



Tecnologías de Programación

Paradigma Orientado a Objetos

II – UML



UML

DIAGRAMAS EN UML

a) Diagramas 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, de uso y de contenido. Estos diagramas son los más comunes del modelado de sistemas orientados a objetos. Direccionan la vista de diseño estático del sistema. Un diagrama de clases esta compuesto por los siguientes elementos:

- **Clase:** atributos, métodos y visibilidad.
- **Relaciones:** Herencia, Composición, Agregación, Asociación y Uso.



UML – Relaciones entre Clases

- Las relaciones existentes entre las distintas clases nos indican cómo se comunican los objetos de esas clases entre sí:

Los mensajes “navegan” por las relaciones existentes entre las distintas clases.

- Tipos de Relaciones:
 - Asociación (conexión entre clases)
 - Dependencia (relación de uso)
 - Generalización/especialización (relaciones de herencia)

● Cardinalidad – Multiplicidad

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: $0..*$ ($0..n$)

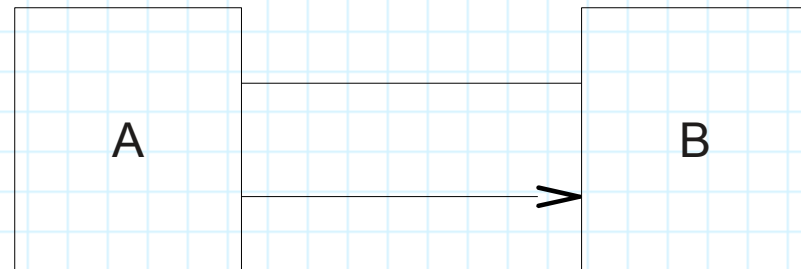
-número fijo: m (m denota el número).

● Navegabilidad

Indica como se puede navegar la relación. Puede ser en una u otra dirección o bidireccional. Se especifica con una flecha sobre la relación.

Bidireccional

De A a B

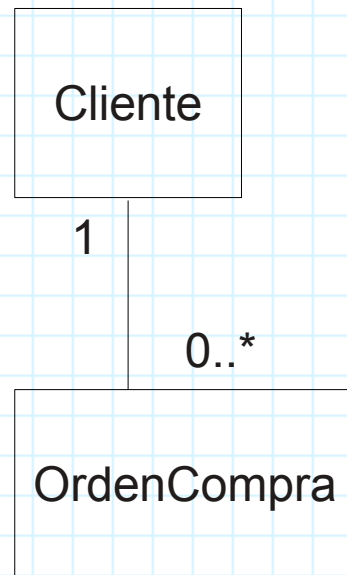


Asociación:

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

Ejemplo:

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

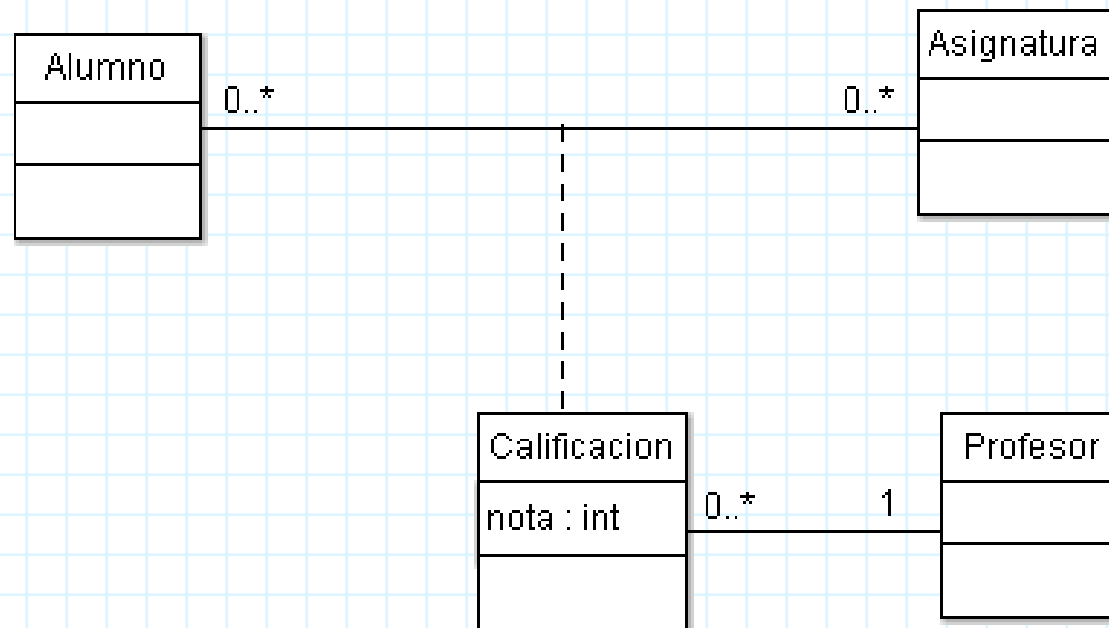



```
public class Cliente {  
    private Collection<OrdenCompra> cOrdenes;  
}  
public class OrdenCompra {  
    private Cliente oCliente;  
}
```

Clases de Asociación

En una asociación entre dos clases, la propia asociación puede tener atributos.

Las **clases de asociación** permiten añadir atributos, operaciones y otras características a las asociaciones.





```
import java.util.Vector;
public class Alumno {

    /**
     * @element-type Asignatura
     */
    private Vector vCalificaciones;

}
```

```
import java.util.Vector;
public class Asignatura {

    /**
     * @element-type Alumno
     */
    private Vector vCalificaciones;

}
```

```
public class Calificacion {

    // Atributos
    private int nota;
    private Profesor oProfesor;
    private Alumno oAlumno;
    private Asignatura oAsignatura;

    // Constructor
    public Calificacion( Alumno oAlumno,
                        Asignatura oAsignatura,
                        int _nota
                        Profesor oProfesor) {

        this.nota = nota;
        this.oAlumno = oAlumno;
        this.oAsignatura = oAsignatura;
        this.oProfesor = oProfesor;

    }

}
```

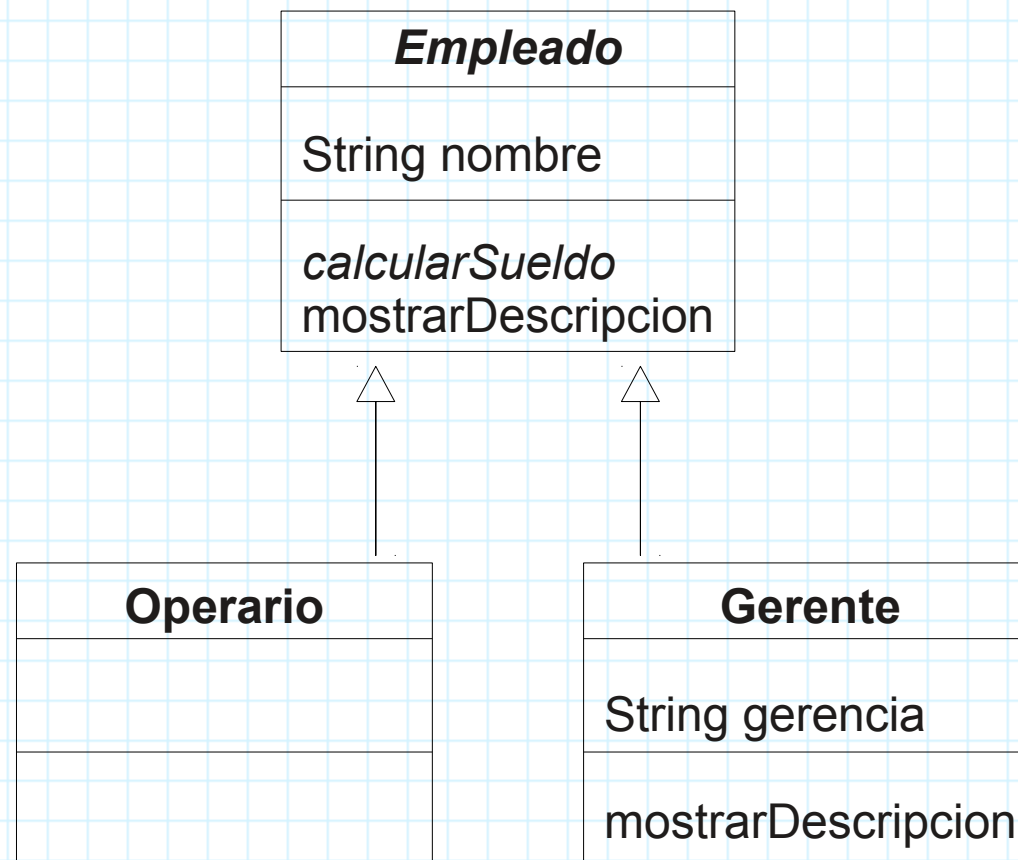

RELACIONES ENTRE CLASES:

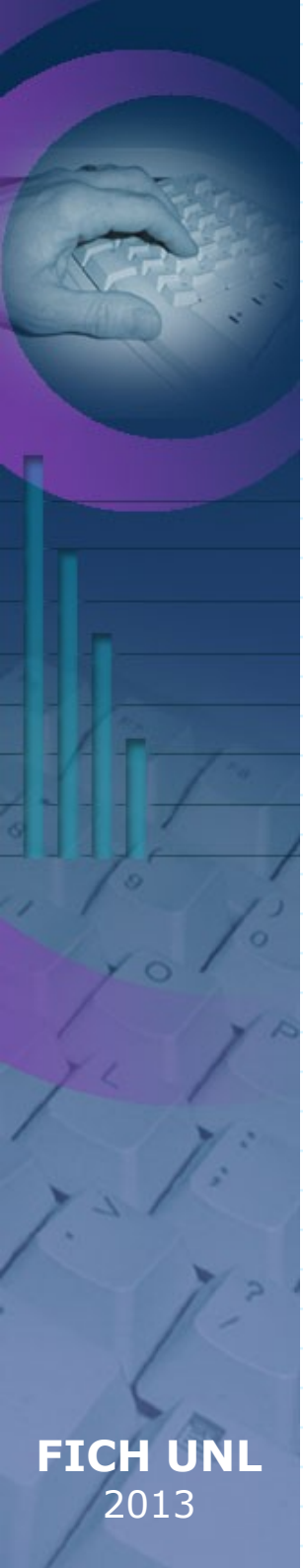
Herencia (Especialización/Generalización):

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).

Responde a la pregunta “es un/a”. Un Operario “es un” Empleado.

Ejemplo:





```
public class Empleado {  
    private String nombre;  
  
    public Empleado(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public float calcularSueldo() {  
        return 0;  
    }  
  
    public void mostrarDescripcion() {  
        System.out.println(this.nombre);  
    }  
}
```

```
public class Operario extends Empleado{  
    public Operario(String nombre) {  
        super(nombre);  
    }  
}
```

```
public class Gerente extends Empleado {  
    private String gerencia;  
  
    public Gerente(String nombre, String gerencia) {  
        super(nombre);  
        this.gerencia = gerencia;  
    }  
  
    public void mostrarDescripcion() {  
        System.out.print(this.gerencia + "-");  
        super.mostrarDescripcion();  
    }  
}
```



Características de la herencia

- **Anulación o sustitución:** cuando redefino un Método heredado en la subclase, se dice que estoy anulando o sustituyendo dicho método. Sería deseable una "**herencia selectiva**": seleccionar lo que se requiere heredar es la mejor forma de anulación.
- **Sobrecarga:** Propiedad que puede darse también sin herencia. Es designar varios elementos (identificadores) con el mismo nombre. No es anulación.
- **Polimorfismo** (sobrecarga con anulación) es la forma de invocar a distintos métodos utilizando el mismo elemento de programa.

Polimorfismo

Polimorfismo es invocar métodos distintos con el mismo mensaje (ligadura en tiempo de ejecución). Para ello es necesaria una jerarquía de herencia: una clase base que contenga un método polimórfico, que es redefinido en las clases derivadas (no anulado).

Se permite que los métodos de los hijos puedan ser invocados mediante un mensaje que se envía al padre.

Este tipo de clase que se usa para implementar el polimorfismo se conoce como **clase abstracta**.

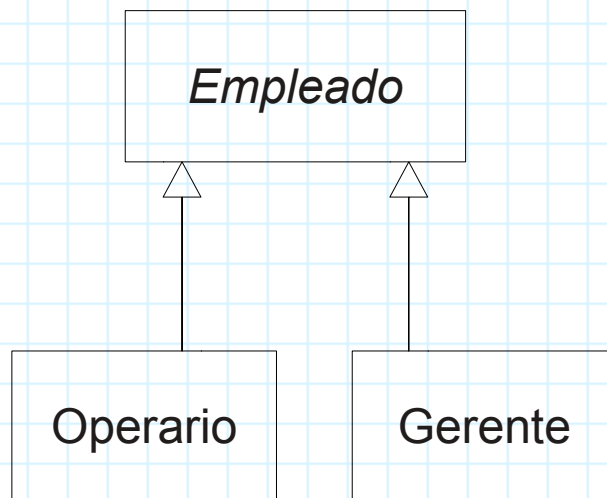
CASOS PARTICULARES:

Clase Abstracta:

Una clase abstracta se denota con el nombre de la clase y de los métodos con letra "itálica".

Esto indica que la clase definida no puede ser instanciada pues posee métodos abstractos (aún no han sido definidos, es decir, sin implementación).

La única forma de utilizarla es definiendo subclases, que implementan los métodos abstractos definidos.

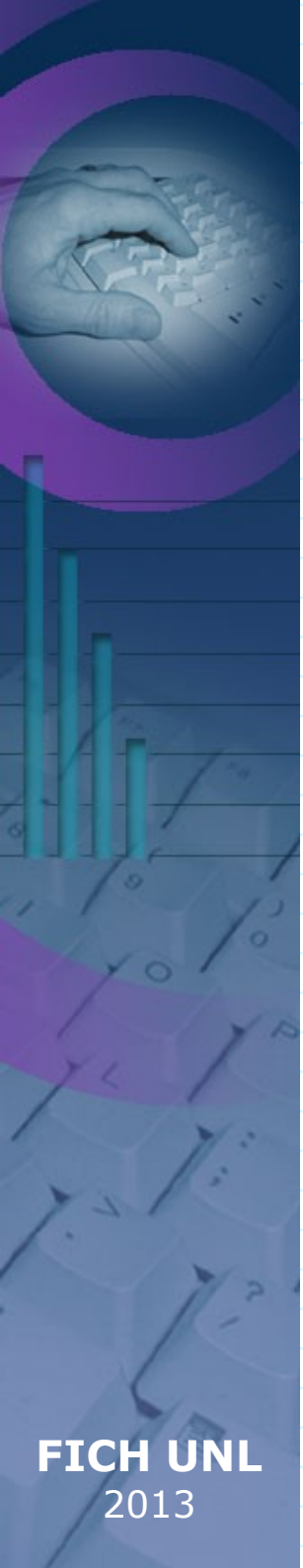


```
public abstract class Empleado {

    // Constructor
    public Empleado() {}
}

public class Operario extends Empleado{

    // Constructor
    public Operario() {
        super();
    }
}
```



```
public abstract class Empleado {  
  
    private String nombre;  
    public final static float basico = 100;  
  
    public Empleado(String nombre) {this.nombre = nombre;}  
  
    public abstract float calcularSueldo();  
  
    public void mostrarDescripcion() {  
        System.out.println(this.nombre);  
    }  
}
```

```
import java.util.ArrayList;  
import java.util.Collection;  
  
public class Empresa {  
  
    private String nombre;  
    private Collection<Empleado> cEmpleados;  
  
    public Empresa(String _nombre) {  
        nombre = _nombre;  
        cEmpleados = new ArrayList<Empleado>();  
    }  
  
    public String mostrarEmpresa() {return this.nombre.toUpperCase();}  
  
    public void agregarEmpleado(Empleado obj) {this.cEmpleados.add(obj);}  
  
    public void listarEmpleados() {  
        for (Empleado oEmpleado: cEmpleados) {  
            oEmpleado.mostrarDescripcion();  
        }  
    }  
}
```

Interfaces:

Una Interface no es una clase, no se puede instanciar. Una interface es implementada por una clase.

Contiene un conjunto de métodos sin su implementación. También puede contener atributos constantes.

Surgen de la necesidad de agrupar funcionalidades de los objetos.

Cuando un objeto implementa una interface esta obligado a codificar todos los métodos incluidos en esta.

Sirve para establecer un “protocolo” entre clases.

Genera métodos polimórficos.

```
public interface Imprimible {  
  
    public String imprimir();  
  
}
```

```
public class Operario extends Empleado  
    implements Imprimible {  
  
    public Operario(String nombre) {  
        super(nombre);  
    }  
  
    public String imprimit() {  
        return "Op." +  
            this.mostrarDescripcion();  
    }  
  
}
```

```
public class Impresora {  
  
    public Impresora() {  
        // Inicializo la impresora  
    }  
  
    public void imprimir(Imprimible oImp) {  
  
        // método polimórfico imprimir  
        oImp.imprimir();  
  
    }  
  
}
```