Trabajo Práctico IV: Polimorfismo

Principio de sustitución de Liskov.
Concepto de Polimorfismo.
Concepto de enlace tardío (Late Binding).
Implementación de polimorfismo.
Herencia y modificador de acceso de miembros sobreescritos.
Acceso a miembros sobreescritos
El verdadero significado de protected.
Concepto de contrato. Definición.
Precondiciones y postcondiciones.
Documentación de los contratos con javadoc.
Como definir el contrato de un método.
Caso de estudio. Comprensión de requerimientos, comprensión del mundo del problema, paquetes, declaración de clases, asignación de responsabilidades, contrato de un método.
Patrón template

Objetivos de la práctica

Comprender la utilidad del polimorfismo.
Comprender y utilizar el modificador de acceso *protected*
Aplicar enlace tardío
Sobreescribir métodos cambiando la accesibilidad de los mismo en las subclases.
Desarrollar interfaces gráficas sencillas
Comprender el uso de componentes gráficos y utilizar sus eventos
Comprender la necesidad de definir contratos, precondiciones y postcondiciones.
Aprender a documentar utilizando JavaDoc
Asignar responsilidades a las clases y los métodos y documentarlo en Javadoc.

Ejercicio 1:

Dadas las siguientes clases indica la salida por pantalla que tendrá el método main:

```
public class Clase1
{     public int atributo=1;

      public int metodo1()
      {     return 10; }

      public void metodo2()
      {System.out.println
      (“Clase1.metodo2”);}

      public void metodo3()
      {System.out.println
       (“Clase1.metodo3”); }

      public void metodo4()
      {System.out.println
      (“Clase1.metodo4”);}

      public int getAtributo()
      {return atributo;}
}
```

```
public class Clase2 extends Clase1
{
      public int atributo=2;

      public int metodo1()
      {return 11;}

      public void metodo2()
      {System.out.println
      (“Clase2.metodo2”);}

      public void metodo4()
      {System.out.println
       (“Clase2.metodo4”);  }

      public int getAtributo()
      {     return atributo;}
}
```

```
public class Clase3 extends
Clase2
{
      public int atributo=3;
      public int metodo1()
      {     return
super.metodo1();
      }
      public void metodo2()
      {     super.metodo2();
            System.out.println
            (“Clase3.metodo2”);
      }
      public void metodo3()
      {     super.metodo3();
            System.out.println
            (“Clase2.metodo3”);
      }
      public int getAtributo()
      {     return atributo;
      }
}
```

```
Public class Main
{
public static void main(String[] args)
{
      Clase3 c3= new Clase3();
      Clase1 c1=c3;
      Clase2 c2=c3;
System.out.println(c1.metodo1());
      c1.metodo2();
      c1.metodo3();
      c1.metodo4();
System.out.println(c2.metodo1());
      c2.metodo2();
      c2.metodo3();
      c2.metodo4();
System.out.println(c3.metodo1());
      c3.metodo2();
      c3.metodo3();
      c3.metodo4();
      System.out.println(c1.atributo);
System.out.println(c2.atributo);
System.out.println(c3.atributo);
System.out.println(c1.getAtributo());
System.out.println(c2.getAtributo());
System.out.println(c3.getAtributo());
}
}
```

Ejercicio 2:

Sean las clases A, B y C. Analizar si son correctas y qué muestra la clase PruebaABC. Justifique.

```
package ejerteorico;

public class A
{
 public int var1=10;
 public int var2=20;

 public void m1()
{System.out.println ("m1 de
A");}

 public void m2()
{System.out.println ("m2 de
A");}
}
```

```
package ejerteorico;

public class B extends A
{
 public int varB1=100;
 public int var2=200;

@Override
 public void m1()
{ System.out.println("m1 de B");}

}
```

```
package ejerteorico;

public class C extends B
{
 public int varC1=500;
 public int var1=700;

 public void m3()
{System.out.println("m3 de C");
}


}
```

```
public class PruebaABC
{
 public static void main(String[] args)
 {
 C datoc = new C();
 datoc.m3();
 datoc.m2();
 datoc.m1();
 System.out.println("El valor de los
atributos es: \n "+ " varC1="+datoc.varC1+ "
var1="+ datoc.var1 + " var2="+ datoc.var2 + "
varB1=" + datoc.varB1);
}
}
```

b) Si se agrega en el Main el siguiente código, se produciría algún error? Qué resultados arrojaría?. Justifique.

```
A datoX = new C();
datoX.m3();
datoX.m2();
datoc.m1();
System.out.println("El valor de los atributos es: \n "+ " varC1="+
datoX.varC1 +
 " var1="+ datoX.var1 + " var2="+ datoX.var2 +" varB1=" + datoX.varB1);
```

Ejercicio 3:

Dadas las siguientes clases:

```
class Vehículo
{
 public int a=10;
 public void detalle()
{ System.out.print("Vehiculo");}

public int getA(){return a;}
}
```

```
class Coche extends Vehiculo
{
 public int a=20;
 public void detalle()
{System.out.print("Coche") ;}
}
```

```
class Bicicleta extends Vehículo
{public int a= 30;
 public void detalle()
 {
 System.out.print("Bicicleta");
 }
@Override
 public int getA(){return a;}
}
```

¿cuál será su salida del método main? (Identificar la lineas que arrojarían error y justificar. Continuar luego con la ejecución del código ignorándolas)

```
public class Main
{
 public static void main
(String[] args)
{
Vehículo v = new Coche();
Coche c = (Coche) v;
Bicicleta b = (Bicicleta) v;
Object o = v;
Vehiculo v2= (Vehiculo) o
Vehiculo b2 = new Bicicleta();
Bicicleta b3 = b2;

v.detalle () ;
c.detalle ();
b.detalle () ;
o.detalle () ;
v2.detalle ();
b2.detalle ();
b3.detalle ();
```

```
System.out.println(v.a);
System.out.println(c.a);
System.out.println(b.a);
System.out.println(o.a);
System.out.println(v2.a);
System.out.println(b2.a);
System.out.println(b3.a);

System.out.println(v.getA());
System.out.println(c.getA());
System.out.println(b.getA());
System.out.println(o.getA());
System.out.println(v2.getA());
System.out.println(b2.getA());
System.out.println(b3.getA());

 }
}
```

Ejercicio 4:

Se desea desarrollar una aplicación que permita calcular los precios de alquiler de una empresa de alquiler de vehículos.

Cada vehículo se identifica por medio de su patente.

La empresa alquila distintos tipos de vehículos, tanto para transporte de personas como de carga. En la actualidad los vehículos alquilados por la empresa: autos, combis, camionetas de carga y camiones.

El precio del alquiler de cualquier vehículo tiene una componente base de alquiler a razón de $500 por día (este precio base podría cambiar).

En el caso de alquiler de un auto, al precio base diario se lo incrementa un 1.5 % por plaza.

El precio de alquiler de las combis es igual al de los coches, salvo que se le añade un 2% por plaza independiente de la cantidad de días de alquiler.

El precio de los vehículos de carga es el precio base diario incrementado un 20% * PMA (PMA = peso máximo autorizado en toneladas), de modo que si el el vehículo tiene una capacidad de carga 1.5 toneladas, al precio base diario se lo incrementará un 30%.

En el caso de los camiones, al precio se le suma un fijo del 40% independientemente de la cantidad de días de alquiler.

Inciso a)
Escribe las clases correspondientes, documentándolas en JavaDoc, indicando en cada método las precondiciones y postcondiciones.

Inciso b)
Escribe la clase Prueba que contenga un método main, en la cual se instanciará un ArrayList de 10 vehículos de diferentes tipos (combis, autos, etc.) y luego muestra por pantalla los precios de alquiler de los mismos.

Ejercicio 5

Tomaremos como base el ejercicio de la empresa de colectivos de la guía 2, que nos presentaba las clases Chofer, Colectivo y Categoría, y lo ampliaremos:

Realiza una aplicación Java que represente una empresa de transporte, en la que conservamos información de los choferes y los vehículos disponibles.

La empresa maneja 3 tipos de vehículos, colectivos de linea, colectivos de larga de distancia, y camiones, estos últimos, a su vez, pueden tener un acoplado.

La clase Categoría, tiene los atributos:

```
String nombreCategoría
double sueldo
boolean habilitaColectivoLinea
boolean habilitaColectivoLarga
boolean habilitaCamion
```

Los choferes de categoría 1, solo pueden conducir colectivos de linea, los choferes de categoría 2 pueden además conducir colectivos de larga distancia. Los de categoría 3 solo pueden conducir camiones, y los choferes de categoría 4 pueden conducir todos los vehículos.

Los sueldos serán los siguientes:

Categoría 1: $10.000

Categoría 2: $15.000

Categoría 3: $15.000

Categoría 4: $17.000

La clase Chofer tiene los atributos nombre, categoria, vehiculoAsignado .

La clase Vehiculo, tendrá el método abstracto `boolean aceptoChofer(Chofer)` que devolverá true o false en caso de que ese vehículo admita o no ser conducido por un determinado chofer.

Los vehículos tendrán los atributos modelo (por ejemplo Mercedes Benz 1114) y número de interno.

Los colectivos tendrán el atributo cantidadPasajeros, los colectivos de larga distancia tendrán además el atributo booleano cocheCama.

Los camiones tendrán el atributo Tara y carga máxima, y acoplado.

El atributo acoplado sera de tipo Acoplado, y representará el acoplado que el camión tenga asignado, en caso de no tener un acoplado enganchado, este atributo tomará el valor null.

Los únicos atributos de acoplado serán:

```
int tara
int cargaMaxima
boolean refrigerado
int numeroAcoplado
int boolean enUso
```

Tener en cuenta que un acoplado no es un vehículo.

Requerimientos:

1. Cada chofer posee un único domicilio, y tiene una categoría que define su sueldo y su capacidad de conducir un determinado tipo de vehículo.
2. El número de interno de un colectivo deberá generarse automáticamente en forma correlativa.
3. Varios choferes pueden tener la misma categoría.
4. A cada chofer se le puede asignar un vehículo, consideremos que no cualquier chofer puede conducir cualquier vehículo.
5. Se deberá poder desvincular un chofer de su vehículo.
6. Un chofer podría no tener un vehículo asignado.
7. A cada camión se le podrá asignar un acoplado.
8. Se deberá poder desvincular un acoplado del camión.
9. Un camión podría no tener un acoplado enganchado.
10. Deberá ser posible mostrar la información de cada chofer, incluyendo la información detallada correspondiente al vehículo asignado, o indicar que el chofer no tiene un vehículo asignado (Considerar la posibilidad de sobreescribir los métodos toString).
11. En caso de que el vehículo sea un camión, al pedir información detallada del mismo se deberá informar si tiene o no un acoplado, y en caso de tenerlo, dar la información del mismo.

Agregar una clase EMPRESA que contenga a las demás.

El sistema debe poder responder a lo siguiente:

1. ¿Cuántos choferes hay de una determinada categoría?
2. ¿Cuántos choferes no tienen un vehículo asignado?
3. ¿Cuántos vehículos posee la empresa en total?

4. ¿Cuántos acoplados posee la empresa?
5. Se deberá poder visualizar información detallada de los vehículos (con su acoplado si lo tuviese), los choferes y los acoplados

Agregue los atributos y métodos que crea convenientes.

Documenta todas las clases y métodos utilizando JavaDoc, indicando en cada método las precondiciones y postcondiciones.

Desarolla una aplicación con un interfáz gráfica sencilla que nos permita agregar choferes, agregar lo diferentes tipos de vehículos (en listas separadas) y agregar acoplados a una empresa.

A través de la interfaz gráfica se debe poder:

- Vincular y desvincular choferes con un vehículo.
- Enganchar y desenganchar un acoplado a un camión.
- Visualizar la información detallada de los vehículos, choferes, y acoplados.

Ejercicio 6 (Patrón template)

Considera el ejercicio de la guia 3 de las cuentas bancarias que tienen un saldo y un titular. y que pueden realizar operaciones de extracciones y depósitos.

Para poder crear una cuenta bancaria debe conocerse el nombre del titular, el cual no podrá modificarse, el saldo inicial de toda cuenta bancaria es igual a cero.
Existen tres tipos de cuentas bancarias: Cuentas corrientes, Cajas de ahorro y Cuentas Universitarias.
Si revisamos el comportamiento nos encontraremos con las siguientes características en común:

- Todas llevan cuenta de su saldo.
- Todas permiten realizar depósitos.
- Todas permiten realizar extracciones.

Pero cada una tiene un tipo de restricción distinto en cuanto a las extracciones:

- Cuentas corrientes: permiten que el cliente gire en descubierto (con un tope pactado con cada cliente). Dicho tope podrá modificarse, pero debe indicarse al momento de crear la cuenta corriente. (Ver constructores encadenados)
- Cajas de ahorro: poseen una cantidad máxima de extracciones mensuales (para todos los clientes). No se permite girar en descubierto.
- Cuenta universitarias: permite extracciones de no más de $1.000.- diarios.

Aplica el patrón Template. El metodo template es extraer(unMonto) y el metodo gancho es validaExtraccion(unMonto)

Ejercicio 7 (Patrón template con hook)

Se desea modelar el proceso de preparación de una infusión.
Para preparar y tomar una infusión deben seguirse la siguiente secuencia de pasos independientemente del tipo de infusión:

```
calentarAgua();
agregarTipoInfusion();
endulzar();
```

A los fines prácticos del presente ejercicio, los métodos expuestos solo mostrarán carteles en pantalla.
Modelar dos tipos de infusiones: Mate y Café, el Café tendrá un atributo que indica si será dulce o no, (el mate SIEMPRE se toma amargo).

- Utilizar el patrón Template para modelar la situación (el método endulzar funcionará como hook).

Las infusiones mostrarán la siguiente secuencia:

Mate	Cafe Dulce	Cafe Amargo
`Calentando el agua`	`Calentando el agua`	`Calentando el agua`
`Se agrega Yerba al mate`	`Se agrega Cafe Molido a la taza`	`Se agrega Cafe Molido a la taza`
`La bebida se tomara amarga`	`Se agrega azucar a la bebida`	`La bebida se tomara amarga`
`Tomando bebida`	`Tomando bebida`	`Tomando bebida`

Ejercicio 8

Se desea escribir un juego de estrategia en el cual habrá diferentes personajes que se atacarán entre ellos, (caballeros, arqueros, paladines, etc.).

Todos los personajes tendrán los atributos, String nombre, int vitalidad, y un atributo de tipo Posicion que nos provee los métodos:

- double getPosx(),
- double getPosy(),
- void setXY(double , double),
- void incrementaPos(double valorX, double valorY)
- double distancia(Posicion) (La distancia calculada es la distancia pitagórica entre dos puntos)

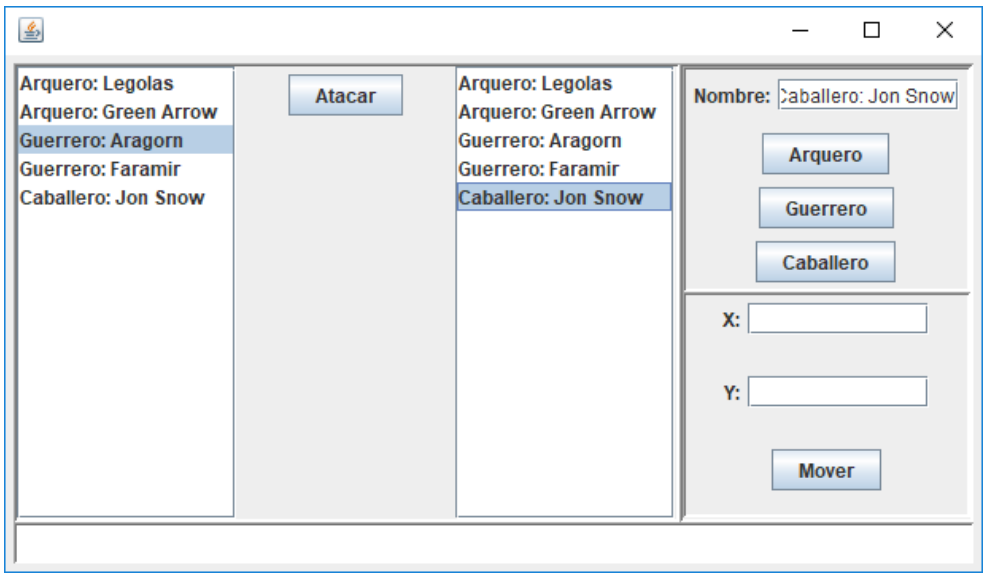
Todos los personajes pueden atacar a otro personaje y por consiguiente pueden recibir un ataque, por lo que deberán tener los métodos:

- boolean ataca(Personaje) que devuelve true o false en caso de poder o no realizar el ataque.
- void recibeDanio(int cantidad)

Existen diferentes tipos de Personajes

Caballero, Guerrero, Arquero.

- El Caballero tendrá una vitalidad inicial de 500 unidades, podrá atacar a los enemigos que estén a una distancia menor a 10 unidades provocando un daño de 10 unidades.
- El Guerrero tendrá una vitalidad inicial de 800 unidades y un atributo entero para indicar su nivel de armadura, podrá atacar a los enemigos que estén a una distancia menor a 5 unidades provocando un daño de 10 unidades. En caso de recibir daño, se reducirá su armadura y al llegar a cero comenzará a disminuir su vitalidad.
- El Arquero tendrá una vitalidad inicial de 500 unidades, y un atributo para indicar la cantidad de flechas que posee (inicialmente 20). Si tiene flechas podrá atacar a personajes a una distancia menor a 100 unidades provocando un daño de 15 unidades, esto gastará una flecha. En caso de quedarse sin flechas solo podrá atacar a enemigos que estén a una distancia menor a 5 unidades, provocando un daño de 5 unidades.
- Escribe las clases antes mencionadas, agregando los métodos y atributos que creas convenientes.
- Crea un JFrame sencillo con con botones, listas y cuadros de texto para probar las funcionalidades de los personajes. El Jframe pordría tener un aspecto similar al de la figura:



Ejercicio 9

Considera el siguiente código:

```
public class Base
{
 String metodo1(Base x)
 {
  return "metodo 1 en clase Base,
parametro Base: "+x.toString();
 }

 String metodo1(Extendida x)
 {
  return "metodo 1 en clase Base,
parametro Extendida: "
+x.toString();
 }

 String metodo2(Extendida y)
 {
  return "metodo 2 en clase Base,
parametro Extendida: "
+y.toString();
 }

@Override
 public String toString()
 {
  return "soy Base";
 }
}
```

```
public class Extendida extends Base
{

 String metodo1(Extendida x)
 {
  return "metodo 1 en clase
Extendida, parametro Extendida: "
+x.toString();
 }

 String metodo2(Base y)
 {
  return "metodo 2 en clase
Extendida, parametro Base: "
+y.toString();
 }

@Override
 public String toString()
 {
  return "soy Extendido";
 }

}
```

```
public class Main
{
    public static void main(String[] args)
    {
        Base b = new Base();
        Base e = new Extendida();
        System.out.println(b.metodo1(e));
        System.out.println(e.metodo1(e));
        System.out.println(b.metodo1((Extendida) e));
        System.out.println(e.metodo1((Extendida) e));
        System.out.println(b.metodo2(b));
        System.out.println(e.metodo2( e));
        System.out.println(e.metodo2((Extendida) b));
        System.out.println(e.metodo2((Extendida) e));
        Extendida x = (Extendida) e;
        System.out.println(x.metodo2(b));
        System.out.println(x.metodo2(e));
    }
}
```

Analiza los resultados de la ejecución del método main. En caso de que se provoque un error de compilación o de ejecución, (analiza cual sería) omitir la linea en cuestión y continuar con la ejecución. Verifica luego los resultados obtenidos en el compilador.