



# DIAGRAMA DE CLASES

Julián Santoyo  
*jsdiaz@unab.edu.co*  
**6436111 Ext. 459**

# DIAGRAMA DE CLASES

- Un diagrama de clases sirve para visualizar las relaciones entre las **clases** que involucran el sistema, las cuales pueden ser asociativas, de herencia y de agregación.
- Un diagrama de clases esta compuesto por los siguientes elementos:
  - Clase: atributos, métodos y visibilidad.
  - Relaciones: Herencia, Composición, Agregación, Asociación y Uso.



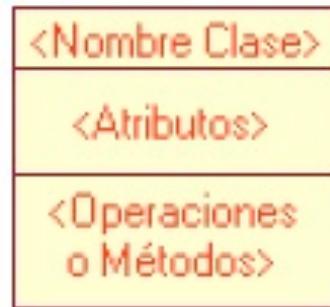
# CLASE

Es la unidad básica que **encapsula** toda la información de un Objeto (un objeto es una **instancia** de una **clase**). A través de ella podemos modelar el entorno en estudio (una Casa, un Auto, una Cuenta Corriente, etc.).



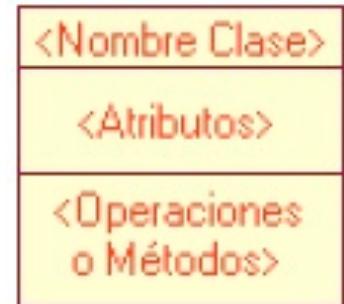
# CLASE

En UML, una clase es representada por un rectángulo que posee tres divisiones:



# CLASE

- **Superior:** Contiene el nombre de la **Clase**
- **Intermedio:** Contiene los **atributos** (o variables de instancia) que caracterizan a la Clase (pueden ser private, protected o public)
- **Inferior:** Contiene los **métodos** u operaciones, los cuales son la forma como interactúa el objeto con su entorno (dependiendo de la visibilidad: private, protected o public)



# EJEMPLO

Una Cuenta Corriente que posee como característica:

- Saldo

Puede realizar las operaciones de:

- Depositar
- Girar
- Estado Actual

[www.gliffy.com](http://www.gliffy.com)



# EJEMPLO

Cuenta Corriente
Saldo : int
depositar(monto:int) : void girar (monto:int) : boolean estadoActual () : int



# ATRIBUTOS Y MÉTODOS

## Atributos:

- Los atributos o características de una Clase pueden ser de tres tipos, los que definen el grado de comunicación y visibilidad de ellos con el entorno, estos son:



# ATRIBUTOS Y MÉTODOS

- **public** (+, ): Indica que el atributo será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.
- **private** (-, ): Indica que el atributo sólo será accesible desde dentro de la clase (sólo sus métodos lo pueden accesar).
- **protected** (#, ): Indica que el atributo no será accesible desde fuera de la clase, pero si podrá ser accesado por métodos de la clase además de las subclases que se deriven (ver herencia).



# ATRIBUTOS Y MÉTODOS

## Métodos:

- Los métodos u operaciones de una clase son la forma en como ésta interactúa con su entorno, éstos pueden tener las características:

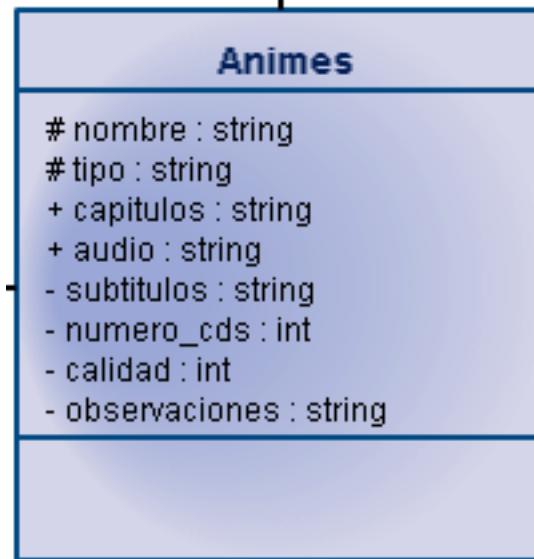
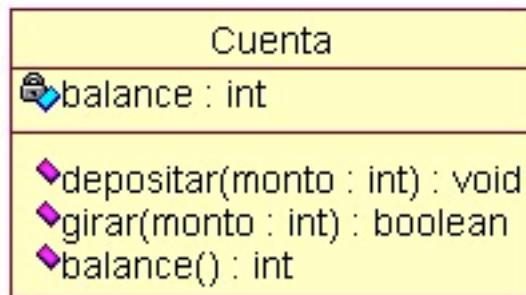


# ATRIBUTOS Y MÉTODOS

- **public** (+,  ): Indica que el método será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.
- **private** (-,  ): Indica que el método sólo será accesible desde dentro de la clase (sólo otros métodos de la clase lo pueden accesar).
- **protected** (#,  ): Indica que el método no será accesible desde fuera de la clase, pero si podrá ser accedido por métodos de la clase además de métodos de las subclases que se deriven (ver herencia).

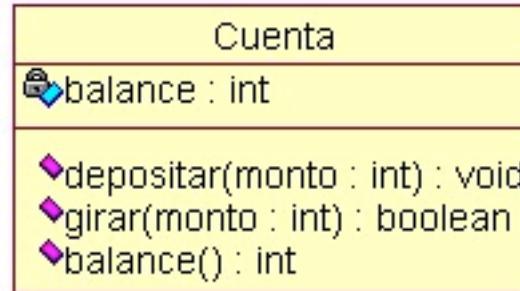


# ATRIBUTOS Y MÉTODOS



# RELACIONES ENTRE CLASES

- Ahora ya definido el concepto de Clase, es necesario explicar como se pueden interrelacionar **dos o más clases** (cada uno con características y objetivos diferentes).



# RELACIONES ENTRE CLASES

Antes es necesario explicar el concepto de cardinalidad de relaciones: En UML, la cardinalidad de las relaciones indica el grado y nivel de dependencia, se anotan en cada extremo de la relación y éstas pueden ser:

- **uno o muchos:**  $1..*$  ( $1..n$ )
- **0 o muchos:**  $0..*$  ( $0..n$ )
- **número fijo:**  $m$  ( $m$  denota el número)

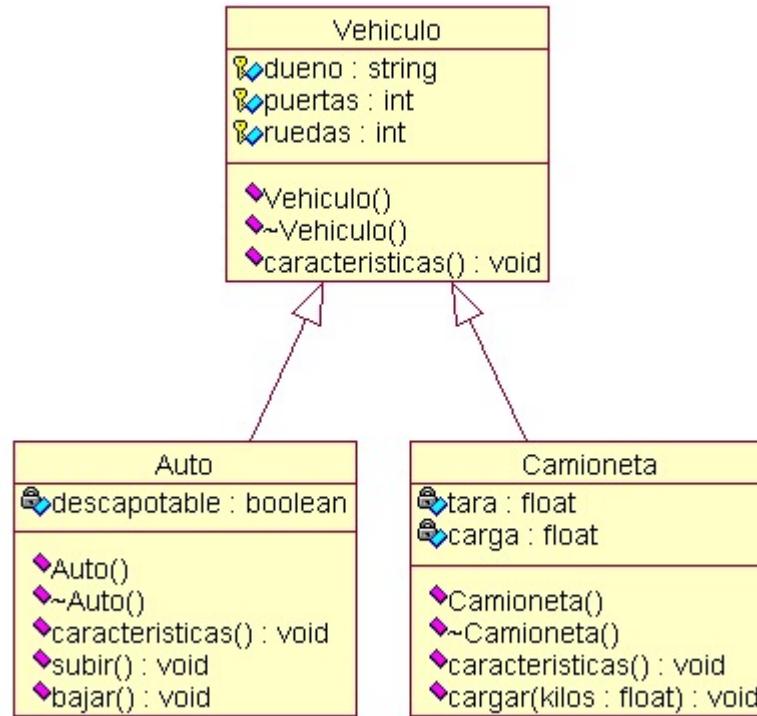


# Herencia (Especialización / Generalización)

- Indica que una subclase **hereda** los **métodos** y **atributos** especificados por una **Super Clase**, por ende la **Subclase** además de poseer sus propios métodos y atributos, poseerá las características y atributos **visibles** de la Super Clase (public y protected), ejemplo:



# Herencia (Especialización / Generalización)



# Herencia (Especialización / Generalización)

- En la figura se especifica que Auto y Camioneta heredan de Vehículo, es decir, Auto posee las Características de Vehículo (dueño, puertas, ruedas etc.) además posee algo particular que es **descapotable**. Camioneta también hereda las características de Vehículo (dueño, puertas, ruedas etc.) pero posee como particularidad propia Tara y Carga.



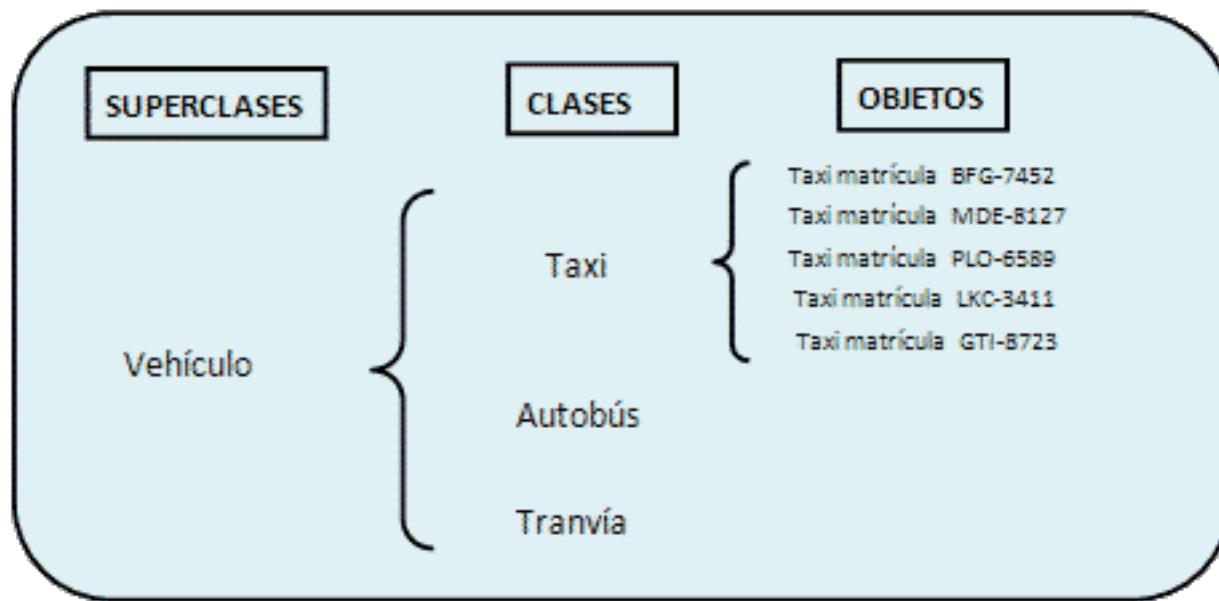
# Herencia (Especialización / Generalización)

- Cabe destacar que fuera de este entorno, lo único "visible" es el método `características()` aplicable a `instancias` de vehículo, auto y camioneta, pues tiene definición `pública`, en cambio atributos como `descapotable` no son visibles por ser privados.



# Objetos

- Un objeto es una instancia de una clase.  
Por ejemplo el taxi matrícula BFG-7452 es una instancia de la clase Taxi.



# Objetos

## Ejercicio

- Considera que queremos representar los aviones que operan en un **aeropuerto**. Crea un esquema análogo al que hemos visto para vehículos, pero en este caso para aviones. Define cuáles podrían ser las **clases** y cuáles podrían ser algunos **objetos** de una clase.





# Asociación

- La relación entre clases conocida como Asociación, permite asociar objetos que **colaboran entre si**, para alcanzar una meta. Cabe destacar que no es una **relación fuerte**, es decir, el tiempo de vida de un objeto **no depende** del otro.





# Asociación

- Ejemplo:





# Asociación

- Ejemplo:



- Un cliente puede tener asociadas muchas Ordenes de Compra, en cambio una orden de compra solo puede tener asociado un cliente.





# Asociación

Ejercicio:

- Representar la relación:
  - El cliente usa tarjeta de crédito.
  - El ingeniero usa una computadora





unab



# Asociación

```
public class Customer {  
  
    private int id;  
    private String firstName;  
    private String lastName;  
    private CreditCard creditCard;  
  
    public Customer() {  
        //Lo que sea que el constructor haga  
    }  
  
    public void setCreditCard(CreditCard creditCard) {  
        this.creditCard = creditCard;  
    }  
  
    // Más código aquí  
}
```





# Asociación

- **Customer** es independiente de **CreditCard**, puesto que el cliente puede existir sin necesidad de tener asignada una tarjeta de crédito.
- Se puede asignar o retirar la tarjeta de crédito, sin que la existencia del **Cliente** se vea afectada (No debería verse afectada, esto significa que **Customer** no debe tener problemas si no hay una **CreditCard** presente).





# Asociación

- Podemos crear un objeto de tipo `Customer` y asignarle un `CreditCard` más tarde mediante el método `setCreditCard`.
- Llamaremos a las clases `Customer` `clase contenedora`.





# Agregación y Composición



- Para modelar objetos **complejos**, bastan los **tipos de datos básicos** que proveen los lenguajes: enteros, reales y secuencias de caracteres. Cuando se requiere componer objetos que son instancias de clases definidas por el desarrollador de la aplicación, tenemos dos posibilidades:





# Agregación y Composición



- **Por Valor:** Es un tipo de relación estática, en donde el tiempo de vida del objeto incluido esta condicionado por el tiempo de vida del que lo incluye. Este tipo de relación es comúnmente llamada **Composición** (el Objeto base se construye a partir del objeto incluido).





# Agregación y Composición



- **Por Referencia:** Es un tipo de relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye. Este tipo de relación es comúnmente llamada **Agregación** (el objeto base utiliza al incluido para su funcionamiento).

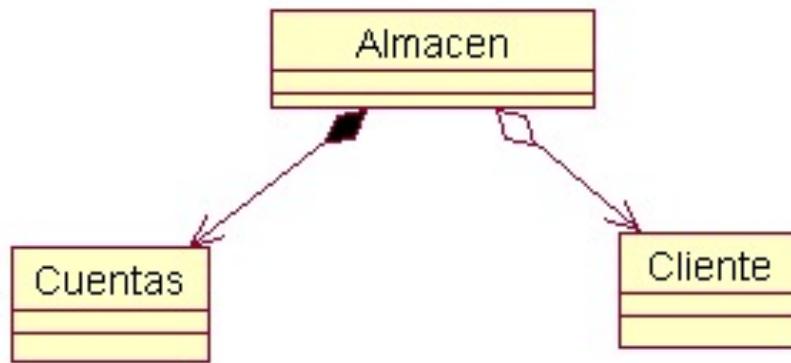




# Agregación y Composición



- Ejemplo 1:





# Agregación y Composición



En donde se destaca que:

- Un Almacén **posee** Clientes y Cuentas (los rombos van en el objeto que **posee las referencias**).
- Cuando se **destruye** el Objeto Almacén también son destruidos los objetos Cuenta asociados, **en cambio** no son afectados los objetos Cliente asociados.





# Agregación y Composición



- La composición (**por Valor**) se destaca por un rombo relleno.
- La agregación (**por Referencia**) se destaca por un rombo transparente.

La flecha en este tipo de relación indica la navegabilidad del objeto referenciado. Cuando no existe este tipo de particularidad la flecha se elimina.



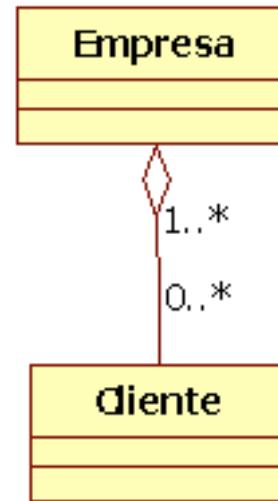


# Agregación y Composición



## Ejemplo 2:

- Tenemos una clase Empresa.
- Tenemos una clase Cliente.
- Una empresa agrupa a varios clientes.

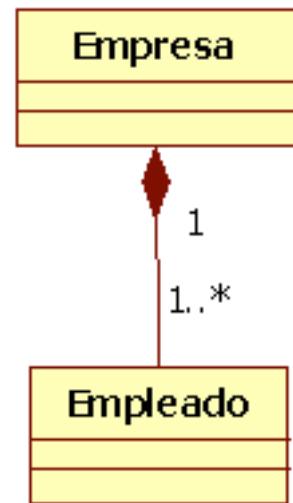




# Agregación y Composición



- Tenemos una clase Empresa.
- Un objeto Empresa está a su vez compuesto por uno o varios objetos del tipo empleado.
- El tiempo de vida de los objetos Empleado depende del tiempo de vida de Empresa, ya que si no existe una Empresa no pueden existir sus empleados.





# Agregación y Composición



- Las relaciones de Agregación y Composición son dos tipos de especialización de la relación de Asociación.
- En la Composición hay una "relación de vida", es decir, el tiempo de vida de un objeto está condicionado por el tiempo de vida del objeto que lo incluye.

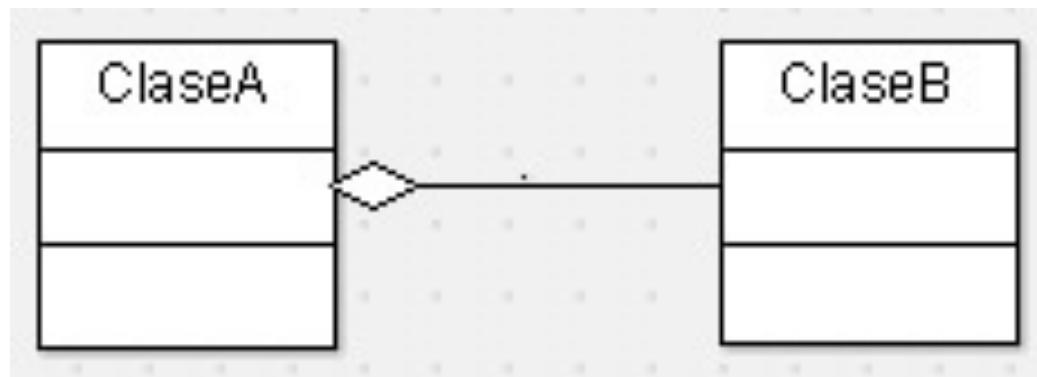




# Agregación



- La Clase A agrupa varios elementos del tipo Clase B.



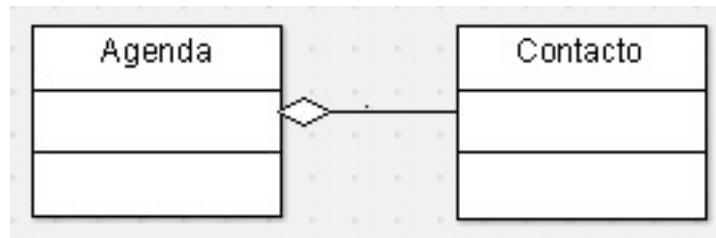


# Agregación



## Ejemplo

- Tenemos una clase Agenda.
- Tenemos una clase Contacto.
- Una Agenda agrupa varios Contactos.





# Agregación



## Ejemplo

- Un objeto de tipo **Ciudad** tiene una lista de objetos de tipo **Aeropuerto**, una ciudad tiene un número de aeropuertos. La cardinalidad del extremo que lleva el rombo, es siempre uno.

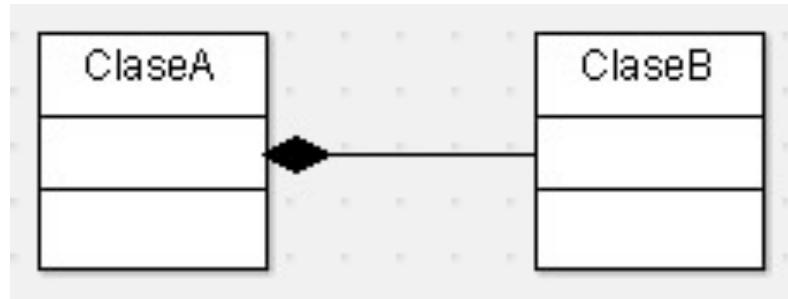




# Composición



- La Clase A agrupa varios elementos del tipo Clase B.
- El tiempo de vida de los objetos de tipo Clase B está condicionado por el tiempo de vida del objeto de tipo Clase A.



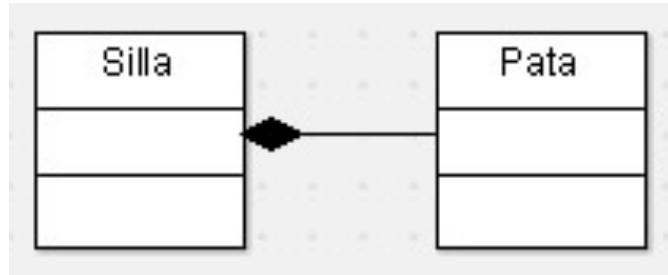


# Composición



## Ejemplo

- Tenemos una clase Silla.
- Un objeto Silla está a su vez compuesto por cuatro objetos del tipo Pata.
- El tiempo de vida de los objetos Pata depende del tiempo de vida de Silla, ya que si no existe una Silla no pueden existir sus Patas.



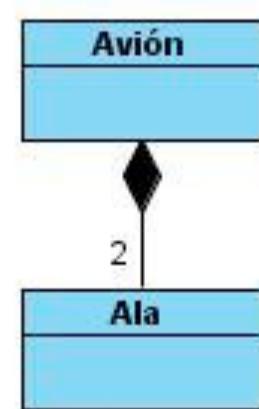


# Composición



## Ejemplo

- El avión tiene sentido por si solo, pero esta claro que esta compuesto de 2 alas, esta relación es de mucha fuerza, mucho más que el caso de los aeropuertos.





# Composición

Ejercicio:

- Representar la relación:
  - El **auto** tiene **llantas**.
  - El **portátil** tiene un **teclado**.





unab



# Composición

```
public class Laptop {  
    private String manufacturer;  
    private String model;  
    private String serviceTag;  
    private KeyBoard keyBoard = new KeyBoard();  
  
    public Laptop() {  
        //Lo que sea que el constructor haga  
    }  
}
```





# Composición

Los objetos que componen a la **clase contenedora**, deben existir desde el principio. (También pueden ser creados en el **constructor**, no sólo al momento de declarar las variables como se muestra en el ejemplo).

No hay momento (No debería) en que la **clase contenedora** pueda existir sin alguno de sus **objetos componentes**. Por lo que la existencia de estos objetos no debe ser abiertamente manipulada desde el exterior de la clase.



# Diferencias entre Composición y Agregación

	Agregación	Composición
Varias asociaciones comparten los componentes	Si	No
Destrucción de los componentes al destruir el compuesto	No	Si
Cardinalidad a nivel de compuesto	Cualquiera	0..1 ó 1
Representación	Rombo Transparente	Rombo Negro



# Dependencia o Instanciación (uso)

- Representa un tipo de relación muy particular, en la que una clase es instanciada (su instanciación es **dependiente** de otro objeto/clase). Se denota por una flecha punteada.



# Dependencia o Instanciación (uso)

- El uso más particular de este tipo de relación es para denotar la **dependencia** que tiene una clase de otra, como por ejemplo una aplicación gráfica que instancia una ventana (la creación del Objeto Ventana está **condicionado** a la instanciación proveniente desde el objeto Aplicación):



# Dependencia o Instanciación (uso)



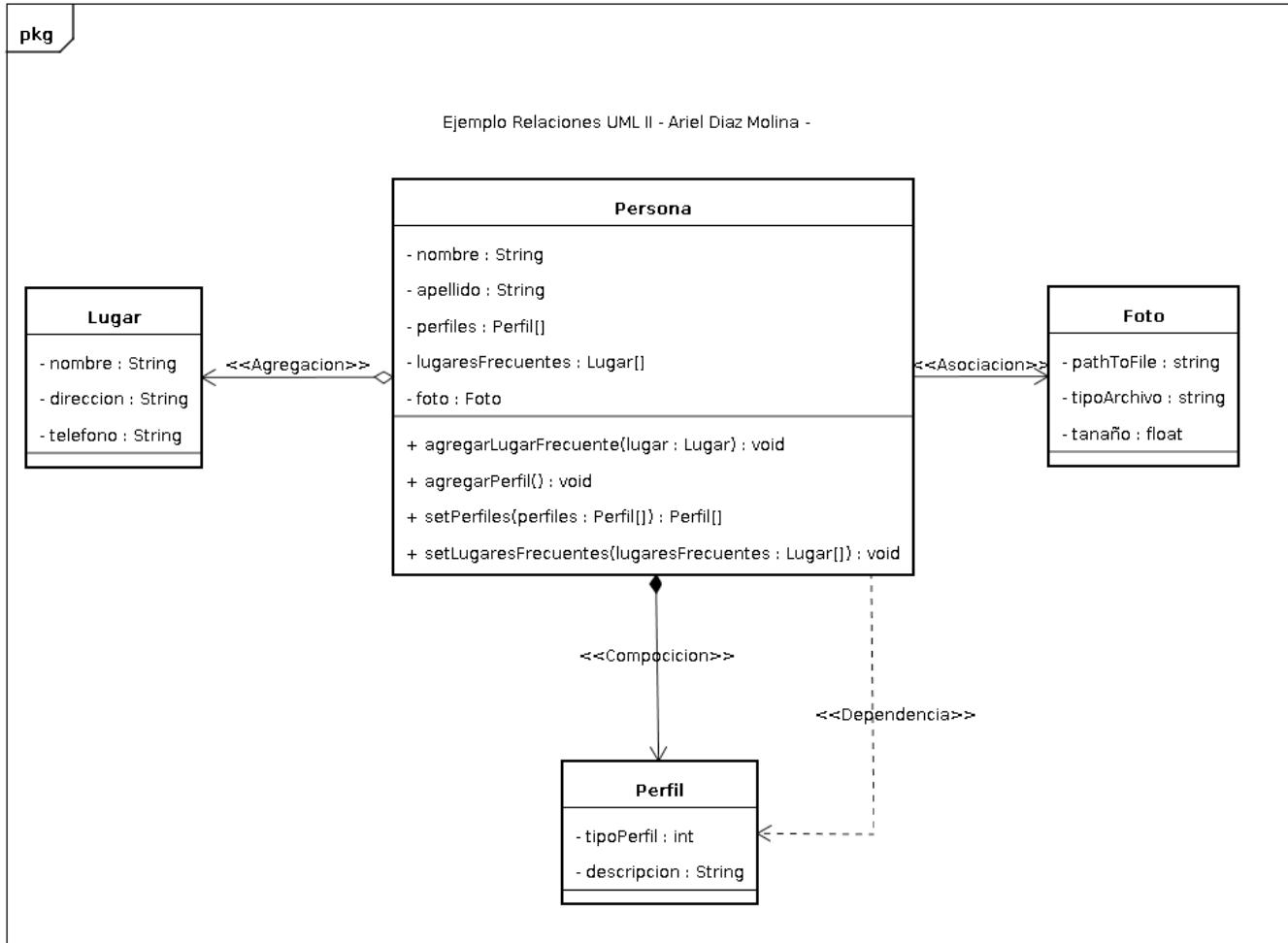
# Dependencia o Instanciación (uso)



- Cabe destacar que el objeto creado (en este caso la Ventana gráfica) **no se almacena** dentro del objeto que lo crea (en este caso la Aplicación).



# Composición, Agregación, Asociación



# Composición, Agregación, Asociación

```
import java.util.LinkedList;
import java.util.List;

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;
    }

    // No confundir 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;
    }
}
```



# Composición, Agregación, Asociación

- La clase comienza normalmente con la declaración de las **variables** de instancia. Donde **perfiles** y **lugaresFrecuentes** son dos **colecciones** (podrían ser **arrays**) que se transformarán en contenedores de elementos.



# Composición, Agregación, Asociación

- Al ser propiedades tienen **getters** y **setters** (**accessors** y **mutators**), que nada tienen que ver con la agregación y la composición.



# Composición, Agregación, Asociación

- Asociación

```
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, Agregación, Asociación

- La primera característica es que la clase contiene **dos métodos uno que agrega elementos a la colección y otro que los elimina de ella.**

```
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, Agregación, Asociación

- Los objetos **son pasados por parámetro**, no han sido instanciados dentro del método, es decir no hemos realizado el **new** del objeto.

```
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, Agregación, Asociación

- Ha "**nacido**" en cualquier otra parte y se lo hemos pasado por parámetro al método para ser agregado a la lista **lugaresFrecuentes**. En otras palabras, el objeto **Persona** podría morir y el objeto **Lugar** aún podría mantener una referencia activa en alguna otra parte de nuestro código por lo tanto sus ciclos de vida no estarían atados. No nace ni muere, dentro del objeto **Persona**.



# Composición, Agregación, Asociación

- ¿Cuál es la diferencia con la **composición**?

```
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); // aca lo quitamos de la lista
}
```



# Composición, Agregación, Asociación

- La composición también tiene los métodos para **agregar** y **borrar**. Pero "**el new del objeto se realiza dentro del método agregar**".

```
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); // aca lo quitamos de la lista
}
```



# Composición, Agregación, Asociación

- La instanciación del objeto p se realiza dentro del método agregar y la referencia no se devuelve (es **void** o boolean).
- La variable de referencia local va a dejar de existir una vez que el **método termine** de ejecutarse, y el ciclo de vida de esa instancia en particular va a quedar atada a la **lista**, y por ende a la **Persona**.

```
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); // aca lo quitamos de la lista
}
```



# Composición, Agregación, Asociación

- Una vez que el objeto **Persona** no se referencie más, (o sea muera, técnicamente esto no es así) el objeto **lista**, quedará sin referencia, y por lo tanto sus elementos también.
- Se deberá crear primero una instancia de **Persona**, para después poder agregar un **Perfil**. Empezando así a "atar" el ciclo de vida de un Perfil, al de una Persona. En cuanto al método **remover**, no existe nada de extraordinario, simplemente quitamos un elemento de la lista.

```
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); // aca lo quitamos de la lista
}
```



# Composición, Agregación, Asociación

- La Clase **Foto** se deduce del diagrama y no tiene complicaciones, son simplemente 3 variables:

```
public class Foto {  
    private String pathToFile;  
    private String tipoArchivo;  
    private float tamano;  
  
    public void setPathToFile(String s){  
        this.pathToFile = s;  
    }  
    public String getPathToFile(){  
        return this.pathToFile;  
    }  
}
```





# DIAGRAMA DE CLASES

Julián Santoyo  
*jsdiaz@unab.edu.co*  
**6436111 Ext. 459**