Sesión Práctica 2 . Relaciones en Orientación a Objetos

Metodología y Desarrollo de Programas

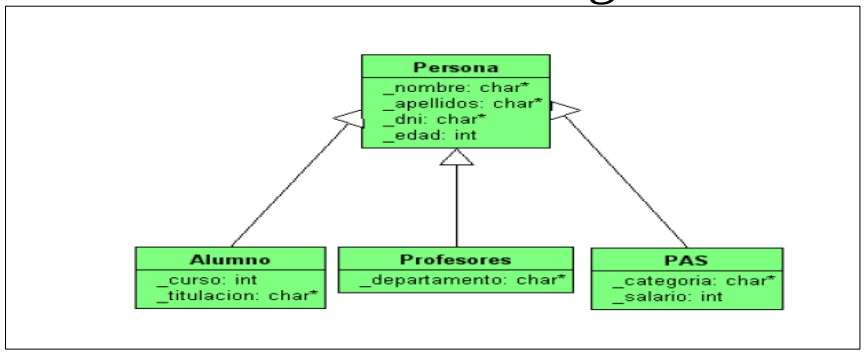
Para la elaboración de esta presentación se ha utilizado algunos de los ejemplos expuestos en:

- Blog de http://arodm.blogspot.com/search/label/Dise%C3%B1o (Blog
- Apuntes de la asignatura de ingeniería del software de Mari Carmen Otero Vidal de la Universidad del País Vasco-Euskal Herriko Unibertsitatea http://www.vc.ehu.es/jiwotvim/ISOFT2010-2011/ingSoftware1011.htm

Índice

3.1. Introducción a las relaciones en OO

- Relaciones
  - Dependencia
    - ✓ se representa mediante una linea discontinua. Utilizada fundamentalmente para la relación entre paquetes. P.ej: Clase Persona tiene un atributo String nombre
  - Herencia
    - ✓ donde un objeto adquiere las propiedades y métodos de un objeto padre. Se representa mediante un triángulo



3.1. Introducción a las relaciones en OO

- Asociación, Composición y Agregación
  - Los atributos de una clase se crean a partir de clases ya existentes que actúan como atributos de la nueva clase
  - Esta nueva característica permite la creación de código a partir de uno ya existente --> **Reutilización**
  - Estas relaciones generalmente se representa con la frase "***tiene un***"
  - La diferencia entre ellas es una cuestión semántica. La implementación a nivel de código son similares

```
class Linea {
    private Punto  _origen;
    private Punto _destino;
    .......
```

¿Es asociación?
¿Es composición?
¿Es agregación?

3.1. Introducción a las relaciones en OO

- Asociación, composición y agregación: relación **TIENE**
  - Asociación: Representa una relación entre clases que colaboran para llevar algo a cabo.

  - Composición
    - ✓Dos objetos están estrictamente limitados por una relación complementaria. Uno no se entiende sin el otro, esto es, cada uno por separado no tiene sentido. El tiempo de vida es dependiente.

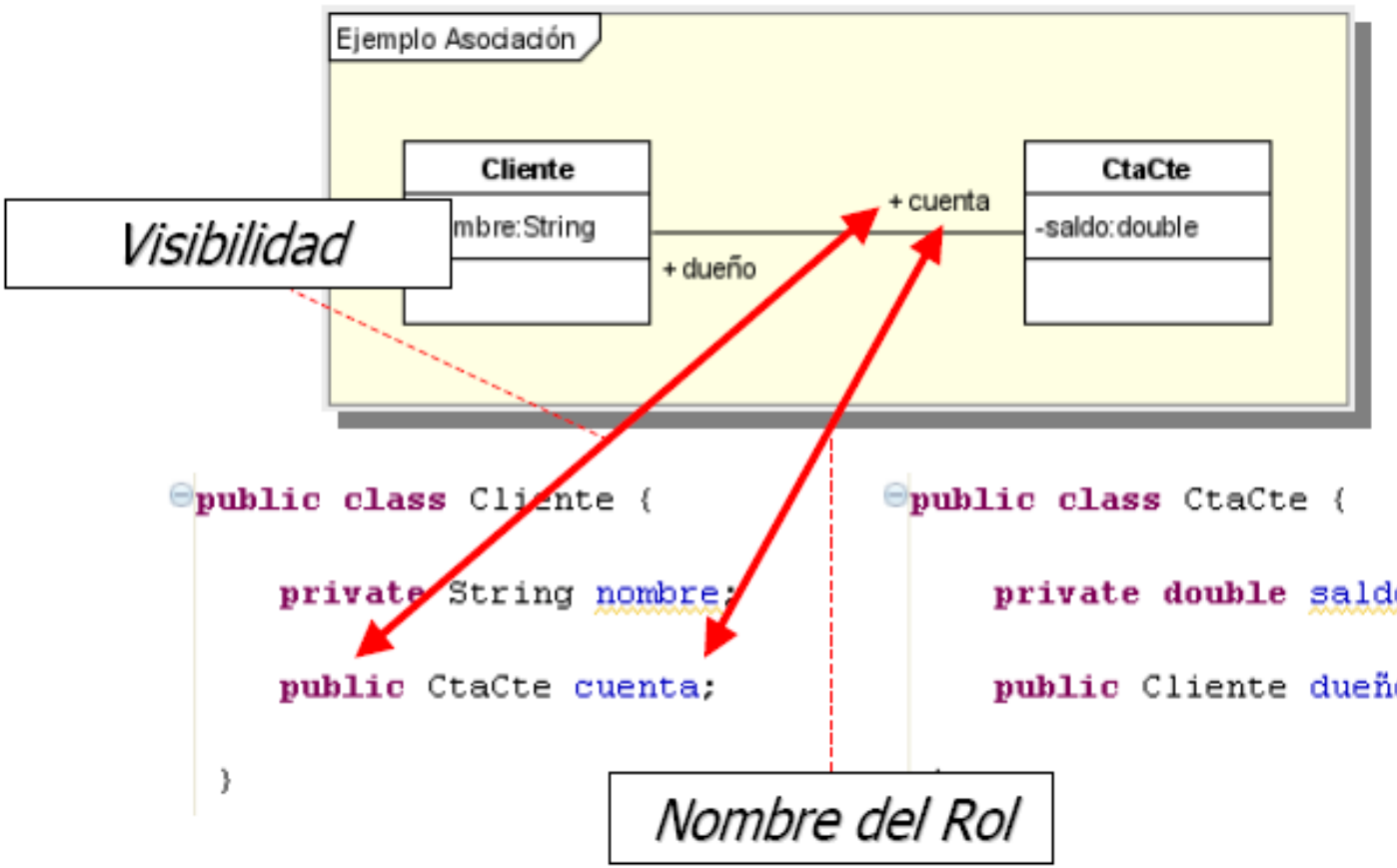
  - Agregación
    - ✓Dos objetos se consideran usualmente como independientes y aun así están ligados. Se puede decir que es de tipo todo / parte. El tiempo de vida de cada uno de los objetos es independiente.

3.2. Asociación, composición y agregación

- Asociación:
  - Se se define como una “relación semántica entre dos o más clases que especifica conexiones entre las instancias de estas clases”.
  - Existen varias opciones de asociación dependiendo de:
    - ✓multiplicidad
      - 0..1 o 1
      - 1..N
    - ✓Direccionales
      - Unidireccionales
      - Bidireccionales

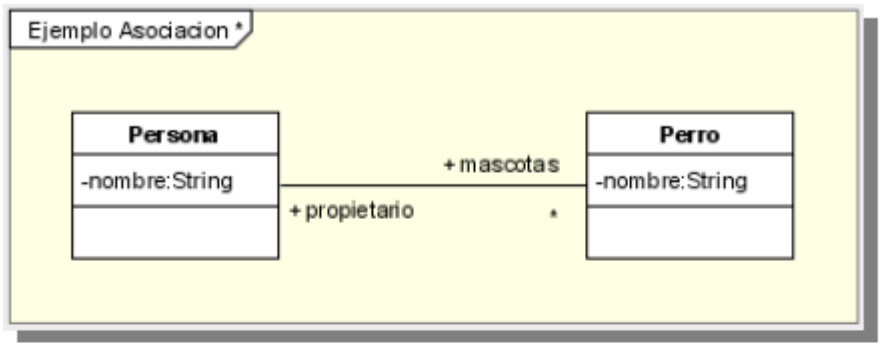
3.2. Asociación, composición y agregación

- Asociación Bidireccional con multiplicidad 0..1 o 1



3.2. Asociación, composición y agregación

- Asociación Bidireccional con multiplicidad *



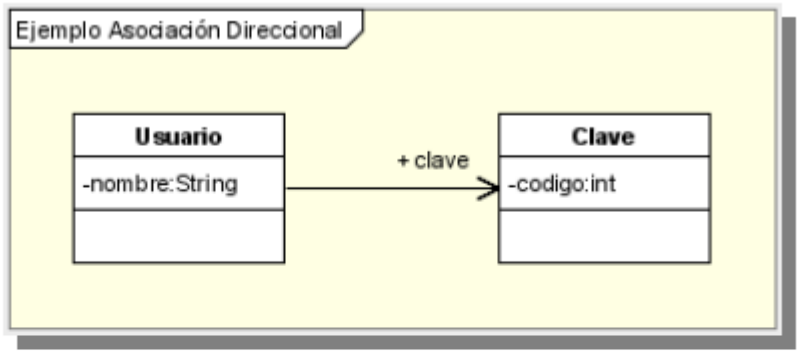

```
public class Persona {

    private String nombre;

    public java.util.Collection mascotas = new java.util.TreeSet();

}
```

```
public class Perro {

    private String nombre;

    public Persona propietario;

}
```

Decisión de Implementación

3.2. Asociación, composición y agregación

- Asociación Direccional con multiplicidad 0..1 o 1

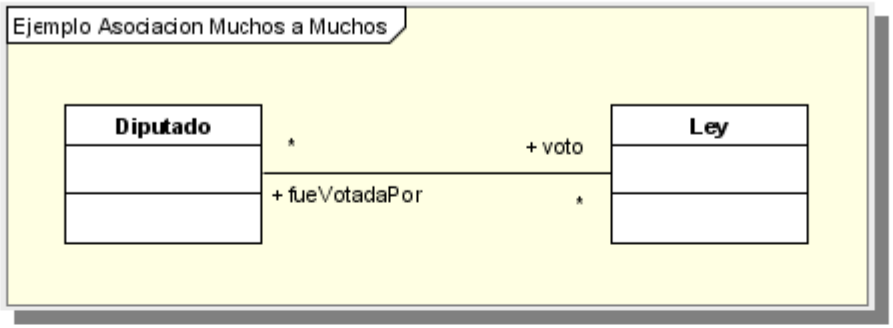



```java
public class Usuario {

    private String nombre;

    public Clave clave;

}
```

```java
public class Clave {

    private int codigo;

}
```

3.2. Asociación, composición y agregación

- Asociación Bidireccional con multiplicidad *

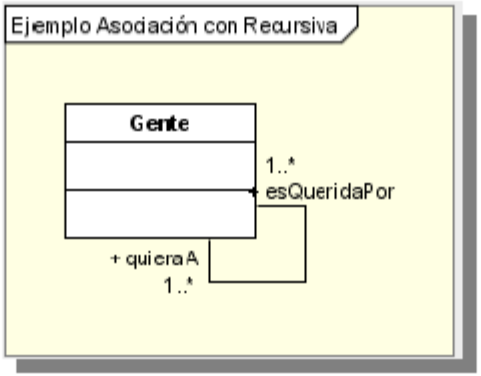



```
public class Diputado {

    public java.util.Collection voto = new java.util.TreeSet();

}
```

```
public class Ley {

    public java.util.Collection fueVotadaPor = new java.util.TreeSet();

}
```

3.2. Asociación, composición y agregación

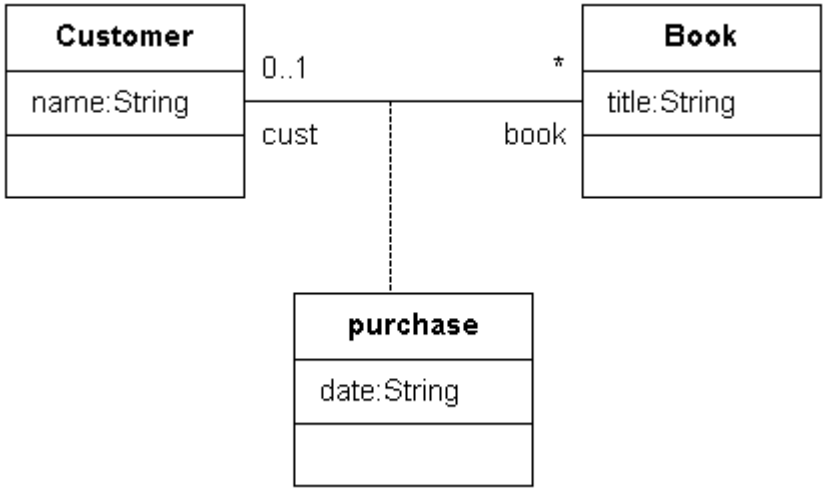
- Asociación




```
public class Gente {

    public java.util.Collection quieraA = new java.util.TreeSet();

    public java.util.Collection esQueridaPor = new java.util.TreeSet();

}
```

3.2. Asociación, composición y agregación

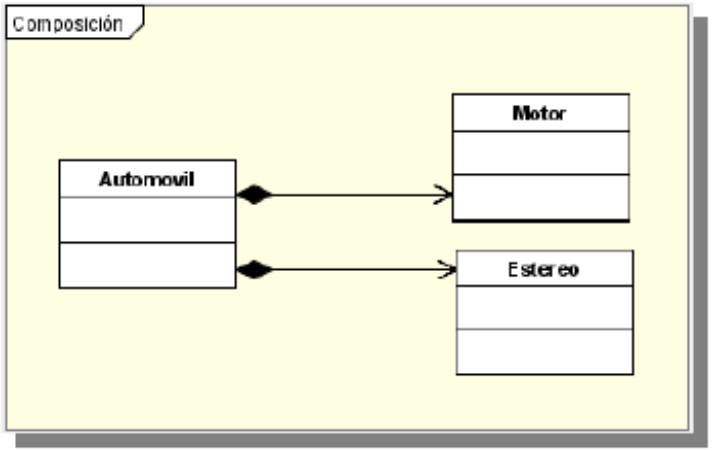
- Asociación ternaria:
  - Se usa clase de asociación para representar la compra (purchase) de un libro (Book) por un comprador (Customer).




```
public class Book {
   // Data attributes
   private String title;
   // Association attributes
   public purchase purchaseOfCust;
  ......
} // class Book

public class Customer {
   // Data attributes
   private String name;
   // Association attributes
   public Collection purchaseOfBookSet;


   .......
} // class Customer

public class purchase {
   // Data attributes
   private String date;
   // Association attributes
   public Customer cust;
   public Book book;
  ....
} // class purchase
```

3.2. Asociación, composición y agregación

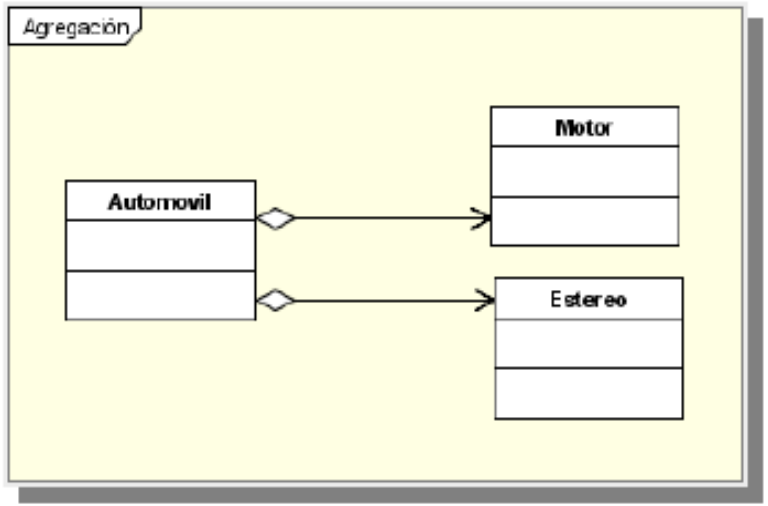
- Composición




```
public class Automovil {

    public Estereo estereo;
    public Motor motor;

    public Automovil() {
        estereo = new Estereo();
        motor = new Motor();
    }

}
```

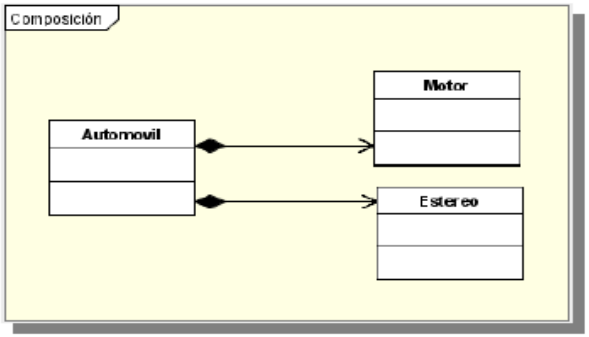
3.2. Asociación, composición y agregación

- Agregación

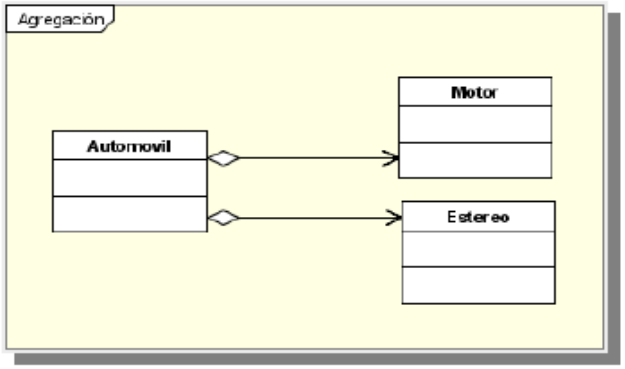



```
public class Automovil {

    public Estereo estereo;
    public Motor motor;

    public Automovil() {
        estereo = new Estereo();
        motor = new Motor();
    }

}
```

3.2. Asociación, composición y agregación




```
public class Automovil {

    public Estereo estereo;
    public Motor motor;

    public Automovil() {
        estereo = new Estereo();
        motor = new Motor();
    }
```



¿Cuál es la diferencia entre composición y agregación?

```
public class Automovil {

    public Estereo estereo;
    public Motor motor;

    public Automovil() {
        estereo = new Estereo();
        motor = new Motor();
    }

}
```

3.2. Asociación, composición y agregación

- Asociación
  - Se se define como una “relación semántica entre dos o más clases que especifica conexiones entre las instancias de estas clases”.
- Composición
  - Dos objetos están estrictamente limitados por una relación complementaria. Uno no se entiende sin el otro, esto es, cada uno por separado no tiene sentido. El tiempo de vida es dependiente.
- Agregación
  - Dos objetos se consideran usualmente como independientes y aun así están ligados. Se puede decir que es de tipo todo / parte. El tiempo de vida de cada uno de los objetos es independiente.

¿Cuál es la diferencia entre asociación composición y agregación a nivel de código?

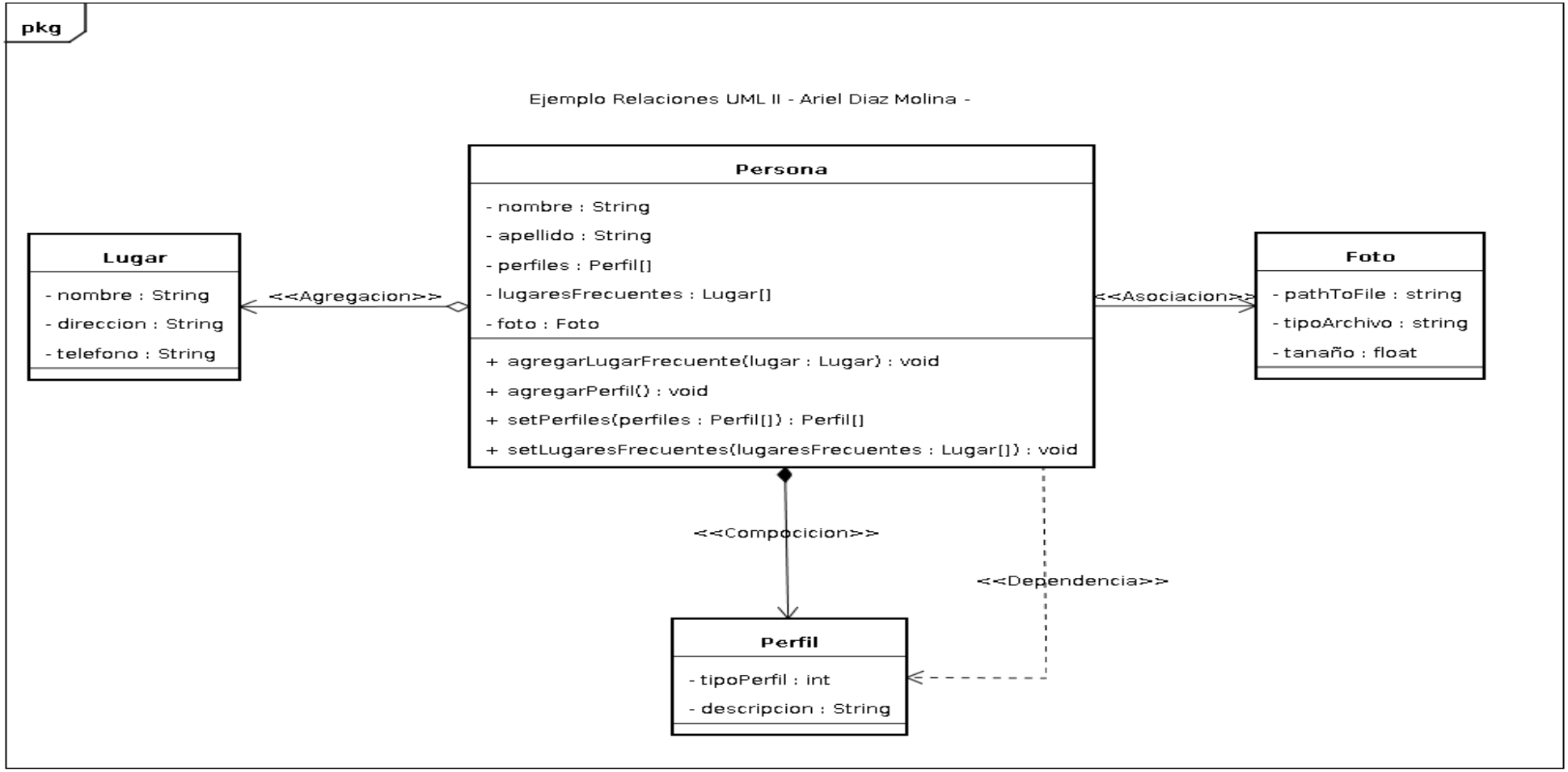
3.2. Asociación, composición y agregación

- Diferencias
  - La relación de **asociación** se suele usar cuando únicamente se tiene **un objeto de otra clase**, que se recibe en el constructor o con el método set.
  - La **agregación** puede, como no, tener los métodos **setter y getter**, mientras que la **Asociación siempre los tiene**, que ponen y obtienen una variable de referencia del mismo tipo de la variable de instancia o de clase.
  - Agregación vs composición: contiene dos métodos uno que **agrega** elementos a la colección y otro que los **elimina** de ella.
    - ✓ **Agregación**: los objetos son pasados por parametro, no han sido instanciados dentro del método, es decir no hemos realizado el new del objeto. **Existen cada uno de forma independiente**.
    - ✓ **Composición**: "el new del objeto se realiza dentro del método agregar", es decir, la instanciación del objeto se realiza dentro del método agregar y la referencia no se devuelve. De este modo, la variable de referencia local va a dejar de existir una vez que el método se termine de ejecutar, y el ciclo de vida de esa variable va a existir mientras el objeto exista.

3.2. Asociación, composición y agregación

- Ejemplos de este apartado obtenido de:

  http://arodm.blogspot.com/search/label/Dise%C3%B1o



3.2. Asociación, composición y agregación

```
public class Persona {

    private String nombre;
    private String apellido;
    private List perfiles = new LinkedList();
    private List lugaresFrecuentes = new LinkedList();

//Setters and Getters
    public String getNombre()
        {return nombre;}
    public void setNombre(String nombre)
        {this.nombre = nombre;}
    public String getApellido()
        {return apellido;}
    public void setApellido(String apellido)
        {this.apellido = apellido;}

    // OJO. Estos son solo setters y getters de las propiedades
    public List getPerfiles() {
        return perfiles;
    }
    public void setPerfiles(List perfiles) {
        this.perfiles = perfiles;
    }
    public List getLugaresFrecuentes() {
        return lugaresFrecuentes;
    }
    public void setLugaresFrecuentes(List lugaresFrecuentes) {
        this.lugaresFrecuentes = lugaresFrecuentes;
    }
```

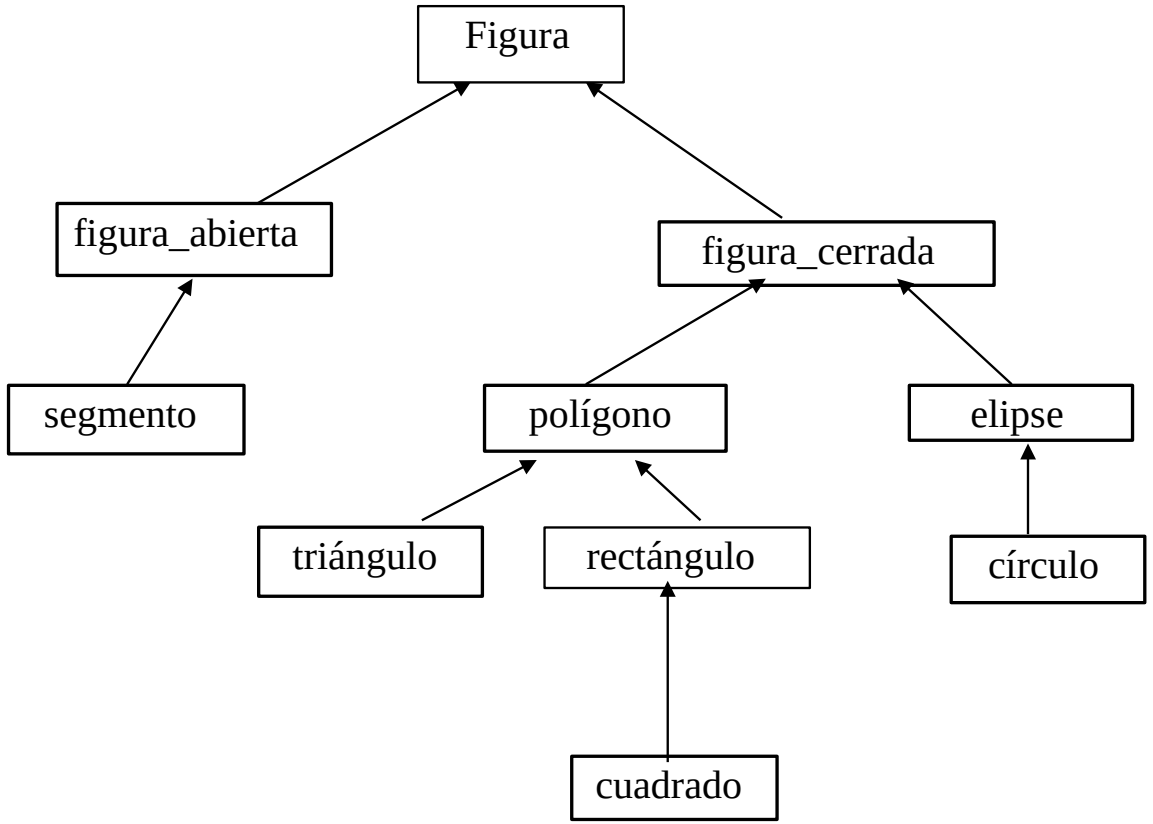
3.2. Asociación, composición y agregación

```
    //Agregación: Añdir y eliminar
    public void agregarLugarFrecuenta(Lugar lugar) {
        if (!lugaresFrecuentes.contains(lugar)) {
            lugaresFrecuentes.add(lugar);
        }
    }

    public void removerLugarFrecuenta(Lugar lugar) {
        if (lugaresFrecuentes.contains(lugar)) {
            lugaresFrecuentes.remove(lugar);
        }
    }

    //Composición: Añadir y eliminar
    public void agregarPerfil() {
        Perfil p = new Perfil();
        perfiles.add(p);
    }
    //sobrecarga
    public void agregarPerfil(String nombre) {
        Perfil p = new Perfil(nombre);
        perfiles.add(p);
    }

    public void removerPerfil(int index) {
        perfiles.remove(index);
    }

}
```

3.3. Concepto de Herencia

- La herencia permite la creación de nuevas clases a partir de otras ya existentes de forma que permite extender o especializar las características y conductas de una o varias clases denominadas base
- Las clases que heredan de la clase/s base/s se denominan derivadas, estas a su vez pueden ser clases bases para otras. Las subclases heredan los métodos y atributos de la clase Base
- Se crean jerarquías de clases entre las clases bases y las clases derivadas
- También se denomina
  - generalización en UML,
  - derivación en C++ y C# y
  - extensión en Java

3.3. Concepto de Herencia



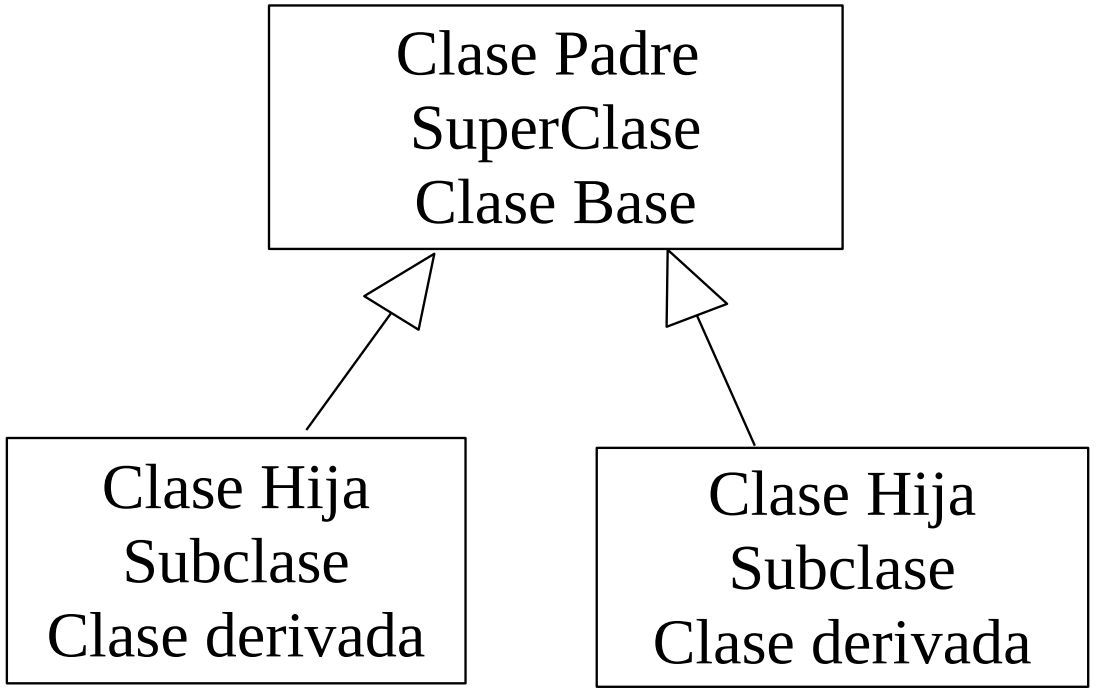
Jerarquía de clases de elementos gráficos

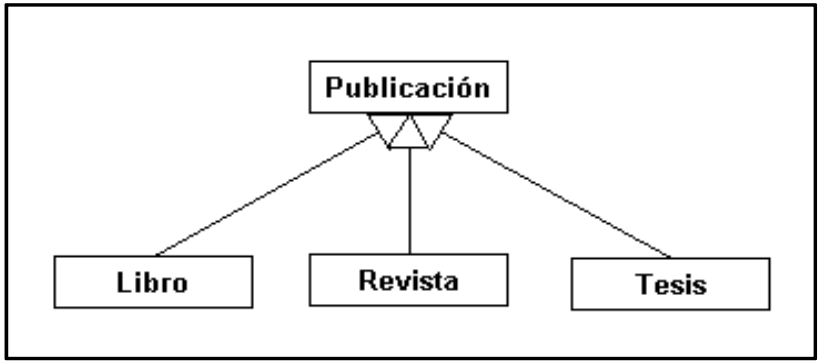
3.3.1. Tipos de herencia

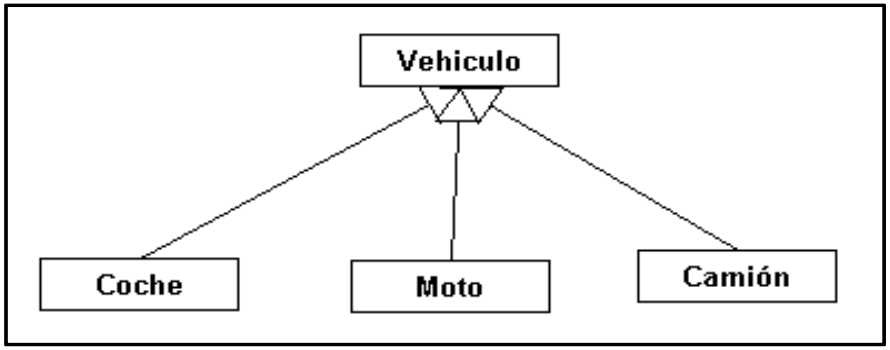
- Distintas clasificaciones:
  - Según el número de clases base: simple o múltiple
  - Según se seleccionen y traten los elementos heredados (en C++): pública, privada, protegida
- “protected” se caracteriza por poder ser accedido desde las funciones miembro de la clase y de sus clases descendientes, pero nunca desde el exterior de la clase o subclases

3.3.1. Tipos de herencia: Simple o múltiple

- *Herencia* simple cuando la subclase o clase derivada sólo hereda de una sola superclase o clase base.

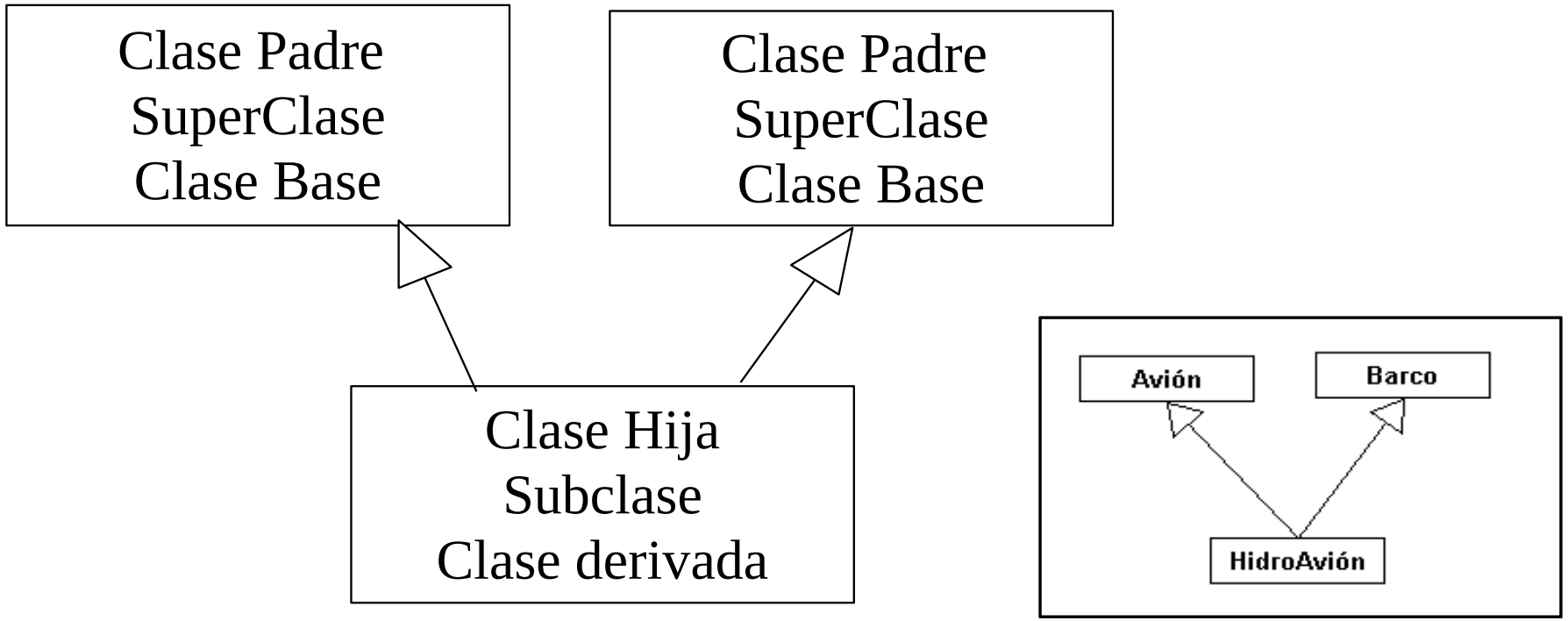






3.3.1. Tipos de herencia: Simple o múltiple

- Herencia múltiple cuando la subclase o clase derivada hereda de varias superclases o clase bases.



3.3.1. Tipos de herencia: Simple o múltiple

- Características de la herencia:
  - La subclase adopta todos los miembros públicos y protegidos de la superclase
  - La subclase puede definir sus propios miembros adicionales, públicos o privados así como redefinir miembros heredados. Lógicamente si define algún método este no podrá ser usado en la clase padre.
  - En la implementación de la subclase aunque se encuentre formado por los atributos privados de la superclase, no se tiene acceso a estos directamente sino mediante las funciones miembros.

- Algunos problemas en Herencia múltiple:
  - Herencia repetida: Una clase hereda dos veces de la misma clase base.
  - Atributos y métodos repetidos -> Ambigüedad en la herencia: Soluciones mediante resolución de ámbito, es decir, indicar expresamente que método se desea ejecutar

Ejemplo de Herencia en Java

- Herencia. **Java**
  - Denominada *extensión*
  - Soporta herencia simple de forma directa
  -

Definición de clase base

```
class clase_base {
// Atributos y métodos
...
};
```

Definición de clase derivada:

```
class clase_derivada: extends clase_base{
    ...
    // Atributos y métodos
    ...
};
```

Tipos de Herencia en Java

Clases dentro del mismo paquete		
Modificador de acceso	**Heredado**	**Accesible**
Por defecto (sin modificador)	Si	Si
private	No	No
protected	Si	Si
public	Si	Si

Clases en distintos paquetes		
Modificador de acceso	**Heredado**	**Accesible**
Por defecto (sin modificador)	No	No
private	No	No
protected	Si	No
public	Si	Si

Ejemplo de Herencia en Java

- Herencia. **Java**

```
public class Persona {
   protected  String nombre;

    protected String dni;

    protected int edad;

    public Persona() {  }
    public Persona(String n, String d, int e) {
              nombre=n;dni=d;edad=e;
    }
    public String getNombre()              { return Nombre;    }
    public void setNombre(String val)      { this.Nombre = val;  }
    public String getDni()                 {return dni;    }
    public void setDni(String val)         { this.dni = val; }
    public int getEdad()                   { return edad; }
    public void setEdad(int val)           { this.edad = val; }
}
```

Ejemplo de Herencia en Java

- Herencia. **Java**

```
public class Alumno extends Persona {
   private int curso=1;
   private String titulacion = "";
   public Alumno() {
                super();
   }
   public Alumno(String n, String d, int e, String tit) {
                super(n,d,e);
                titulacion=tit;
   }
   public int getCurso() {
      return curso;
   }
   public void setCurso(int val) {
      this.curso = val;
   }
   public String getTitulacion() {
      return titulacion;
   }
   public void setTitulacion(String val) {
      this.titulacion = val;
   }
  public String toString(){
      String temp="Alumno: "+super.getNombre()+ " Dni: "+super.getDni()+ " Titulacion:"
                   +this.titulacion;
      return temp;
   }
}
```

Ejemplo de Herencia en Java

- Herencia. **Java**

```
public class EJemplo_Herencia {

    /** Creates a new instance of EJemplo_Herencia */
    public EJemplo_Herencia() {
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        Persona p=new Persona("Luis","44444",30);

        Alumno a=new Alumno("Luis","44444",30,"YY");
        System.out.println(a.toString());
    }

}
```

Características Adicionales de Herencia de Java

- En el lenguaje Java, todas las clases derivan implícitamente de la clase base Object, por lo que heredan las funciones miembro definidas en dicha clase. La clase Object aporta una serie de funciones básicas comunes a todas las clases (deben ser redefinidas):
  - equals. Se utiliza para comparar, en valor, dos objetos.
  - hashCode(): Devuelve un código hash para ese objeto, para poder almacenarlo en una Hashtable.
  - clone(): Devuelve una copia de ese objeto.
  - getClass(): Devuelve el objeto concreto, de tipo Class, que representa la clase de ese objeto.
  - finalize(): Realiza acciones durante la recogida de basura.
  - toString(): El método toString imprime por defecto el nombre de la clase a la que pertenece el objeto y su código (hash)

Características Adicionales de Herencia de Java

- getClass. Clase Class
  - Sirve para identificar el tipo de los objetos en tiempo de ejecución
  - Describe las propiedades de la clase de un objeto
  - String getName(): devuelve el nombre de la clase
  - Ejemplo:

Persona p; …

System.out.println(e.getClass().getName()+ e.getNombre());

- ✓SALIDA: Persona, Luis Arévalo

- instanceof
  - "¿Eres de esta clase o de una clase derivada de ésta?"
  - comparando los objetos Class
  - "¿Eres exactamente de esta clase?"

```
Rectangulo r = new Rectangulo();
(r instanceof Figura) --> true
r.getClass().equals(Figura.class))-->false
```

Características Adicionales de Herencia de Java

- equals
  - Disponible para todo objeto

    public boolean equals(Object obj)
  - Comportamiento por defecto: this==obj
  - Utilizado para implementar la igualdad de objetos.
- toString
  - Devuelve un String con la información del estado actual del objeto
  - Casi todas las clases deberán redefinir este método
  - Son equivalentes:

    System.out.println(punto);

    System.out.println(punto.toString());
  - Al concatenar con el operador “+” automáticamente se invoca al método toString

Sobrecarga vs Redefinición en Java

- Sobrecarga: mismo nombre de método con distinto número y/o tipo de parámetros. (A nivel de clase)

Ej.- métodos constructores

Persona ()

Persona (Persona p)

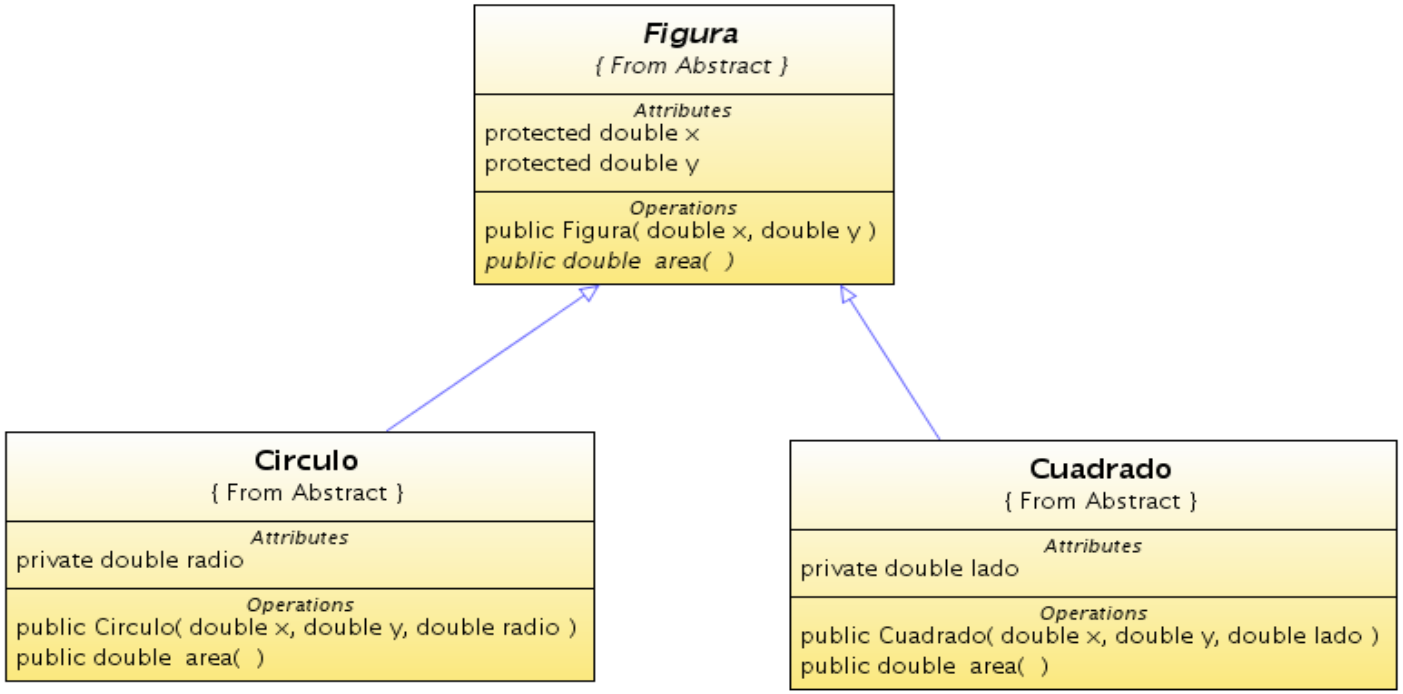
- Redefinición: métodos con el mismo nombre y lista de argumentos, pero con funcionalidad distinta. (A nivel de herencia)
  - Para finalizar la jerarquía de herencia se declara la clase como final.

Persona: void toString()

Alumno: void toString()

3.4 Clase Abstractas

- Una clase abstracta es una clase que no tiene instancias y que se diseña para ser clase base en una jerarquía de herencias donde le aporten un determinado significado.
- Por ejemplo: Una aplicación Gráfica donde se almacenan figuras geométricas. En esta aplicación nunca se instanciará una figura como tal sino Círculos, Cuadrados, Líneas, ...



3.4 Clases Abstractas en Java

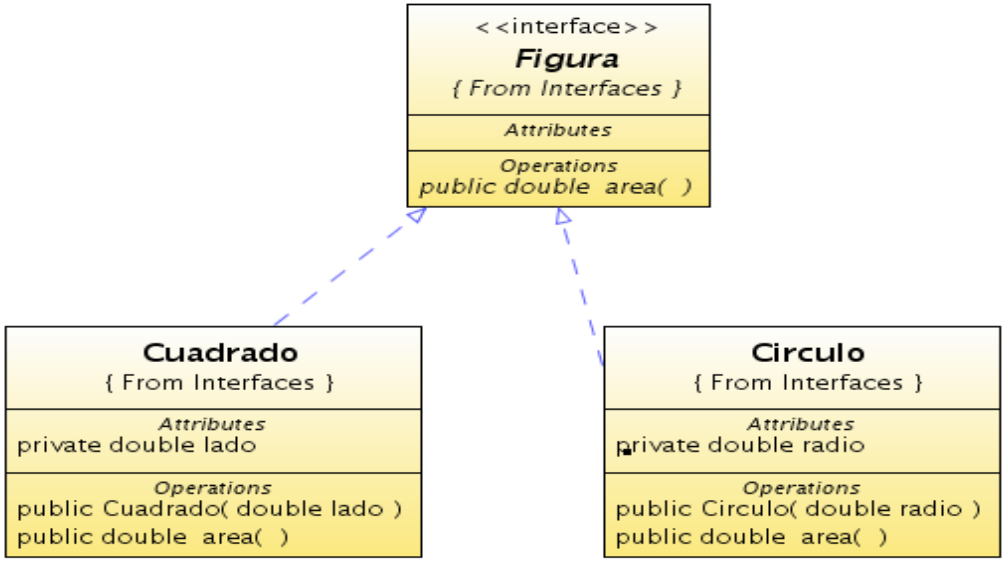
- En java usando la palabra reservada abstract se define una clase/método como abstracto.
  - Una clase abstract puede no tener ninguna operación abstracta, pero una clase con un método abstracto debe ser definida como abstracta.
  - En UML, suele aparecer la palabra abstract a continuación del nombre de la clase.

```
abstract class Figura {
  protected double x;
  protected double y;
  public Figura (double x, double y)
{
    this.x = x; this.y = y;
  }
  public abstract double area ();
};
};
```

```
class Circulo extends Figura {
  private double radio;
  public Circulo (double x, double y, double radio)  {
    super(x,y);  this.radio = radio;
  }
  public double area ()  {return
Math.PI*radio*radio;  }
};
class Cuadrado extends Figura {
  private double lado;
  public Cuadrado (double x, double y, double lado)  {
    super(x,y);  this.lado = lado;
  }
  public double area ()  {    return lado*lado;  }
```

3.5 Interfaces (Java)

- Herencia: Interfaces
  - Un interface es una colección de declaraciones de métodos (sin implementación) junto con atributos estáticos y constantes, es decir, es la definición de una clase "esquema" sin implementación que tienen que cumplir todas las clases que lo implementen.
  - Para crear una interfaz, se usa la palabra clave interface en vez de la palabra clave class. En las clases que implementan la interfaz en vez de extends se usa la palabra implements.
  - En UML se representa mediante una clase con la palabra interface.



3.5 Interfaces (Java)

- Herencia: Interfaces
  - Las clases implementadoras debe implementar los métodos definidos en la interfaz. Pueden existir interfaces anidados, donde la clases que finalmente implemente los interfaces, debe implementar todos los métodos definidos en las clases interfaces.
  - Diferencias entre un interface y una clase abstracta: Un interface es simplemente una lista de métodos no implementados, además puede incluir la declaración de constantes. Una clase abstracta puede incluir métodos implementados y no implementados o abstractos, miembros dato constantes y otros no constantes.
  - Mediante “interface” se puede implementar herencia múltiple, pues una clase solamente puede derivar extends de una clase base, pero puede implementar varios interfaces. Los nombres de los interfaces se colocan separados por una coma después de la palabra reservada implements.
  - Los atributos definidos en las intefaces son estaticos y constantes, por tanto deben ser inicializado cuando se carga la clase por primera vez.

3.5 Interfaces (Java)

- Herencia Múltiple: Interfaces
- Una clase puede implementar varios interfaces simultáneamente, pese a que, en Java, una clase sólo puede heredar de otra clase (herencia simple de implementación, múltiple de interfaces).
  - Aspectos Positivos
    - Proporcionar mucha flexibilidad
    - Facilidad
  - Aspectos negativos
    - Complica el diseño
    - Problemas de eficiencia.
    - Herencia repetiva

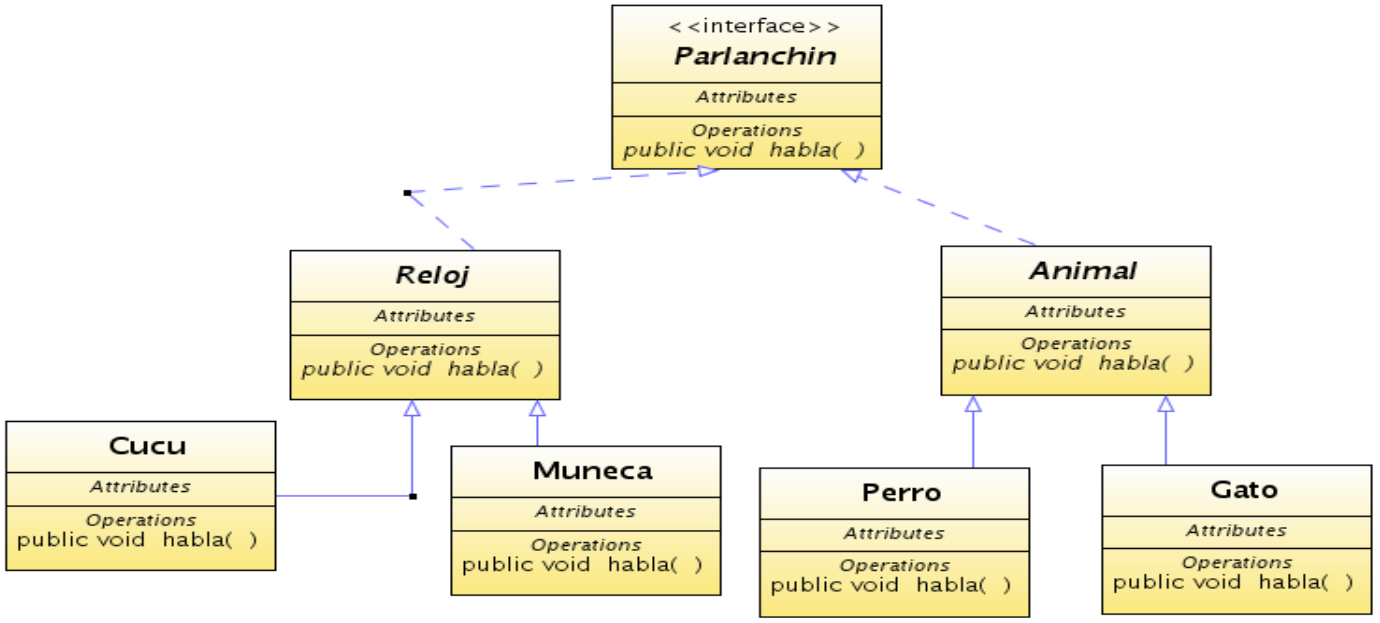
3.5 Interfaces (Java)

```
public interface Figura {
  public double area ();
}
```

```
class Circulo implements Figura {
  private double radio;
  public Circulo (double radio)  {
    this.radio = radio;
  }
  public double area ()   {
      return Math.PI*radio*radio;
  }
}
class Cuadrado implements Figura {
  private double lado;
  public Cuadrado (double lado)   {
    this.lado = lado;
  }
  public double area ()   {
    return lado*lado;
  }
}
```

3.5 Interfaces (Java)

- Definir la interfaz parlachin que contiene una función denominada *habla*. A continuación implementaremos unas subclases Animal (con perro y Gato) y Reloj que implementen dicho método.
- Definir una interfaz de números aleatorios. Esta interfaz debe especificar un método 'double getAleatorio()'. Definir una clase 'NumeroAleatorio01' que devuelve números aleatorios entre 0 y 1 y una clase 'NumeroAleatorioCte' que devuelva siempre el mismo número. Nota: el método 'random()' de la clase 'Math' de Java devuelve un número (double) mayor o igual que 0 y menor que 1.



3.6 Polimorfismo en Java

- Podemos asignar instancias de las CD a la CB. **Problemas:**
  - Perdida de contenido semántico de la instancia
  - Perdida de atributos
  - Perdida de comportamiento
- ¿Podemos asignar instancias de una clase_base a una instancia de una clase_derivada --> NO.

```
int main(){
        Publicación P1;
        Libro L1;
        P1=L1; // correcto, con perdidas
        L1.mostrar();
        P1.mostrar();  //Perdida de comportamiento.
        L1=P1; // Incorrecto (no todos las publicaciones son libros). Fallo de compilación
}
```

Solución → POLIMORFISMO

3.6 Polimorfismo en Java

- Problemas relacionados con herencia:
  - Perdida de contenido semántico en asignaciones de objetos jerárquicos. Por ejemplo imaginemos que se quieren almacenar en una estructura de datos publicaciones y libros ¿Cómo se puede declarar un vector de Publicación y Libros? ¿Y si se define como un vector de “publicación” y se vincula un libro --> Perdida de contenido semántico?
  - ¿Qué sucede si se realiza una llamada desde un objeto alumno al método cumpleanios?
    - ✓ Se incrementa la edad correctamente
  - Se produce una llamada a mostrar ¿De qué clase, Alumno o Persona?. Se realiza la llamada a la clase persona, debido a que se olvida la clase del objetivo que recibe el mensaje original que se ha vinculado con el método
- El problema es **Ligadura estática**.

Solución: POLIMORFISMO

3.6 Polimorfismo en Java

- En Java
  - Java incorpora de forma nativa el polimorfismo, gracias a lo que se conoce como ligadura tardía o dinamica, es decir, cuando se resuelve la correspondencia entre el método y el objeto en tiempo de ejecución, en lugar de en tiempo de compilación (ligadura temprana).
  - Todos los objetos tienen un puntero a la clase Base de --> Object
  - Para la conversión entre objetos de distintas clases, Java exige que dichas clases estén relacionadas por herencia (una deberá ser sub-clase de la otra).
  - Se realiza una conversión implícita o automática cuando se quiere tiene un objeto definido mediante una clase base pero realmente es un objeto de la clase derivada y se quiere ejecutar un método de la clase derivada.

3.6 Polimorfismo en Java

- Duplicación de objetos polimórfico
  - Con los objetos estáticos, para hacer una copia del objeto se puede usar dos soluciones:
    - ✓ Constructor Copia
    - ✓ Operador Asignación
  - Con objetos polimórficos, puesto que el tipo de dato del objeto es un puntero a la clase base, se realizaría la llamada al operador asignación o constructor copia de la clase base.
  - ~~Solución: Operador asignación virtual, no existe.~~
  - Solución real: método **clone** que lo simule

3.6 Polimorfismo en Java

- Método clone() en java esta definido en la clase **Object**
  - El tipo objeto a copiar debe haber declarado que es susceptible de copia implementando el interfaz vacío Clonable
  - Es un método protegido de Object, por tanto para que pueda ser invocado desde otro paquete hay que redeclararlo como público y simplemente invocar a la clase base.

```
public class ClaseClonable implements Cloneable {
// Otro código de la clase
   public Object clone () throws CloneNotSupportedException   {
            return super.clone();
   }
}
```

3.6 Polimorfismo en Java

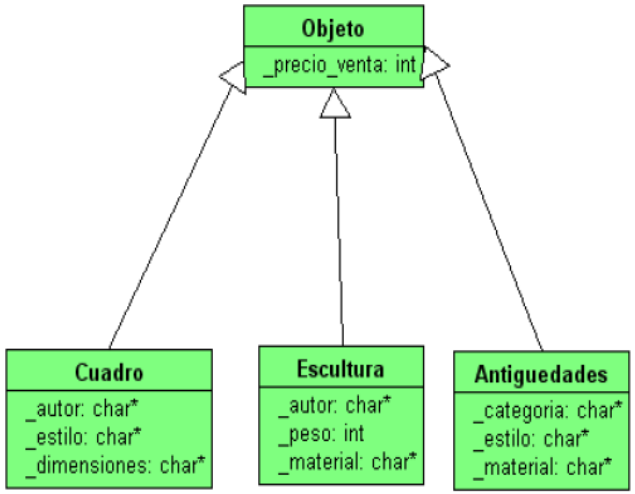
- Si los miembros de una clase son otros objetos que necesiten ser copiados habrá que añadir el código necesario
  - Llamar a los métodos clone() de cada uno de los objetos de la composición
- Reglas generales
  - Implementar el interfaz Cloneable si se desea que la clase se pueda copiar
  - No utilizar constructores en la implementación de clone() si no llamadas a los métodos clone()
  - Cuidado todas las subclases de una clase Clonable también son susceptibles de ser clonadas.

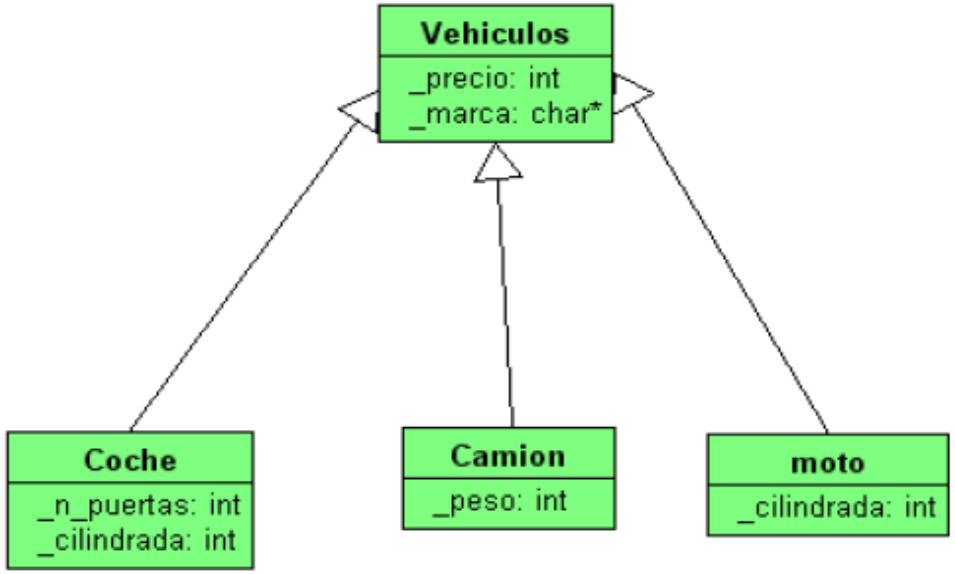
```
public class Persona implements Cloneable {
        String nombre;
        String apellidos;
        .......
        public Object clone () throws
CloneNotSupportedException{
                return super.clone();
        }

}
```

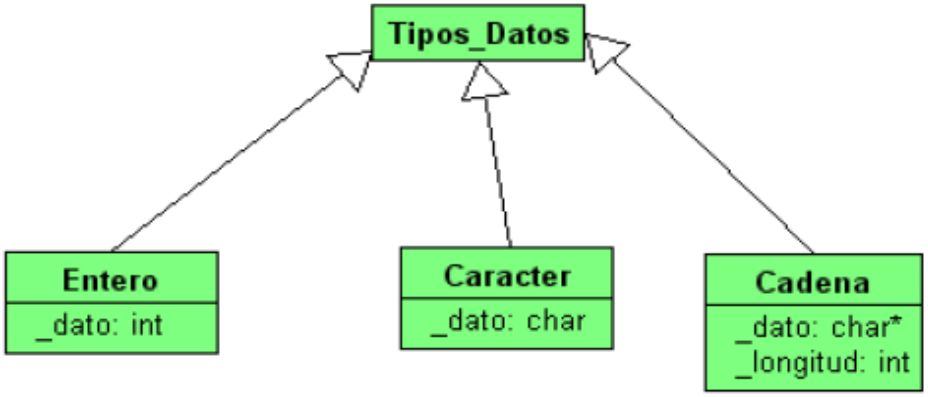
Bibliografía Recomendada

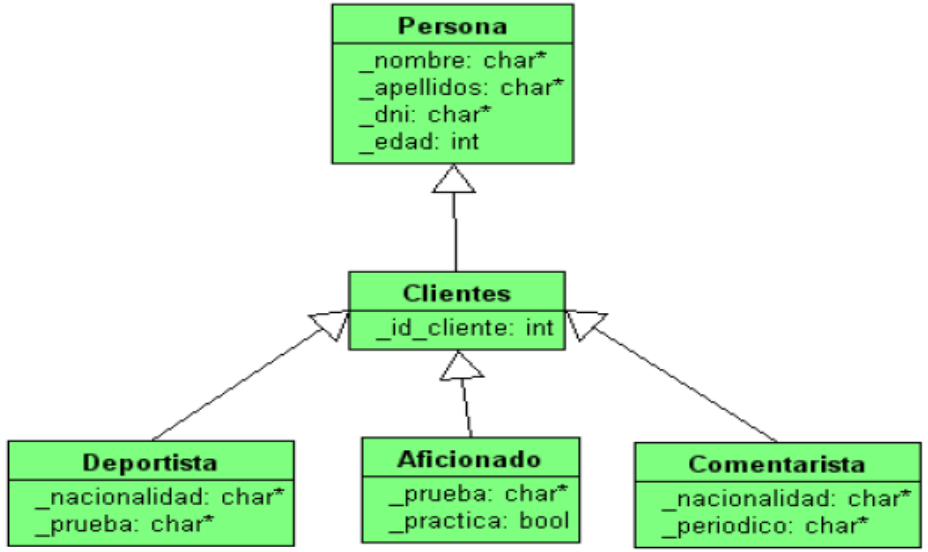
- Libros en Java
  - Piensa en Java. 4ª Edición. Bruce Eckel. Pearson Prentice Hall.
  - Core Java 2. Autores Cay S. Horstmann Y Gary Cornell. Editorial Pearson Educación
  - Java 2. Manual De Programación. Luis Joyanes Aguilar; Matilde Fernández Azuela. Editorial McGraw-Hill
- URL usadas:
  - http://arodm.blogspot.com/search/label/Dise%C3%B1o
- Apuntes de otra universidad:
  - Ingeniería del Software
    - ✓ http://www.vc.ehu.es/jiwotvim/ISOFT2010-2011/ingSoftware1011.htm

Ejercicios


Objeto
_precio_venta: int
Cuadro
_autor: char*
_estilo: char*
_dimensiones: char*
Escultura
_autor: char*
_peso: int
_material: char*
Antiguedades
_categoria: char*
_estilo: char*
_material: char*


Vehiculos
_precio: int
_marca: char*
Coche
_n_puertas: int
_cilindrada: int
Camion
_peso: int
moto
_cilindrada: int


Tipos_Datos
Entero
_dato: int
Caracter
_dato: char
Cadena
_dato: char*
_longitud: int


Persona
_nombre: char*
_apellidos: char*
_dni: char*
_edad: int
Clientes
_id_cliente: int
Deportista
_nacionalidad: char*
_prueba: char*
Aficionado
_prueba: char*
_practica: bool
Comentarista
_nacionalidad: char*
_periodico: char*