

# Relaciones entre Clases

---

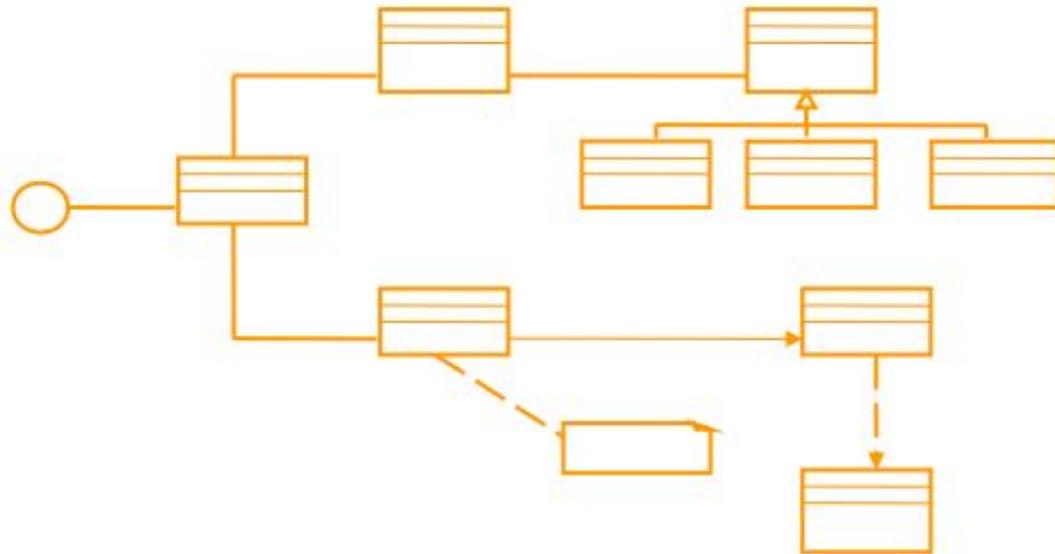
UNJu - Programación Orientada a Objetos



# Interfaz funcional

---

- Un diagrama de clases muestra las clases del sistema y sus relaciones. Describe la vista estática del sistema





# Visibilidad de atributos y operaciones de una clase

---

- La visibilidad puede ser:
  - Público (+): objetos de cualquier clase.
  - Protegido (#): objetos de las subclases.
  - De paquete (~): objetos de cualquier clase del mismo paquete.
  - Privado (-): objetos donde está definido el atributo u operación.

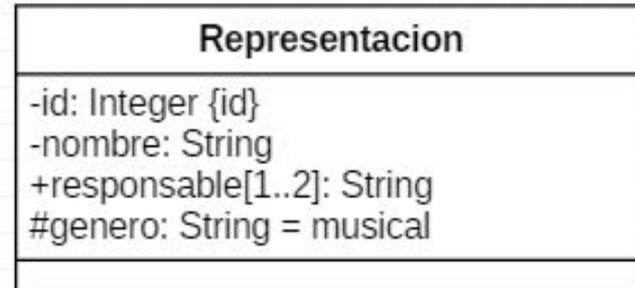
<b>MODIFICADOR</b>	<b>CLASE</b>	<b>PACKAGE</b>	<b>SUBCLASE</b>	<b>TODOS</b>
<b>public</b>	Sí	Sí	Sí	Sí
<b>protected</b>	Sí	Sí	Sí	No
<b>No especificado</b>	Sí	Sí	No	No
<b>private</b>	Sí	No	No	No



# Clase: atributo

---

- [visibilidad] nombre [multiplicidad][:tipo][=valor][{lista de propiedades}]
- Ejemplos





# Clase: operación

---

- Sintaxis
- [visibilidad] nombre[(lista de parámetros)][:tipo de retorno][[propiedades]]
- Declaración de un parámetro:
- [dirección]nombre: tipo[multiplicidad][=valor]

Donde dirección puede ser:

- **in:** parámetro de entrada
- **out:** parámetro de salida
- **Inout:** parámetro de entrada y salida.



# Diagrama de clases: relaciones

---

## ¿Qué son las relaciones entre clases?

- Es un a **conexión semántica** entre objetos. Proveen un camino de comunicación entre ellos.

## ¿Qué tipos de relaciones existen?

- Asociación
- Generalización
- Dependencia
- Realización



# Diagrama de clases: relaciones

---

## Notación

- Asociación 
- Agregación 
- Composición 
- Generalización 
- Dependencia 
- Realización 



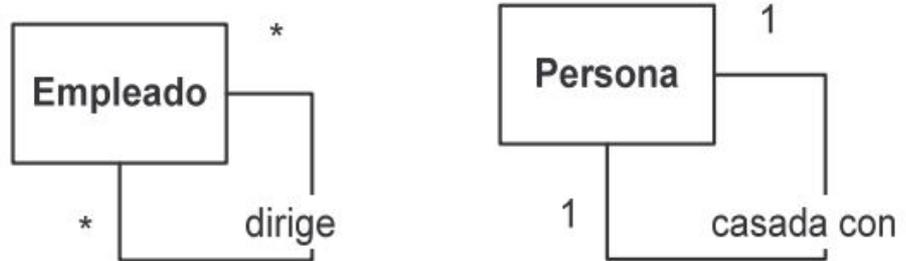
# Asociación

---

- Es una relación estructural que especifica que los objetos de un elemento están conectados con los otros de otro.



- Relaciones involutivas





# Asociación Involutiva

```
public class Empleado {  
    private Integer id;  
    private String nombre;  
    private Empleado responsable;  
    public Integer getId() {  
        return id;  
    }  
}
```

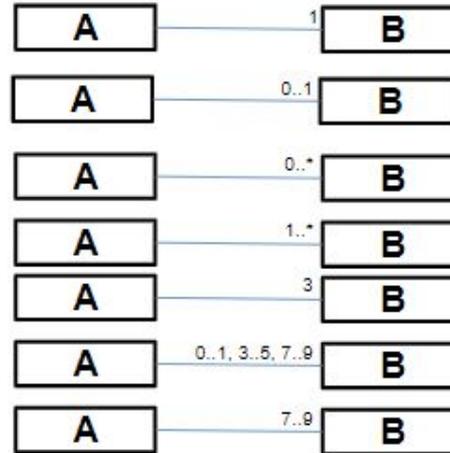
id	nombre	id_responsable
1	A	
2	B	1
3	C	1
4	D	2
5	E	2



# Multiplicidad

Indica cuántos objetos pueden conectarse a través de una instancia de una asociación:

- Exactamente uno
- Cero o uno
- Muchos:
- Uno o más:
- Número exacto
- Lista
- Rango

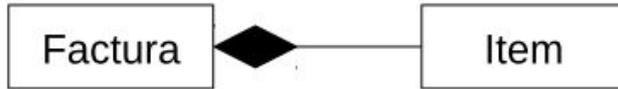




# Relaciones de conocimiento

---

- Compuesto-de 
- La relación que existe entre dos partes es tal que:
  - Una de las partes “contiene” a la otra.
  - La parte contenida no puede existir sin la parte contenedora

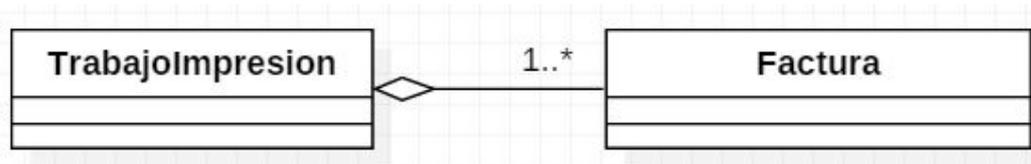


- ¿Puede existir un ítem si factura?
- ¿Puede existir una factura por un monto X sin ítems?
- Pueden implicar nuevas responsabilidades para la parte contenedora (cascada):
  - Creación
  - Destrucción
  - Rectificación de datos (como fechas)



# Relaciones de conocimiento

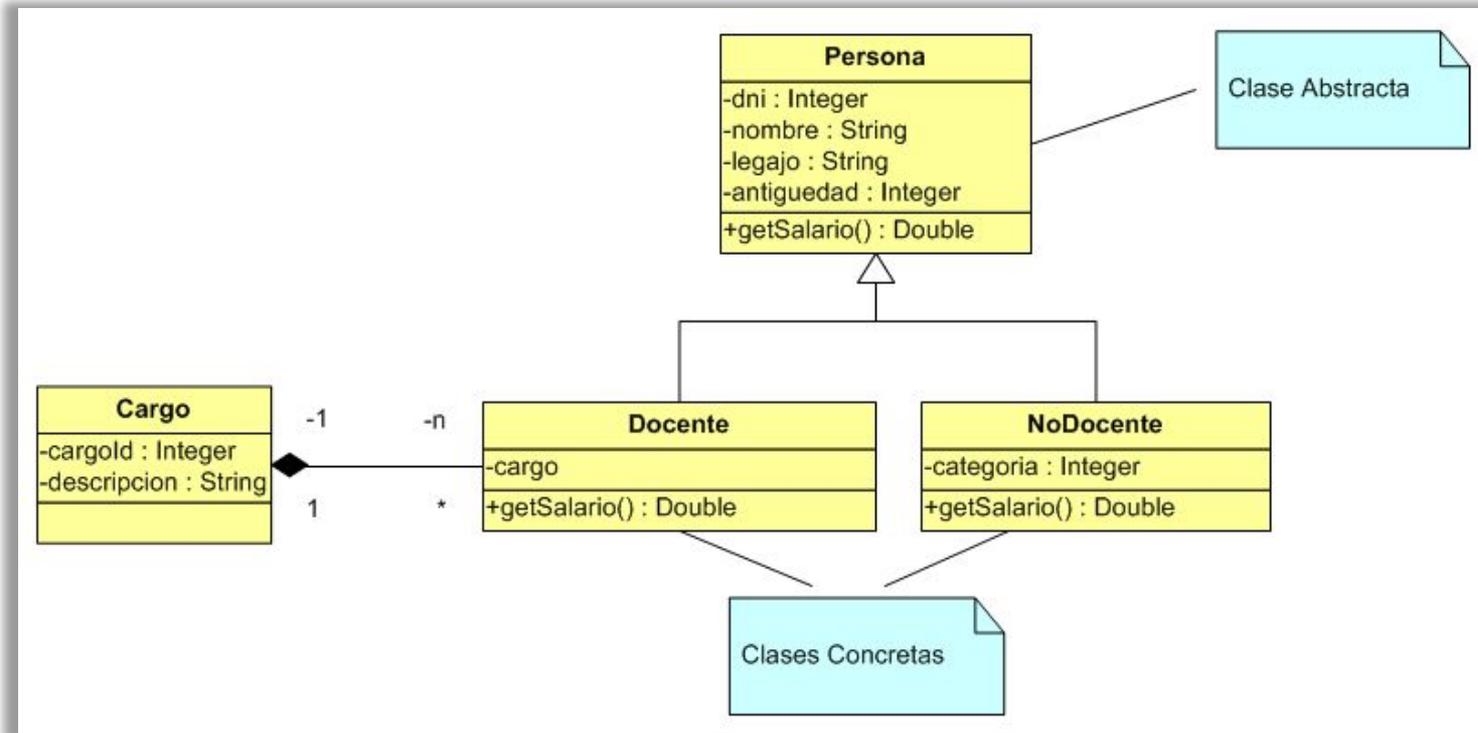
- Es-parte-de 
- La relación que existe entre dos partes es tal que:
  - Una de las partes “tiene” a la otra.
  - La parte “**tenida**” puede existir sin la parte poseedora



- ¿Puede existir una Factura sin **TrabajoImpresion**?
- ¿Puede existir un **TrabajoImpresion** sin Factura?



# Asociaciones: Ejemplo





# Implementación

```
/**
 * Representa los cargos docentes disponibles
 * @author José Zapana
 */
public class Cargo {
    private int cargoId;
    private String descripcion;

    private List<Docente> docentes = new ArrayList<Docente>();

    public Cargo(int cargoId, String descripcion) {
        this.cargoId = cargoId;
        this.descripcion = descripcion;
    }
}
```

Collection de  
docentes  
asociados a un  
cargo

Instancia de  
la Clase  
cargo

```
/**
 * Representa al personal docente de la Universidad
 *
 * @author José Zapana
 */
public class Docente extends Persona {

    private Cargo cargo;
    public static final double ADICIONAL_DOCENTE = 2000;
}
```



# Herencia

---

- Define una relación de clases, en la que una clase comparte los **comportamientos** y la **estructura** de datos de la otra
- Constituye una técnica valiosa, porque posibilita y fomenta la reutilización del software



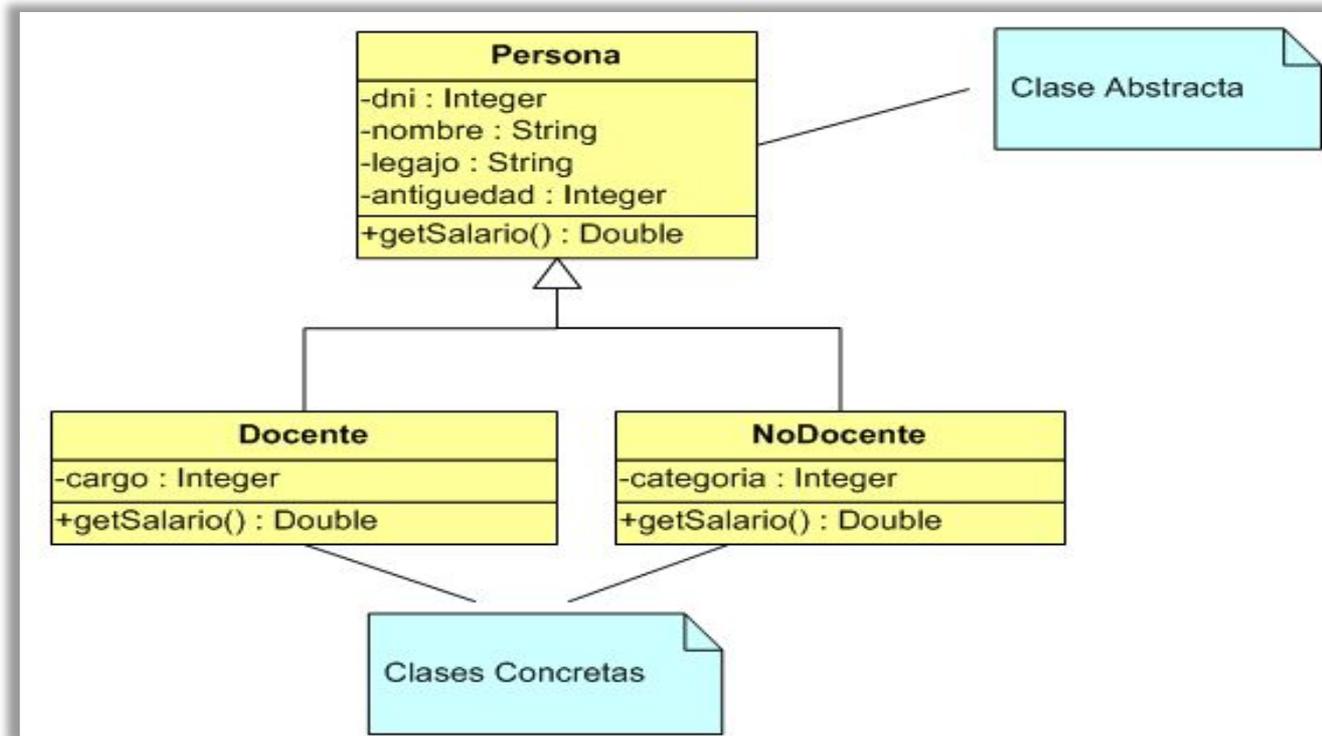
# Ventajas

---

- Modelado de la realidad: las relaciones de especialización/generalización entre las entidades del mundo real.
- Evitar redundancias
- Facilitar la reutilización
- Sirve de soporte al polimorfismo



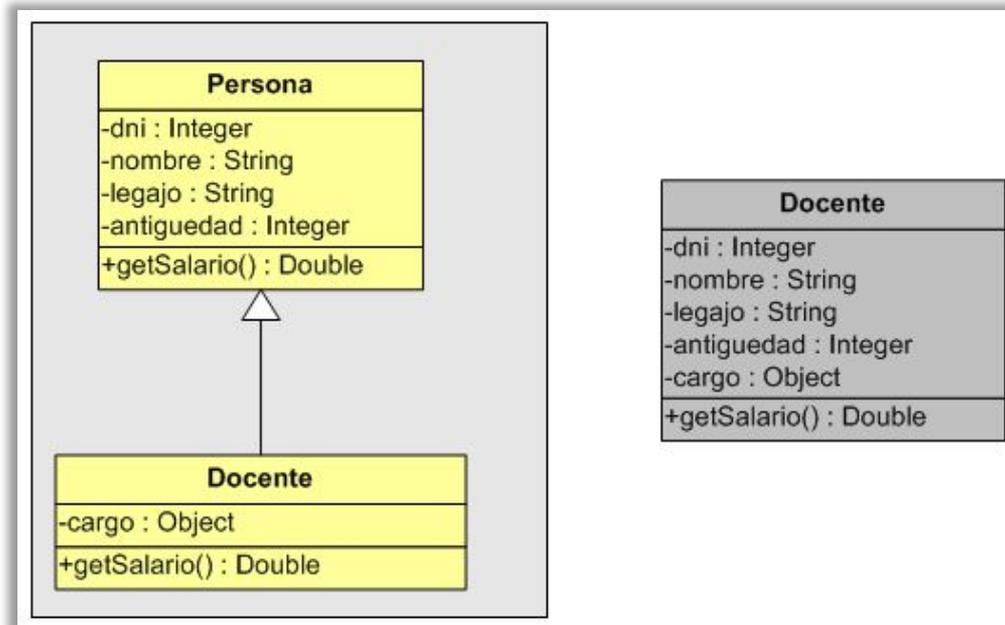
# Esquema de Herencia





# Aspecto de una clase

Una subclase hereda todas las variables de instancia de su superclase





# Uso de **super()** en constructores

```
/**
 * Superclase abstracta
 * @author José Zapana
 */
public abstract class Persona {
    private int dni;
    private String nombre;
    private String legajo;
    private int antiguedad;
    private static final double SUELDO_BASICO = 1000;

    public Persona(int dni, String nombre, String legajo, int antiguedad) {
        this.dni = dni;
        this.nombre = nombre;
        this.legajo = legajo;
        this.antiguedad = antiguedad;
    }
}
```

```
/**
 * Representa al personal docente de la Universidad
 *
 * @author José Zapana
 */
public class Docente extends Persona {

    private Cargo cargo;
    public static final double ADICIONAL_DOCENTE = 2000;

    public Docente(Cargo cargo, int dni, String nombre, String legajo, int antiguedad) {
        super(dni, nombre, legajo, antiguedad);
        this.cargo = cargo;
    }
}
```





# Especificación de Métodos Adicionales

---

- La superclase define métodos para todos los tipos de Persona.
- La subclase puede especificar métodos adicionales que específicos de Docente

```
/**
 * Representa al personal docente de la Universidad
 *
 * @author José Zapana
 */
public class Docente extends Persona {

    private Cargo cargo;
    public static final double ADICIONAL_DOCENTE = 2000;

    public Docente(Cargo cargo, int dni, String nombre, String legajo, int antiguedad) {
        super(dni, nombre, legajo, antiguedad);
        this.cargo = cargo;
    }

    /**
     * Devuelve el Dni con los puntos. Ej: 31.458.321
     * @return
     */
    public String getFormattDni() {
        return getDniConPuntos(getDni());
    }
}
```



# Sustitución de Métodos de Superclase

- Una subclase hereda todos los métodos de su superclase.
- La subclase puede sustituir un método por su propia versión especializada

```
/**
 * Superclase abstracta
 * @author José Zapana
 */
public abstract class Persona {
    private int dni;
    private String nombre;
    private String legajo;
    private int antiguedad;
    private static final double SUELDO_BASICO = 1000;
```

```
/**
 * Sueldo Básico común a todo e
 * @return
 */
public double getSalario(){
    return SUELDO_BASICO;
};
```



```
public class Docente extends Persona {

    private Cargo cargo;
    public static final double ADICIONAL_DOCENTE = 2000;

    /**
     * Calculo del salario del Docente
     *
     * @return
     */
    @Override
    public double getSalario() {
        //Ejemplo de Calculo
        System.out.println("Calculando Sueldo del Docente...");
        return (super.getSalario() + ADICIONAL_DOCENTE) * getAntiguedad();
    }
}
```





# Consideraciones Importantes

---

- La herencia sólo se debe utilizar para relaciones “es un tipo de” reales:
  - Debe ser siempre posible sustituir un objeto de subclase por uno de superclase.
  - Todos los métodos de la superclase deben tener sentido en la subclase
- El uso de la herencia como solución a corto plazo provoca problemas en el futuro.