¿Qué ventajas presenta el uso de excepciones respecto de trabajar con códigos de error?
5. ¿Qué es un MockObject? ¿En qué casos cree conveniente su utilización?

Ejercicio 2

Una mueblería desea diseñar un sistema que le permita gestionar sus ventas. La mueblería, por ahora, sólo fabrica sillas, mesas y sillones. De cada mueble fabricado se conocen sus medidas, alto, largo y ancho, todas expresadas en centímetros, su color, y precio. En el caso de los sillones se puede definir el material con el cual están tapizados. Las sillas pueden o no estar tapizadas; en caso de que lo estén se debe definir el material con el cual están tapizadas. Las mesas definen la cantidad de patas, material de la tabla, forma (redonda, cuadrada, rectangular).

Con el objetivo de incrementar sus ventas, la mueblería crea promociones que involucren conjunto de muebles a los que denomina combos. La mueblería asigna a cada combo un nombre y un porcentaje de descuento a aplicar sobre el precio total de los muebles que lo componen. Algunos ejemplos de combo pueden ser:

- Comedor Estándar: compuesto por una mesa y 4 sillas
- Comedor Extendido: compuesto por una mesa y 6 sillas
- Living E1: 2 sillones y 1 mesa chica
- Living L1: 3 sillones

Tener en cuenta las distintas combinaciones que los clientes pueden realizar. Un cliente puede comprar los muebles por separado o bien optar por un combo. También pueden comprar un combo y además algún otro mueble por separado.

Se pide

- a. Diagrama de clases de la solución. Debe incluir todos los métodos y atributos.
- b. Codificar y demostrar el uso de los métodos que permitan a un cliente obtener el precio de un pedido que involucre:
 - i. uno o más muebles
 - ii. uno o más combos

③ ~~Arbolito~~

③ Muebleria

- muebles : ArrayList<Mueble>
- Combo : Hashtable<String, Combo>
- + ~~Crear Combo (Combo c) : void~~
- + ~~Crear Mueble (Mueble m) : void~~
- + Crear Promocion (Promocion p) : void
- + Crear Mueble (Mueble m) : void
- + Crear Combo (Combo c) : void

③ Cliente

- combo : ArrayList<Combo>
- nombre : String
- precio : Integer
- + agregarCombo (Combo c) : void
- + RealizarCombo (Combo c) : void
- + verPrecio (Combo c) : void

③ Mueble

- nombre : String
- alto : Double
- largo : Double
- ancho : Double
- color : String
- precio : Double

③ Combo

③ Combo

- nombre : String
- descuento : Double
- Combo : ArrayList<Mueble>

- + ~~Crear Promocion (Promocion p) : void~~
- + Calcular Descuento (Mueble m) : Double
- + agregar Mueble

③ Madera

- Cant. Pks : Integer
- Forma : String
- Material : String

③ Sillon

③ Silla

- tieneTapa : boolean

③ Tapiz

- + tapizarMueble (Mueble m) : void

Lucas Navarro

```
public abstract class Yheble implements compra {
```

```
private nombre String nombre;  
private double alto;  
private double largo;  
private double ancho;  
private String color;  
private double precio;
```

```
public getNombre String getNombre() {  
    return nombre;  
}
```

```
public void setNombre(String n) {  
    this.nombre = n;  
}
```

```
public double void setHebidas(double a, double l, double an) {  
    this.alto = a;  
    this.largo = l;  
    this.ancho = an;  
}
```

```
public double get Hebidas Alto() {  
    return alto;  
}
```

```
public double get largo() {  
    return largo;  
}
```

```
public double get Ancho() {  
    return ancho;  
}
```

```
public String get Color() {  
    return color;  
}
```

```
public void setColor(String c) {  
    this.color = c;  
}
```



```
public double getPrecio() {
    return precio;
}
```

```
public void setPrecio(double p) {
    this.precio = p;
}
```

* Falta sobrescribir el método toString.

```
public class Mesa extends Mueble {
    private Integer cant.Patas;
    private String forma;
    private String material;
}
```

// getters y setters.

```
public class Sillon extends Mueble implements Tapizex {
```

```
    public void tapizarMueble() {
        // tapizo el sillón
    }
```

```
public class Silla Silla extends Mueble implements Tapizex {
    private boolean tiene tapiz;
    public void tapizarMueble() {
```

```
        if (!this.tieneTapiz) {
```

```
            System.out.println("No tiene tapizado");
        }
```

```
        // tapizo la silla
    }
```

Se podría usar una excepción

```

public Interface tapizar {
    public void tapizarMueble();
}

```

```

public class Combo implements Compra {
    private String nombre;
    private double descuento;
    private ArrayList < Mueble > combo;

```

```

    public double calcularDescuento ( Mueble m ) {
        double calc =
return m.getPrecio() * this.descuento / 100
        return m.getPrecio - calc;
    }

```

```

    public void agregar Mueble ( Mueble m ) {
        this.combo.add(m);
    }

```

```

    public void verCombo() {
        for (int i=0; i<this.combo.size(); i++) {
            System.out.println (this.combo.get(i));
        }
    }

```

// getters y setters
 # Faltó el método toString donde al sobrescribir este
 método muestro el nombre, descuento, y invoco al método
 verCombo donde muestro el detalle de los muebles que lo
 componen.

```

public class Cliente {
    private String nombre;
    private Integer nroCliente;
    private Compra ArrayList < Compra > compra;

```

```

    public void agregarCarrito ( Compra c ) {
        this.compra.add(c);
    }

```



```

public void verPrecio (Combo c) {
    System.out.println(c.tostring);
}

```

```

public void ver
}

```

```

public class Huelteria {
    private ArrayList < Huelde > Hueldes;
    private ArrayList Hashtable < String, Combo > Combos;
    private Hash set < Cliente > Clientes

```

```

public void crearPromocion (Combo c) {
    if (this.Combos.contains(c)) {
        System.out.println("Combo ya creado");
    }
    this.Combos.set(c.getNombre, c);
}

```

Debería
poter usar
una excepción
de tipo checked

```

public void crearMarcha (Huelde m) {
    this.hueldes.add(m)
}

```

```

public void realizarPedido (Cliente c) {
}

```

2) public class Animal {

public void sonido () {

system.out.println("No sabemos que sonido hace un animal"),

}

public void sonido (String msg) {

system.out.println(msg);

}

}

public class Perro extends Animal

Override

public void sonido () {

system.out.println("guau guau");

}

1) ~~el concepto de polimorfismo es la capacidad de un programa para aceptar diferentes formas de datos~~

El polimorfismo es uno de los conceptos principales de POO al que nos da la posibilidad de escribir varios métodos de distintas formas, por ejemplo, en el ejemplo de la pregunta 2, si creamos más clases que hereden de la clase Animal, podríamos sobrescribir el método sonido en cada una de ellas, esto es uno de los ventajas que nos da el polimorfismo, ya que nos permite utilizar el método sonido en una clase sin importar que clase estamos llamando. y si en un futuro se quiere agregar una clase que utilice dichos métodos, no nos va a afectar a nuestros códigos.

por lo tanto esto lo hace más mantenible y rentable
nuestro código.

5) ~~Una excepción~~ Un MockObject se suele utilizar en los
testing unitarios. Es un objeto, que suplanta una clase que aún
no fue codificada, esto nos sirve para poder ganar tiempo
y poder ir testeando lo que tenemos de código hasta el
momento.

4)