LECTURA 9 HERENCIA INTERFASE Y METACLASE

PROGRAMACION II MARKETING DIGITAL

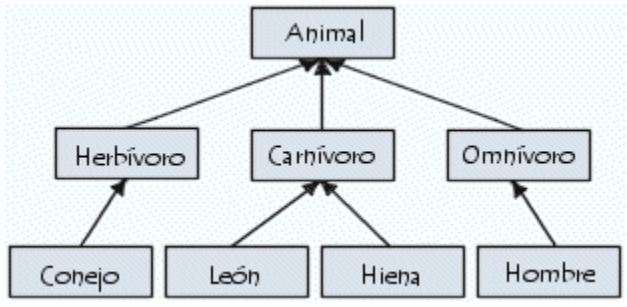
El concepto de herencia

La **herencia** es específica de la programación orientada a objetos, donde una clase nueva se crea a partir de una clase existente. La herencia (a la que habitualmente se denomina **subclase**) proviene del hecho de que la subclase (la nueva clase creada) contiene las atributos y métodos de la clase primaria. La principal ventaja de la herencia es la capacidad para definir atributos y métodos nuevos para la subclase, que luego se aplican a los atributos y métodos heredados.

Esta particularidad permite crear una estructura jerárquica de clases cada vez más especializada. La gran ventaja es que uno ya no debe comenzar desde cero cuando desea especializar una clase existente. Como resultado, se pueden adquirir bibliotecas de clases que ofrecen una base que puede especializarse a voluntad (la compañía que vende estas clases tiende a proteger las datos miembro usando la encapsulación).

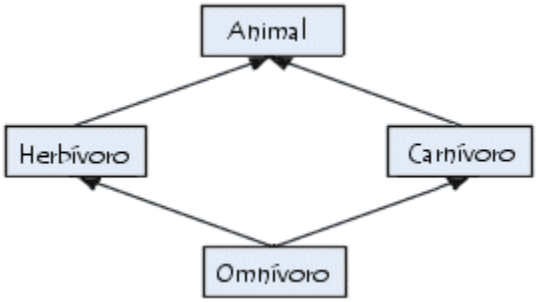
Jerarquía de clase

La relación **padre-hijo** entre clases puede representarse desde un punto de vista jerárquico, denominado **vista de clases en árbol**. La vista en árbol comienza con una clase general llamada superclase (a la que algunas veces se hace referencia como clase primaria, clase padre, clase principal, o clase madre; existen muchas metáforas genealógicas). Las clases derivadas (clase secundaria o subclase) se vuelven cada vez más especializadas a medida que van descendiendo en el árbol. Por lo tanto, se suele hacer referencia a la relación que vincula una clase secundaria con una clase primaria mediante la frase **es una** x o y.



Herencia múltiple

Algunos lenguajes orientados a objetos, como C++ permiten herencias múltiples, lo que significa que una clase puede heredar los atributos de otras dos superclases. Este método puede utilizarse para agrupar atributos y métodos desde varias clases dentro de una sola.



L2

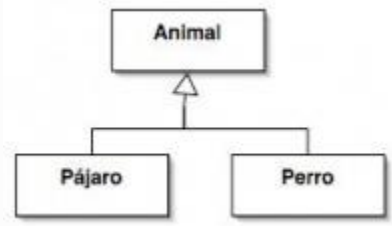
Herencia. Es una propiedad que permite que los objetos sean creados a partir de otros ya existentes, obteniendo características (métodos y atributos) similares a los ya existentes.

Sumario

Qué es Herencia

Es la relación entre una clase general y otra clase mas especifica. Es un mecanismo que nos permite crear clases derivadas a partir de clase base, nos permite compartir automáticamente métodos y datos entre clases subclases y objetos. Por ejemplo: Si declaramos una clase párrafo derivada de un clase texto todos los métodos y variables asociadas con la clase texto son automáticamente heredados por la subclase párrafo. La herencia, junto con la encapsulación y el polimorfismo, es una de las tres características principales (o "pilares") de la programación orientada a objetos. La herencia permite crear nuevas clases que reutilizan, extienden y modifican el comportamiento que se define en otras clases. La clase cuyos miembros se heredan se denomina clase base y la clase que hereda esos miembros se denomina clase derivada.

Tipos de Herencia

- 

  Herencia Simple

  Herencia Simple: Indica que se pueden definir nuevas clases solamente a partir de una clase inicial

- 

  Herencia Multiple

  Herencia Multiple: Indica que se pueden definir nuevas clases a partir de dos o más clases iniciales.

- **Herencia de implementación**: La implementación de los métodos es heredada. Puede sobreescribirse en las clases derivadas.
- **Herencia de interfaz**: Sólo se hereda la interfaz, no hay implementación a nivel de clase base (interfaces en Java, clases abstractas en C++)

Ejemplo

En el ejemplo siguiente se muestra cómo se expresan en C# las relaciones de clase presentadas en la ilustración anterior. En el ejemplo también se muestra cómo WorkItem invalida el método virtual Object.ToString y cómo la clase ChangeRequest hereda la implementación de WorkItem del método. // WorkItem implicitly inherits from Object class public class WorkItem {

```
    private static int nextID;
    protected int ID { get; set; }
    protected TimeSpan jobLength { get; set; }
    protected string Title { get; set; }
    protected string Description { get; set; }
    // Default constructor
    public WorkItem()
    {
        ID = 0;
        Title = "Default title";
        Description = "Default description.";
        jobLength = new TimeSpan();
    }
    // Static constructor for static member.
    static WorkItem()
    {
        nextID = 0;
    }
    // Instance constructor.
    public WorkItem( string title, string desc, TimeSpan joblen)
    {
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = joblen;
    }
    protected int GetNextID()
    {
       return ++nextID;
    }
    public void Update(string title, TimeSpan joblen)
    {
        this.Title = title;
        this.jobLength = joblen;
    }

    // Virtual method override.
    public override string ToString()
    {
        return String.Format("{0} - {1}", this.ID, this.Title);
    }
```

}

// ChangeRequest derives from WorkItem and adds two of its own members. public class ChangeRequest : WorkItem {

```
   protected int originalItemID {get; set;}
   public ChangeRequest() { }
   public ChangeRequest(string title, string desc, TimeSpan jobLen,
int originalID)
   {
       this.ID = GetNextID();
       this.Title = title;
       this.Description = desc;
       this.jobLength = jobLen;
       this.originalItemID = originalID;
   }
```

}

class Program {

```
   static void Main()
   {
       WorkItem item = new WorkItem(
                       "Fix Bugs",
                       "Fix all bugs in my source code branch",
                       new TimeSpan(3, 4, 0, 0));

       ChangeRequest change = new ChangeRequest("Change design of
base class",
                                                "Add members to base
class",
                                                new TimeSpan(4, 0,
0),
                                                1);

       Console.WriteLine(item.ToString());

       // ChangeRequest inherits WorkItem's override of ToString
       Console.WriteLine(change.ToString());

       // Keep the console open in debug mode.
       Console.WriteLine("Press any key to exit.");
       Console.ReadKey();
   }
```

}

Interfaces

Buenas amigos, en este nuevo tema vamos a comentar las **"Interfaces"**. Una interfaz es un conjunto de métodos abstractos y de constantes cuya funcionalidad es la de determinar el funcionamiento de una clase, es decir, funciona como un molde o como una plantilla. Al ser sus métodos abstractos estos no tiene funcionalidad alguna, sólo se definen su tipo, argumento y tipo de retorno.

La construcción de una Interfaz es la siguiente:

```
public Interfaz NombreInterfaz{
        //Código
}
```

Para declarar una o más Interfaces ponemos, a la derecha del nombre de la clase, la palabra clave **"implements"** seguido de los nombres de las interfaces separadas por comas.:

```
class NombreClase implements Interfaz1,Interfaz2,Interfaz3{
        //Código
}
```

También es posible heredar e implementar al mismo tiempo:

```
class NombreClase extends Herencia implements Interfaz1,Interfaz2,Interfaz3{
        //Código
}
```

Veamos algunas características de las Interfaces:

- Las Interfaces solo pueden tener visibilidad de **package** o **public**.
- Todos los métodos declarados en una Interfaz son **public** y **abstract**, si no se le indica, Java lo pondrá implícitamente.
- Los métodos de una Interfaz no pueden ser estáticos, ya que estos deben ser redefinidos, y al ser estáticos les sería imposible.
- Todos los atributos declarados en una Interfaz son **"public static final"** y deberán tener asignado un valor constante. Todos los nombres de constantes van en **"MAYÚSCULAS"**
- Una clase puede implementar una o más interfaces.
- Una Interfaz puede heredar de una o varias interfaces, pero no pueden implementar **NADA**. Como sabemos, las clases solo pueden heredar de otra clase, pero las Interfaces pueden hacer herencia múltiples.

- Todas las clases que implementen una interfaz deben de redefinir todos los métodos de esa interfaz.
- Las interfaces no tienen constructor, por lo que no es posible crear objetos con el operador **"new"** de ésta.
- Si se implementa en una clase una Interfaz, y esta Interfaz hereda de otra, la clase deberá implementar todos los métodos de la interfaz que implementa y de los métodos que esta hereda.
- Para la herencia de las interfaces se pone la palabra clave **extends** y se ponen todas las interfaces que se deseen separadas por comas.
- Una Interfaz puede ocultar constantes y métodos heredados de otras Interfaces si los vuelve a declarar en la clase que hereda.

Veamos un ejemplo:

```
public interface Operaciones {

    public static final double NUMERO1=3.6;
    public static double NUMERO2=5.9;

    double suma();
    double resta();
    double multiplicacion();
    double division();

}
```

Esta interfaz contiene:

- 2 Atributos constantes **NUMERO1** y **NUMERO2**. Observaréis que un atributo es declarado como **"final"** y otro no, pues bien, aunque no se lo pongamos, java nos lo incluye implícitamente, con lo cual, aunque o esté puesto, sigue siendo **"final"**
- 4 métodos. Dijimos anteriormente que los métodos de una interfaz deben ser **"public"** y **"abstract"**, yo en este ejemplo no los he puesto, ya que, como en el caso de los atributos, java nos pone los métodos **"public"** y **"abstract"** implícitamente.

```java
public class Principal implements Operaciones {

    public double suma() {
        // TODO Auto-generated method stub
        return Operaciones.NUMERO1+Operaciones.NUMERO2;
    }

    @Override
    public double resta() {
        // TODO Auto-generated method stub
        return Operaciones.NUMERO1-Operaciones.NUMERO2;
    }

    @Override
    public double multiplicacion() {
        // TODO Auto-generated method stub
        return Operaciones.NUMERO1*Operaciones.NUMERO2;
    }

    @Override
    public double division() {
        // TODO Auto-generated method stub
        return Operaciones.NUMERO1/Operaciones.NUMERO2;
    }
    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Principal p=new Principal();
        System.out.println("La suma es: "+p.suma());
        System.out.println("La resta es: "+p.resta());
        System.out.println("La multiplicación es: "+p.multiplicacion());
        System.out.println("La división es: "+p.division());
    }
}
```

- Creamos una clase con método **MAIN** e implementamos la interfaz Operaciones.
- Posteriormente deberemos redefinir sus métodos. Veréis que en el tipo de retorno obtengo los atributos de la interfaz Operaciones y puesto que no podemos crear objetos de la interfaz, ya que no utiliza constructores, hacemos su llamada a los atributos como:
- **Operaciones.NUMERO1**
- **Operaciones.NUMERO2.**
- Esto es debido a que son estáticos e inmutables. Este tipo de llamada también la vimos en la clase Math
- Posteriormente en la **MAIN** creamos un objeto de la clase Principal y llamamos a todos sus métodos y los mostramos por pantalla.

Una cosa que me gustaría dejar más o menos clara, y es que en la **MAIN** creamos un objeto de la clase Principal para llamar a sus métodos y no hacemos la llamada simplemente. Esto es debido a que la **MAIN** es un método estático y como tal solo puede aceptar atributos y métodos estáticos como los métodos que redefinimos de la interfaz no pueden ser estáticos, creo un objeto de la propia clase y los voy llamando. En temas posteriores haré una entrada para explicar mejor la palabra reservada **"static"**.

METACLASE

Definicion Metaclases

En programación orientada a objetos, una metaclase es una clase cuyas instancias son clases. En otras palabras, como los objetos son instancias de una clase, las clases son instancias de una metaclase.

No todos los lenguajes orientados a objetos soportan metaclases. Además, los lenguajes que lo soportan tienen sus propias reglas que definen como los objetos, clases y metaclases interactúan.

Las metaclases permiten saber el tipo exacto de un objeto cuando lo único que se tiene es un manejador a su clase base. suministra un conjunto de metaclases con este objetivo:
Class, Field, Method, Constructor,
etc. Asímismo, la clase
Object
posee el método
getClass(),
que permite obtener un objeto de tipo
Class
con toda la información sobre el verdadero objeto apuntado.* El objeto
.class
correspondiente a una clase cualquiera
Xxx
puede obtenerse sencillamente mediante la expresión
Xxx.class
.* La verdad es que, durante la ejecución, por cada clase que haycargada, Java mantiene un objeto oculto de tipo
Class
quedescribe a dicha clase.* Ese objeto de tipo
Class
se utiliza en tiempo de ejecución parachequear las conversiones de tipo efectuadas por el programador.* El operador
instanceof
nos dice si un objeto es instancia deuna clase determinada o no.