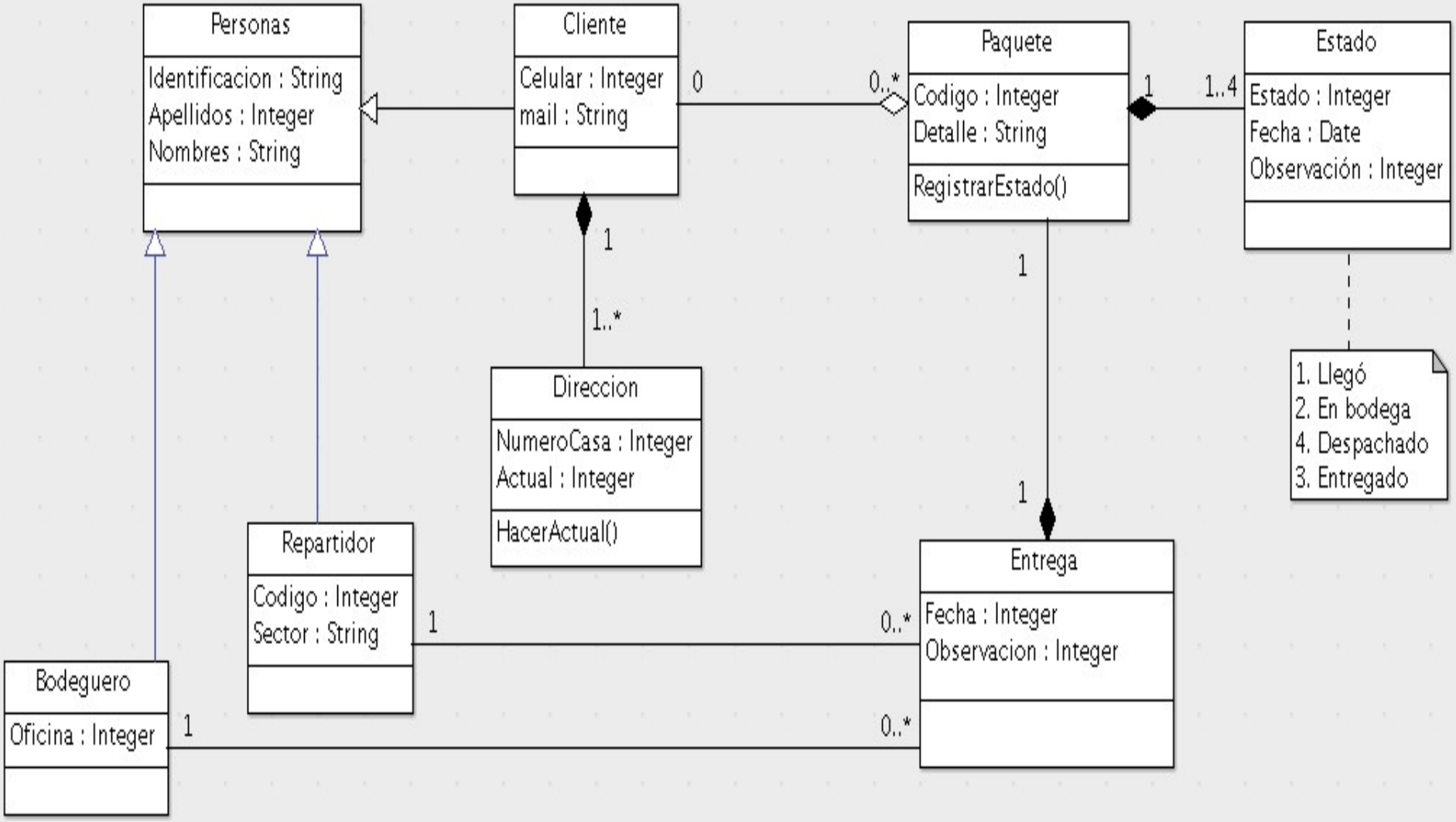
UTPL

UNIVERSIDAD TÉCNICA PARTICULAR DE LOJA

Agenda

- Relaciones entre clases con UML
  - Herencia
  - Agregación
  - Composición
  - Ejemplos de relaciones entre clases con UML
- Del diseño UML a la codificación con java
  - Herencia
  - Agregación
  - Composición
  - Ejemplos

Elementos Diagrama de Clases



1. Relaciones entre clases

- En una aplicación mínimamente compleja habrá varias clases cuyos métodos se llamarán unos a otros.
- Para que un método de la clase A pueda llamar a otro método de la clase B, la clase A debe poseer una referencia a un objeto de la clase B.
- Esta relación de posesión se representa con líneas que unen las distintas clases en el diagrama.

1. Relaciones entre clases

- Las relaciones pueden ser varias:
  - Asociación
  - Agregación
  - Composición
  - Dependencia
- También puede ocurrir que varias clases tengan operaciones o atributos en común y queramos abstraerlos utilizando mecanismos conocidos de los lenguajes orientados a objetos como la herencia y los interfaces.

1. Relaciones entre clases

- UML también permite representar estas relaciones entre clases mediante:
  - Generalización (equivalente a la herencia en POO)
  - Realización (equivalente a interfaces en Java)

2. Asociación

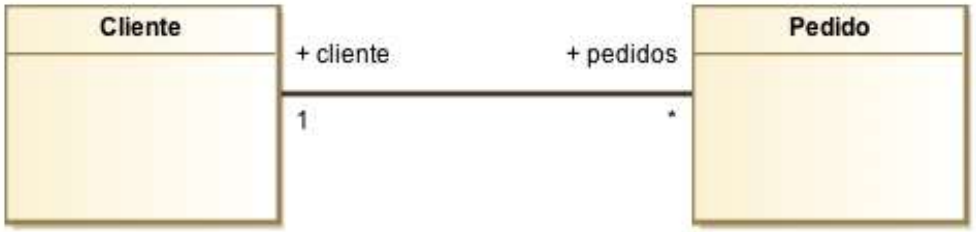
- Es el tipo de relación más frecuente entre clases.
- Existe una relación de asociación entre Clase A y Clase B cuando en Clase A hay un atributo de tipo Clase B y/ o viceversa.
- Se representa como una línea continua, que puede estar o no acabada en una flecha en “V”, según la navegabilidad.
- Además se suelen indicar los roles y la multiplicidad.

2. Asociación

- Navegabilidad
  - Si la flecha apunta de la ClaseA a la ClaseB, se lee como "ClaseA tiene una ClaseB" y se dice que la asociación o navegabilidad es unidireccional.
    - Traducido a Java, esto significa que hay un atributo en la clase A que hace referencia a un objeto de la clase B.
  - Si no dibujamos la flecha, se lee "ClaseA tiene una ClaseB y ClaseB tiene una ClaseA", y se dice que la asociación o navegabilidad es bidireccional.
    - En Java, ambas clases tendrían atributos que se hacen referencia recíprocamente.

2. Asociación

- Ejemplo:
  - Supongamos una aplicación de gestión con las clases Cliente y Pedido. En el siguiente diagrama, Cliente guarda información sobre los pedidos y Pedido tiene información sobre el cliente que lo realizó. La navegabilidad es, por tanto, bidireccional:

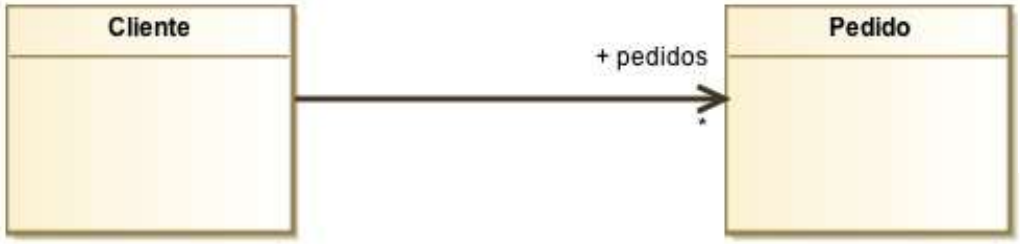



```
public class Cliente {
    private String Nombre;
    private ArrayList<Pedido> pedidos;
}
```

```
public class Pedido {
    private int Orden;
    private Cliente cliente;
}
```

2. Asociación

- Ejemplo de asociación unidireccional

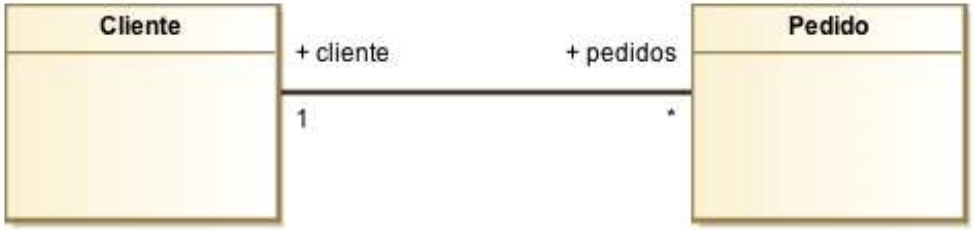



```
public class Cliente {
    private String Nombre;
    private ArrayList<Pedido> pedidos;
}

public class Pedido {
    private int Orden;
}
```


```
public class Cliente {
    private String Nombre;
}

public class Pedido {
    private int Orden;
    private Cliente cliente;
}
```

2. Asociación

- Roles y multiplicidad
  - El rol o papel es el nombre de los atributos que intervienen en la relación.




```
public class Cliente {
    private String Nombre;
    private ArrayList<Pedido> pedidos;
}
```

  - En el ejemplo, se dice que “la clase Pedido asume el papel pedidos” en la asociación, lo que simplemente significa que el atributo de Cliente que hace referencia al vector de objetos Pedido se llamará “pedidos”.

2. Asociación

- Es importante notar que el nombre de los atributos se escribe en el lado contrario a la clase que los contiene.
- Un error común cuando representamos asociaciones es volver a incluir los atributos dentro de la clase. Lo siguiente sería incorrecto:

2. Asociación

- Multiplicidad
  - Se rotula en los extremos de la relación.
  - Indica cuántos objetos de una clase se pueden relacionar como mínimo y como máximo con objetos de la otra clase y se expresa del siguiente modo:

muliplicidad mínima .. multiplicidad máxima

  - Cuando las multiplicidades mínima y máxima son iguales, se suele representar con un único número.
  - La multiplicidad de tipo “muchos” se representa con un asterisco: *.
  - Por último, la multiplicidad 0..* (cero o más) se suele expresar como simplemente *.

2. Asociación

- Un cliente se puede relacionar 0 o más pedidos (multiplicidad *, equivalente a 0..*)
- Un pedido debe tener un (y solo un) cliente (multiplicidad 1, equivalente a 1.. 1).

- Ejemplo:

2. Asociación

- La multiplicadad puede ser:
  - 1.. 1: Una instancia de cliente debe estar relacionada con exactamente una instancia de película, ni más ni menos.
  - 1: Significa lo mismo que 1.. 1.
  - 0..*: Un cliente puede no tener ninguna película alquilada, o puede tener varias (sin límite).
  - *: Significa lo mismo que 0..*
  - 0.. 3: Un cliente puede alquilar entre 0 y 3 películas.
  - 2.. 3: Un cliente debe alquilar como mínimo 2 películas, pero como mucho puede alquilar 3

3. Agregación y Composición

- Según la referencia de UML, la agregación y la composición son "**asociaciones que representan una relación entre un todo y sus partes**".
- Se puede leer como "ClaseB es un componente de ClaseA", o también "ClaseA está compuesto por ClaseB"
- Las diferencias entre ellas son:
  - **Agregación**, los objetos "parte" pueden seguir existiendo independientemente del objeto "todo".

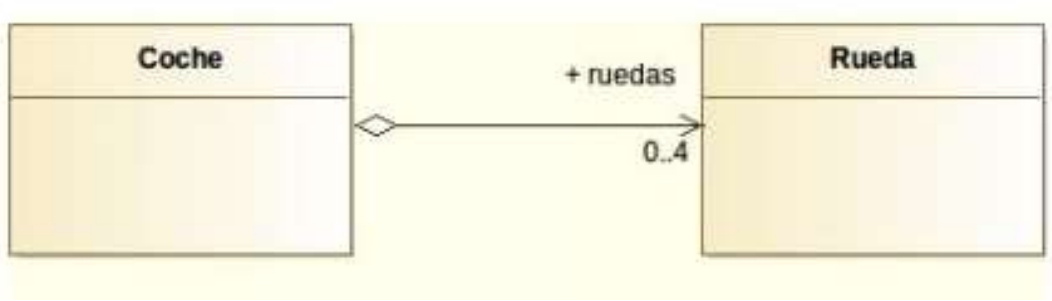
3. Agregación y Composición

  - **Composición**, por el contrario, la vida de los objetos compuestos está íntimamente ligada a la del objeto que los compone, de manera que si el “todo” se destruye, las “partes” también se destruyen.

- La agregación se representa uniendo las dos clases (todo/parte) con una línea continua y poniendo un rombo hueco en la clase “todo”.
- La composición es igual, pero con el rombo relleno.

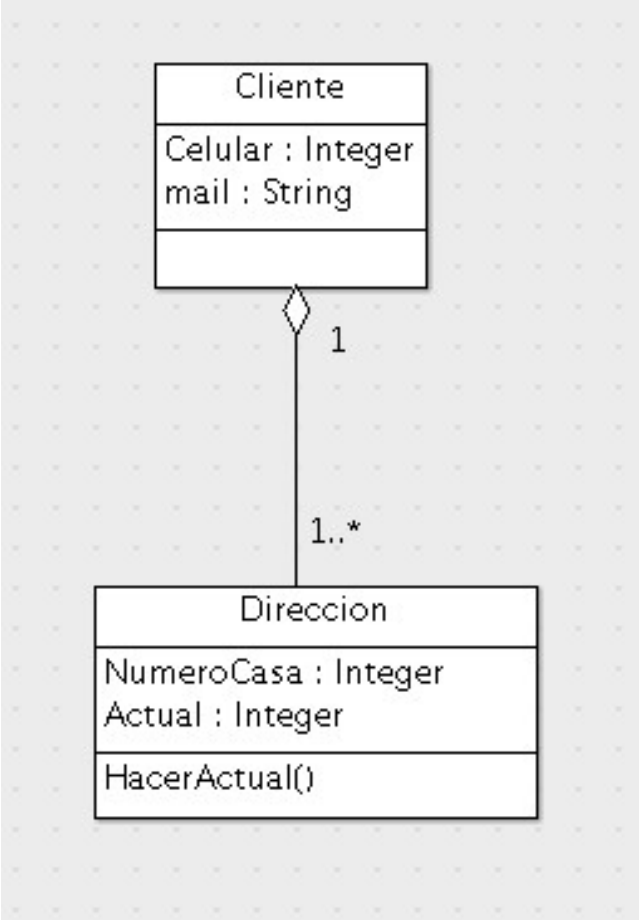
3. Agregación y Composición

- Agregación



La primera relación indica que el Coche está compuesto por Ruedas. Las ruedas pueden existir por sí mismas, por tanto la relación es de agregación.

Agregación - Representación UML

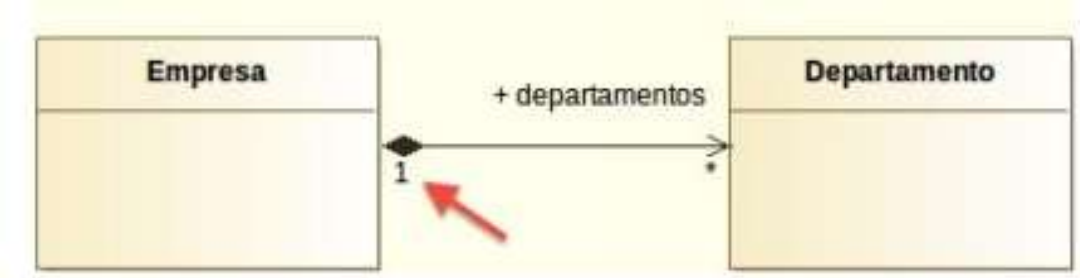


Agregación - Java

```
package CLA_tramil;

public class Cliente extends Personas {
  private String Celular;
  private String mail;
  private Direccion myDireccion;

    public Cliente(String Identificacion,String Apellidos, String Nombres, //padre
                   String Celular, String mail, Direccion myDireccion ) {//hijo
        super(Identificacion,Apellidos,Nombres);
        this.Celular = Celular;
        this.mail = mail;
        this.myDireccion = myDireccion;
    }
```

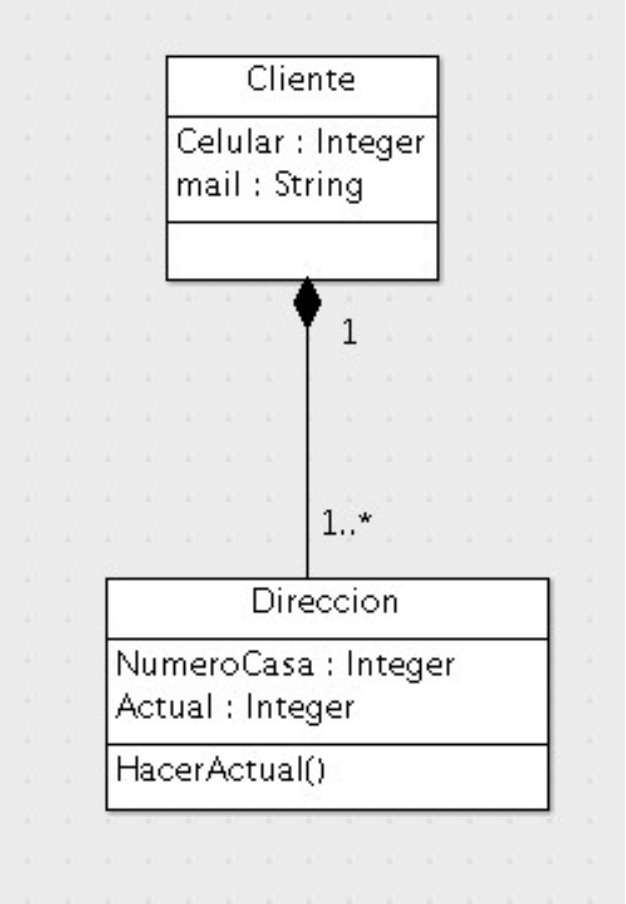
Agregación y Composición

- Composición



Empresa está compuesta por Departamentos. Los departamentos por sí mismos no pueden existir, ya que deben estar ligados a una empresa..

Composición – Representación UML



Composición

```
package CLA_tramil;

public class Cliente extends Personas {
  private String Celular;
  private String mail;
  private Direccion myDireccion;

  public Cliente(String Identificacion, String Apellidos, String Nombres, //padre
                 String Celular, String mail,//hijo
                 int casa, String telefono )  { //dirección
        super(Identificacion,Apellidos,Nombres);
        this.Celular = Celular;
        this.mail = mail;
        this.myDireccion = new Direccion(casa,telefono);
    }
```

4. Generalización

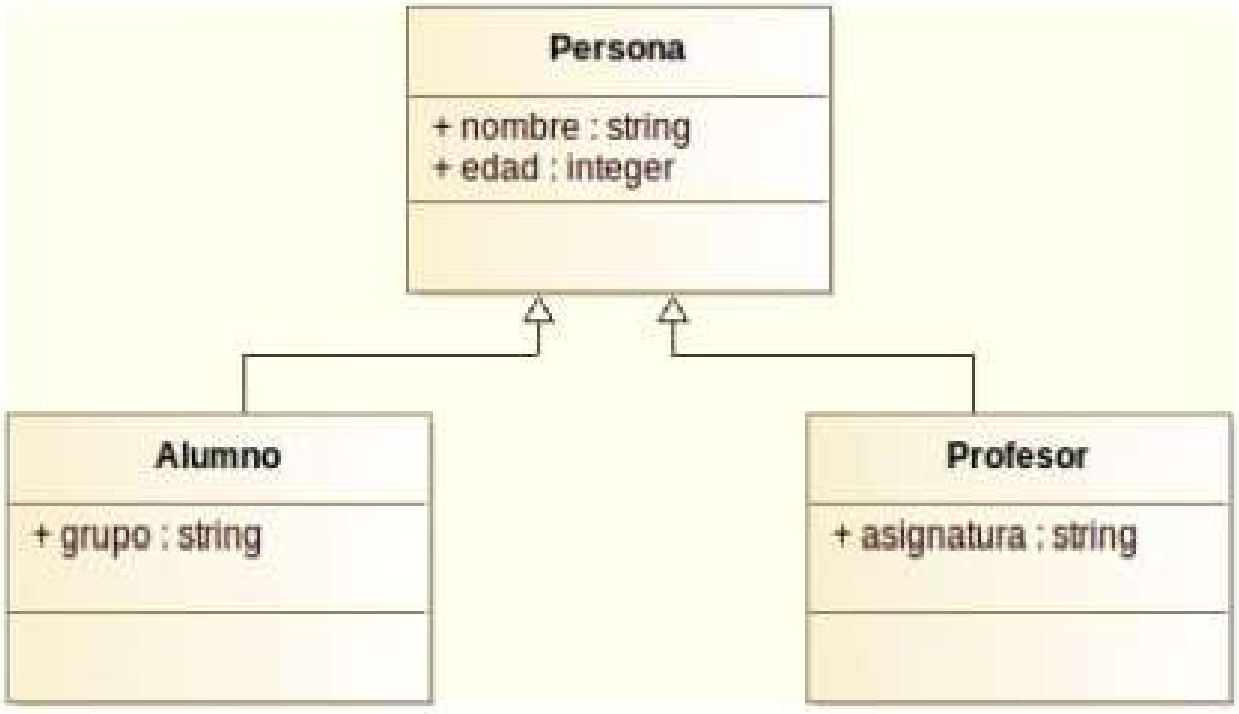
- La relación de generalización entre dos clases indica que una de las clases (la **subclase**) es una especialización de la otra (la **superclase**).
- Si ClaseA es la superclase y ClaseB la subclase, la generalización se podría leer como "ClaseB es una ClaseA" o "ClaseB es un tipo de ClaseA".
- En Java (y la mayoría de lenguajes orientados a objetos), la generalización se conoce como herencia.

4. Generalización

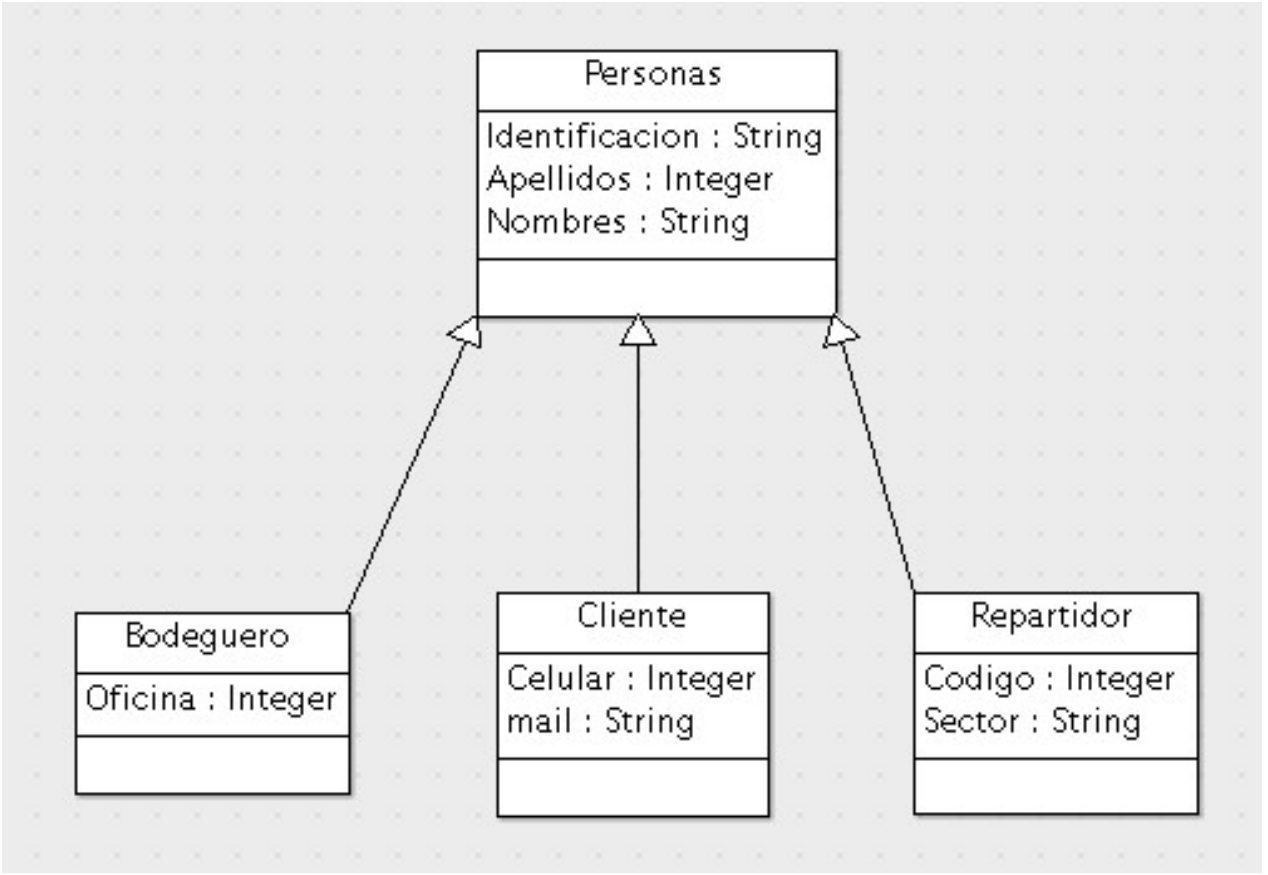
  - La herencia se utiliza para abstraer en una superclase los métodos y/ o atributos comunes a varias subclases.
  - A la subclase también se le puede llamar clase hija o clase derivada. A la superclase también se le puede llamar clase padre o clase base.
- La generalización se expresa con una línea acabada en una flecha triangular hueca, dibujada desde la clase hija hacia la clase padre.

4. Generalización

- Ejemplo



Herencia – Representación UML



Herencia - Java

```
public class Personas {
    private String Identificacion;
    private String Apellidos;
    private String Nombres;

    public Personas(String Identificacion, String Apellidos, String Nombres) {
        this.Identificacion = Identificacion;
        this.Apellidos = Apellidos;
        this.Nombres = Nombres;
    }
}
```

```java
package CLA_tramil;

public class Bodeguero extends Personas {

  public Integer Oficina;

    public Bodeguero(Integer Oficina,
            String Identificacion, String Apellidos, String Nombres) {
        super(Identificacion, Apellidos, Nombres);
        this.Oficina = Oficina;
    }
```

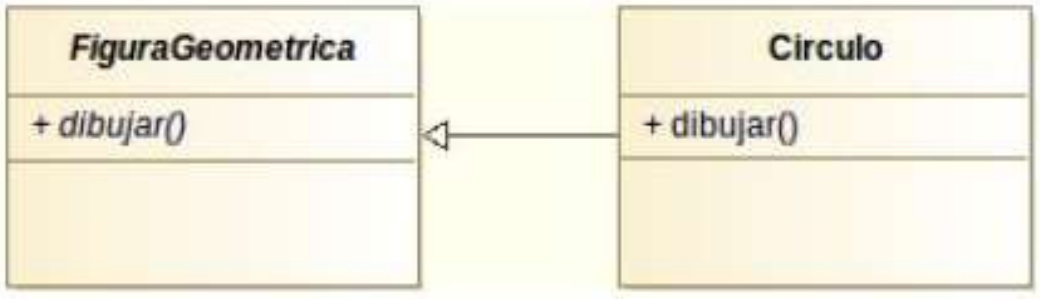
4. Generalización

- Métodos abstractos
  - En Java, una clase padre puede tener métodos abstractos, lo que significa que para esos métodos no se proporciona ninguna implementación.
  - Una clase abstracta es una clase que tiene al menos un método abstracto. Al tener uno o más métodos sin implementación, las clases abstractas no se pueden instanciar.
  - Una clase que extiende a una clase abstracta debe implementar los métodos abstractos (escribir el código) o bien volverlos a declarar como abstractos, con lo que ella misma se convierte también en clase abstracta.

4. Generalización

– En UML, tanto las operaciones como las clases abstractas se expresan poniendo el nombre de la operación o la clase en cursiva.

- Ejemplo clase abstracta



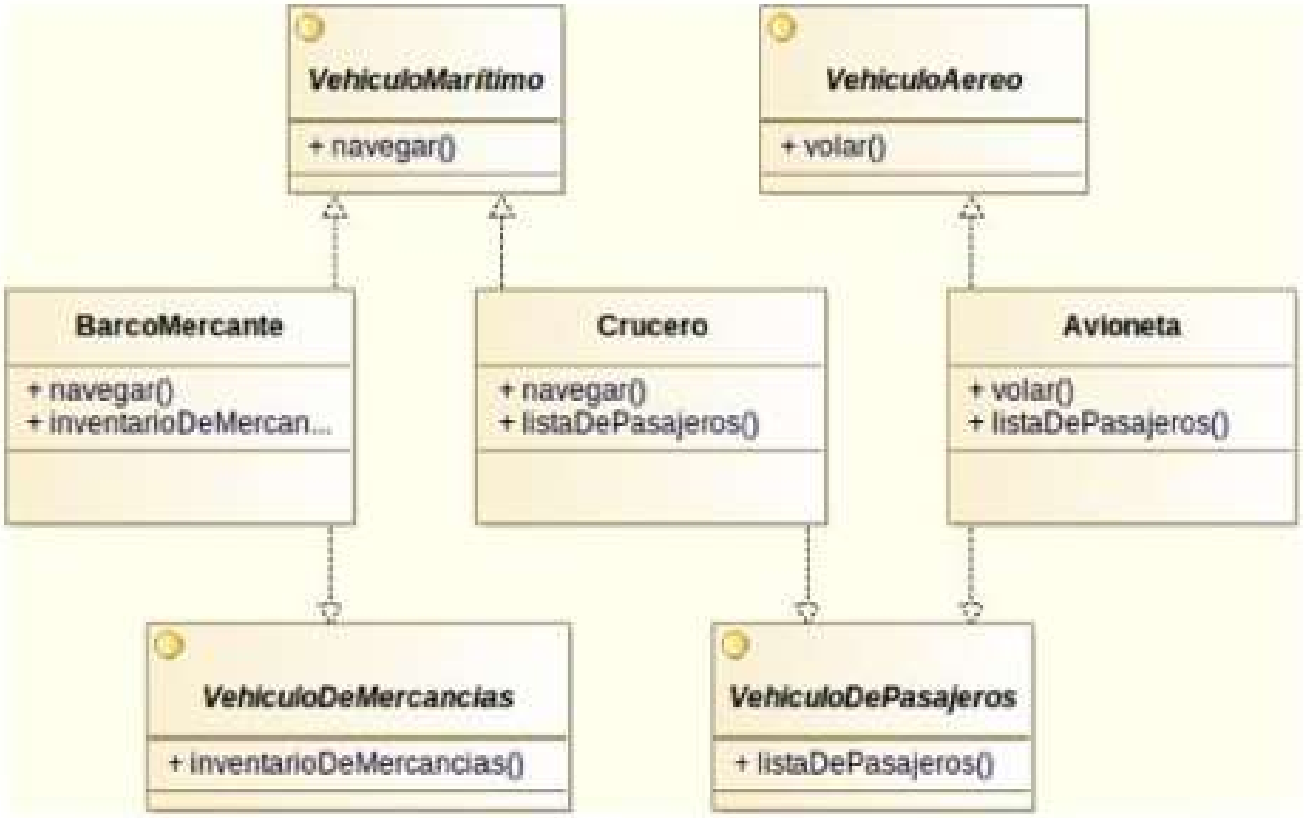
Interfaces

- ¿Clases abstractas o interfaces?
  - En Java, ¿qué diferencia hay entre un interface y una clase que tiene todos sus métodos abstractos?
  - En primer lugar, se diferencian en el modo como se definen y se utilizan:
  - Un interfaz puede también contener variables, pero siempre static y final. Por el contrario, una clase abstracta pura sí que puede tener atributos de instancia.

Interfaces

- **¿Generalización o realización?**
- ¿Qué diferencia hay entre la generalización (herencia) y la realización (interfaces)?
  - La diferencia más importante es que una clase puede implementar múltiples interfaces, pero sólo puede heredar de (como máximo) una clase padre.
  - Por ejemplo, en el siguiente diagrama tenemos los interfaces VehículoMarítimo, VehículoAéreo, VehículoDeMercancías y VehículoDePasajeros.

Ejemplo


VehiculoMaritimo
+ navegar()
VehiculoAereo
+ volar()
BarcoMercante
+ navegar()
+ inventarioDeMercan...
Crucero
+ navegar()
+ listaDePasajeros()
Avioneta
+ volar()
+ listaDePasajeros()
VehiculoDeMercancias
+ inventarioDeMercancias()
VehiculoDePasajeros
+ listaDePasajeros()

Interfaces

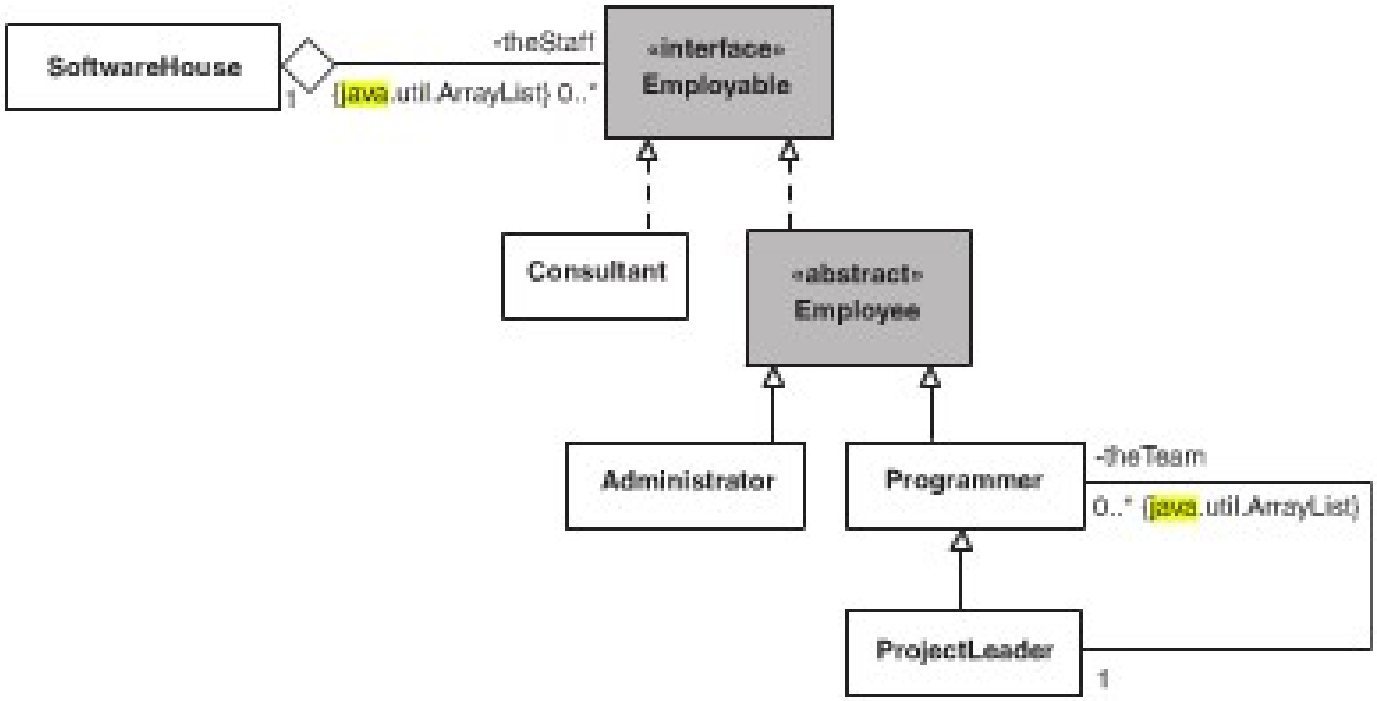


Figure 5.8 *Class diagram with an* ***interface***

Interfaces

- Colección de métodos abstractos y propiedades constantes
- Las clases implementan la interface
- No se puede instanciar una interface
- No contiene constructor
- Interfaces sirven para lograr polimorfismo
- Una clase que implemente la interface debe implementar todos los métodos de la interface
  - Debe tener mismo nombre y firma
- Clase con más de una interface
- Interface – implements
- Abstract - extends

Enum Types

- Un Enum Type es un tipo de datos especial que permite que una variable sea un conjunto de constantes predefinidas. La variable debe ser igual a uno de los valores que han sido predefinidos para él.

Genéricos

- List<?>
- ArrayList<??>
- Map<K,V>

Operaciones

- Add
- Remove
- Get
- Put
- Size

Dependencia

- Según el Manual de Referencia de UML, una dependencia indica una relación semántica entre dos o más clases del modelo. La relación puede ser entre las clases propiamente dichas, es decir, no tiene por qué involucrar a instancias de las clases (como en la asociación, composición y agregación).

Dependencia

- Si ClaseA depende de ClaseB, significa que un cambio en ClaseB provoca un cambio en el significado de ClaseA.
- Según esa definición, todas las relaciones vistas hasta ahora (asociación, agregación, composición, generalización y realización) serían dependencias, pero por tener significados muy específicos se les ha dado nombres concretos. Por tanto, se podría decir que las dependencias son todas aquellas relaciones que no encajan en ninguna de las categorías anteriores.

Interfaces

- Las interfaces en Java nos solucionan en parte la no existencia de la herencia múltiple; aumentando así las posibilidades de polimorfismo en la herencia múltiple sin los problemas que esta conlleva.
- Son un tipo de clase especial que no implementa ninguno de sus métodos. Todos son abstractos, por tanto no se pueden instanciar.
- Los métodos se definen como abstractos. Cuyo objetivo es forzar una interfaz (API) pero no una implementación.

Interfaces

- De las interfaces también se hereda, aunque se suele decir se implementa.
- Una clase puede heredar de múltiples interfaces.
- Una clase puede heredar de otra clase y a la vez heredar de múltiples interfaces.
- Un interfaz puede definir también constantes
- Si una clase que hereda de un interfaz, no implementa todos los métodos de éste, deberá ser definida como abstracta.

Interfaces

Ejemplo

```
public interface Mascota
{
  public abstract void jugar();
  public abstract void vacunar();
}

public class Perro extends Canino implements Mascota
{
  public void comer() { … }

  public void hacerRuido() { … }

  public void rugir() { … }

  public void jugar() { … }

  public void vacunar() { … }
}
```