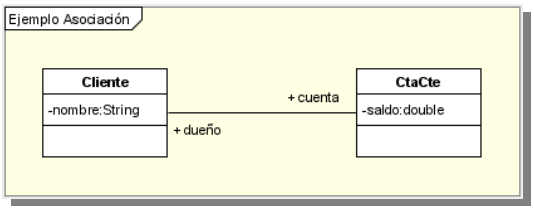
Clases UML a Código Java

Asociación

- Bidireccional con multiplicidad 0..1 o 1

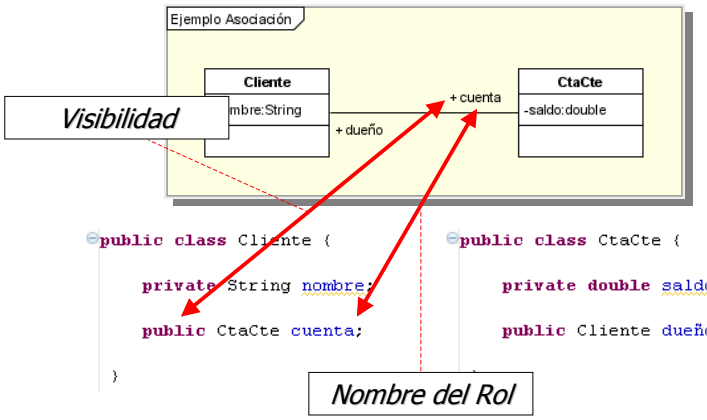



```
public class Cliente {

    private String nombre;

    public CtaCte cuenta;

}
```

```
public class CtaCte {

    private double saldo;

    public Cliente dueño;

}
```

Asociación

- Bidireccional con multiplicidad 0..1 o 1

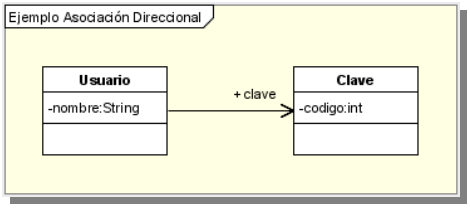



```
public class Cliente {
    private String nombre;
    public CtaCte cuenta;
}
```

```
public class CtaCte {
    private double saldo;
    public Cliente dueño;
}
```

Asociación

- Direccional con multiplicidad 0..1 o 1

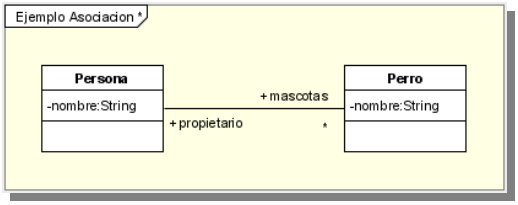



```
public class Usuario {
    private String nombre;
    public Clave clave;
}
```

```
public class Clave {
    private int codigo;
}
```

Asociación

- Bidireccional con multiplicidad *

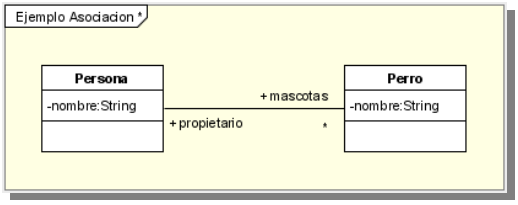



```
public class Persona {

    private String nombre;

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

}
```

```
public class Perro {

    private String nombre;

    public Persona propietario;

}
```

Asociación

- Bidireccional con multiplicidad *



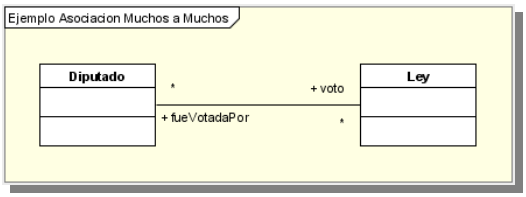

```
public class Persona {

    private String nombre;

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

}
```

```
public class Perro {

    private String nombre;

    public Persona propietario;

}
```

Decisión de Implementación

Asociación

- Bidireccional con multiplicidad *

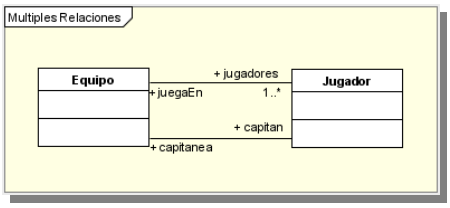



```
public class Diputado {

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

}
```

```
public class Ley {

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

}
```

Asociación

- ¿Con más de una relación?

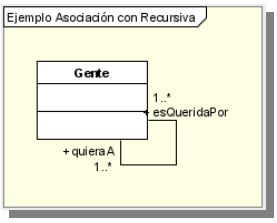



```
public class Equipo {

    public Jugador capitan;

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

}
```

```
public class Jugador {

    public Equipo capitanea;

    public Equipo juegaEn;

}
```

Asociación

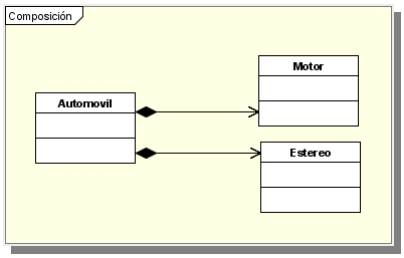
- ¿Y con esto?




```
public class Gente {

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

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

}
```

Composición

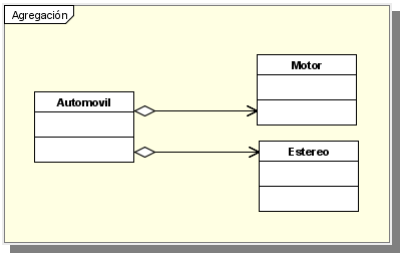
- Hay una dependencia en los ciclos de vida




```
public class Automovil {

    public Estereo estereo;
    public Motor motor;

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

}
```

Agregación

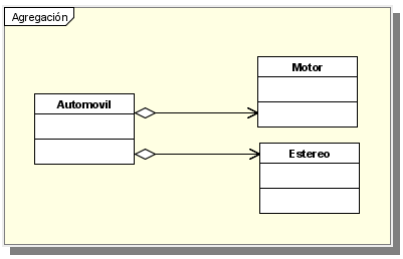
- Algo suena extraño...




```
public class Automovil {

    public Estereo estereo;
    public Motor motor;

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

}
```

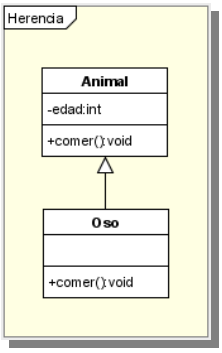
Agregación

- Algo con más sentido...




```
public class Automovil {

    public Estereo estereo;
    public Motor motor;

    public Automovil() {
    }

    public void ensamblar(Estereo e, Motor m) {
        estereo = e;
        motor = m;
    }

}
```

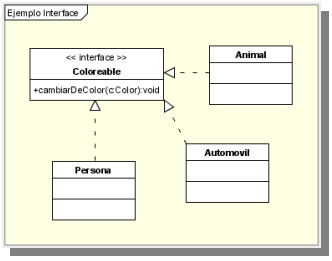
Herencia

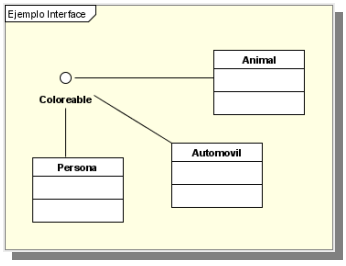



```
public class Animal {

    private int edad;

    public void comer() {
        // your code here
    }

}

public class Oso extends Animal {

    public void comer() {
        // Para el oso significa otra cosa...
    }

}
```

Según el lenguaje, puede ser necesario hacer explícito el override

Interface





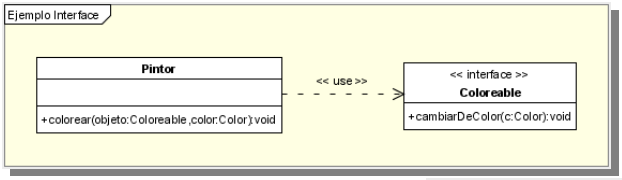

```
public interface Coloreable {

    public void cambiarDeColor(Color c);
}
```

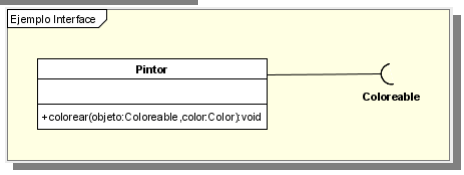
```
public class Persona implements Coloreable {

    public void cambiarDeColor(Color c) {
        // Se debe implementar
    }
}
```

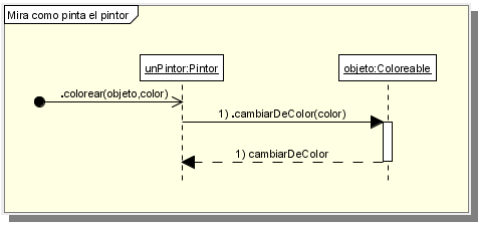
```
public class Automovil implements Coloreable {

    public void cambiarDeColor(Color c) {
        // Se debe implementar
    }
}
```

```
public class Animal implements Coloreable {

    public void cambiarDeColor(Color c) {
        // Se debe implementar
    }
}
```

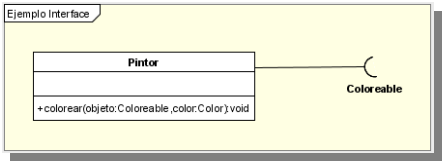
Interface

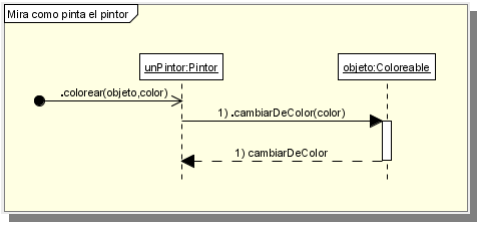






Interface

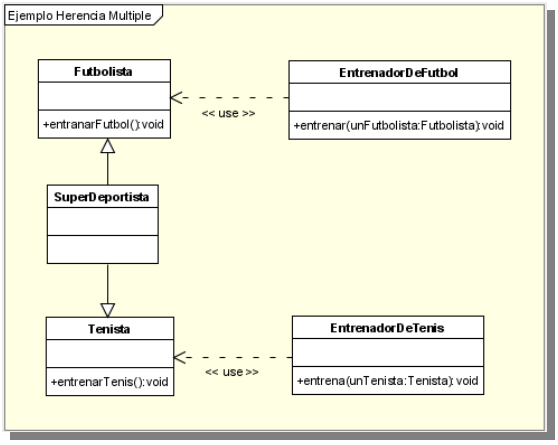





```
public class Pintor {

    public void colorear(Coloreable objeto, Color color) {
        objeto.cambiarDeColor(color);
    }
}
```

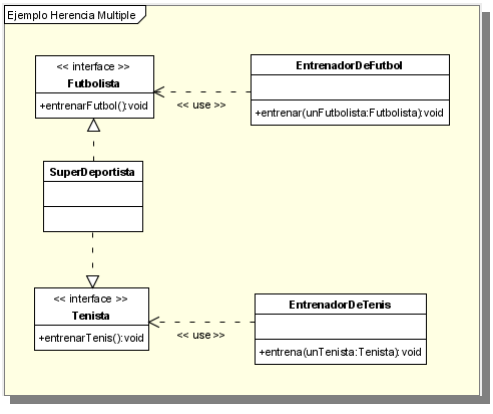
```
Persona unaPersona = new Persona();
Automovil unAutomovil = new Automovil();
Animal unAnimal = new Animal();
Color rojo = new Color();
Pintor unPintor = new Pintor();

unPintor.colorear((Coloreable) unaPersona, rojo);
unPintor.colorear((Coloreable) unAnimal, rojo);
unPintor.colorear((Coloreable) unAutomovil, rojo);
```

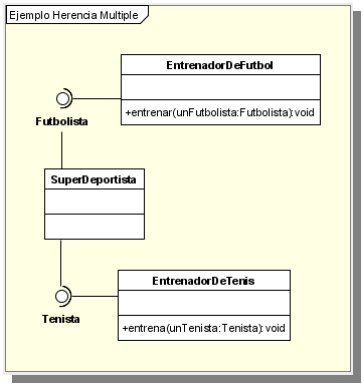
Herencia Multiple



Uno esperaría que el SuperDeportista pueda ser entrenado por cualquiera de los entrenadores

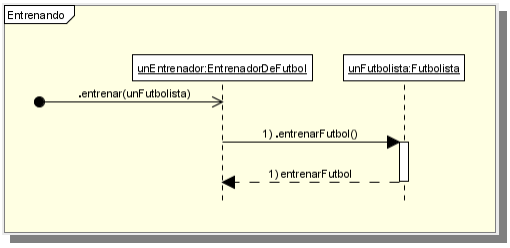
Herencia Multiple: Alternativa

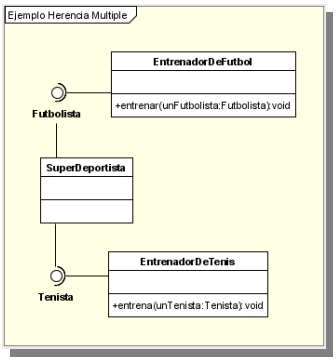




Ahora, cada entrenador "puede ver al futbolista o al tenista" detrás del superdeportista

Herencia Multiple: Alternativa






```
SuperDeportista unSuperDeportista = new SuperDeportista();
EntrenadorDeFutbol unEntrenadorDeFutbol = new EntrenadorDeFutbol();
EntrenadorDeTenis unEntrenadorDeTenis = new EntrenadorDeTenis();

unEntrenadorDeFutbol.entrenar((Futbolista) unSuperDeportista);
unEntrenadorDeTenis.entrenar((Tenista) unSuperDeportista);
```

Sobre secuencias

- Un diagrama de secuencia, debería ser "fácilmente" traducible a pseudo código. Además, contribuye a encontrar agujeros en el diseño.