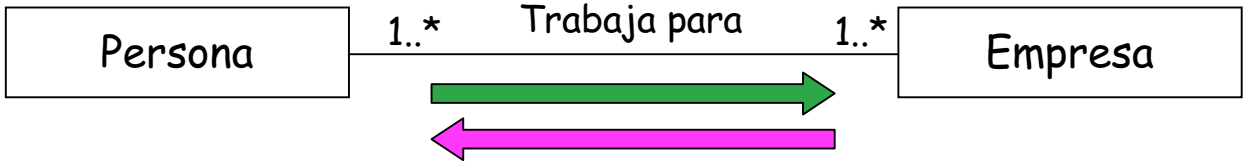
Introducción a relaciones Java

- A continuación se mostrará algunas reglas prácticas para transformar las relaciones de diagramas UML de clases como los vistos hasta ahora en código Java.
- Las reglas no se justifican aunque pueden considerarse como decisiones de "sentido común"

Navegabilidad

- Informalmente, cuando se traduce un diagrama de clases en una aplicación utilizando un lenguaje de poo (por ej. Java), se tiene que decidir los accesos necesarios a través de las relaciones entre las clases (sólo aplicable a relaciones entre 2 clases)

- **Ejemplo:** si se tiene el siguiente diagrama de clases UML

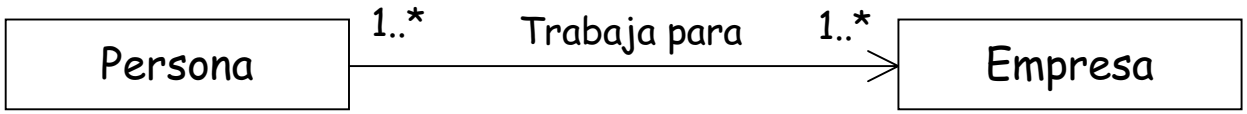


  - A la hora de codificar una aplicación Java basada en él hay que decidir cuáles de los siguientes "accesos" son necesarios:
    - Si será necesario conseguir **las empresas en las que trabaja una persona determinada**.
    - Si será necesario conseguir **los empleados que trabajan en una determinada empresa**

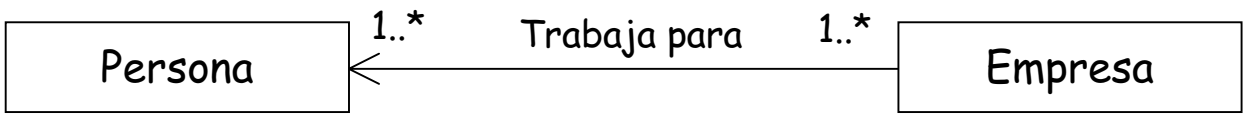
- La navegabilidad hace referencia a esta situación.

Navegabilidad (II)

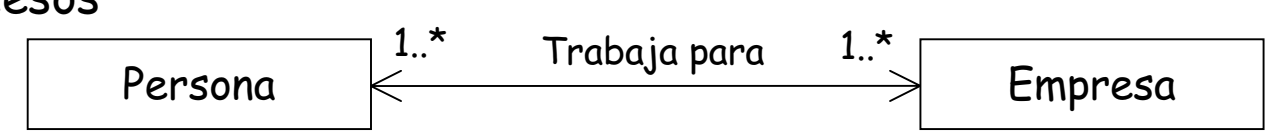
- Representación en UML
  - La siguiente figura muestra la representación de la navegabilidad en el primer sentido (flecha verde en el gráfico anterior), es decir, sólo interesa conseguir, a partir de una persona concreta, las empresas en las que trabaja.



  - La siguiente figura muestra la representación de la navegabilidad en el segundo sentido (flecha rosa en el gráfico anterior), es decir, sólo interesa conseguir, a partir de una empresa concreta, las personas que trabajan en ella.



  - En la siguiente figura se muestra la necesidad de codificar los dos accesos

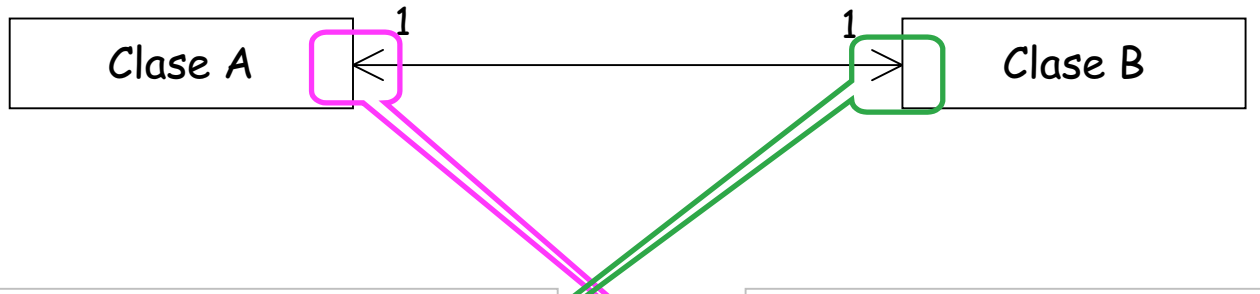


Transformación diagrama → código

- Observaciones
  - Se presentará a continuación una clasificación de casos en los que se puede agrupar esta transformación.
  - Los casos están determinados
    - Por la navegabilidad
    - Por el tipo de relación
      - De composición.
      - No de composición (de agregación o de asociación)
    - Por las multiplicidades
    - Por la existencia de clases de asociación
  - Los casos analizados son los siguientes
    - No de composición, multiplicidades 1-1
    - No de composición, multiplicidades 1-n (exactamente n)
    - No de composición, multiplicidades 1-0..*
    - No de composición, multiplicidades (máximas) *-*
    - De composición, multiplicidades 1-1
    - De composición, multiplicidades 1-n (exactamente n)
    - De composición, multiplicidades 1-0..*

Relaciones no de composición: 1-1 (I)

- En cualquiera de los dos extremos que la navegabilidad determine, se añade una referencia al objeto relacionado.
- La transformación puede resumirse gráficamente de la siguiente manera

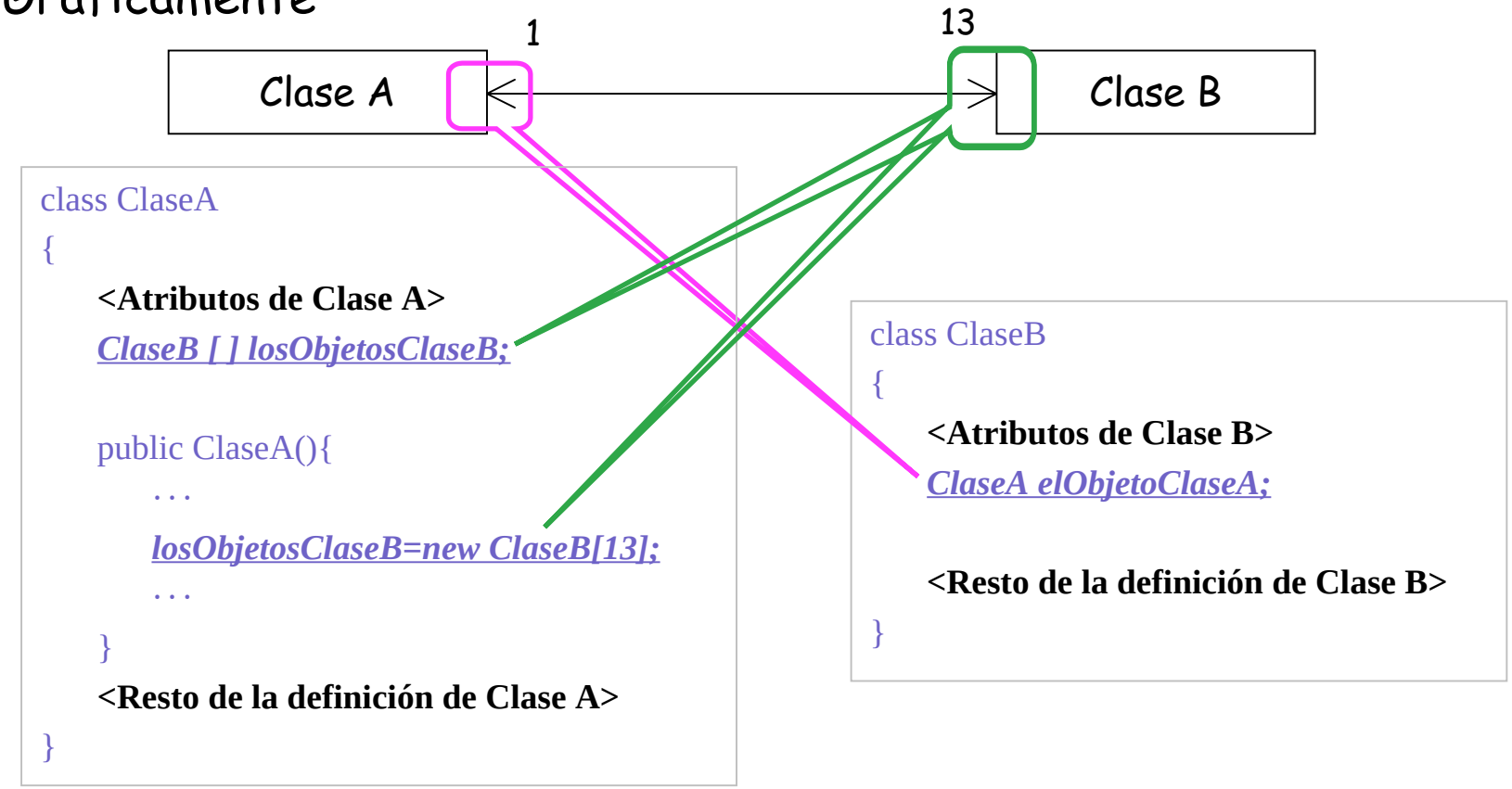



```
class ClaseA
{
    <Atributos de Clase A>
    ClaseB elObjetoClaseB;

    <Resto de la definición de Clase A>
}
```

```
class ClaseB
{
    <Atributos de Clase B>
    ClaseA elObjetoClaseA;

    <Resto de la definición de Clase B>
}
```

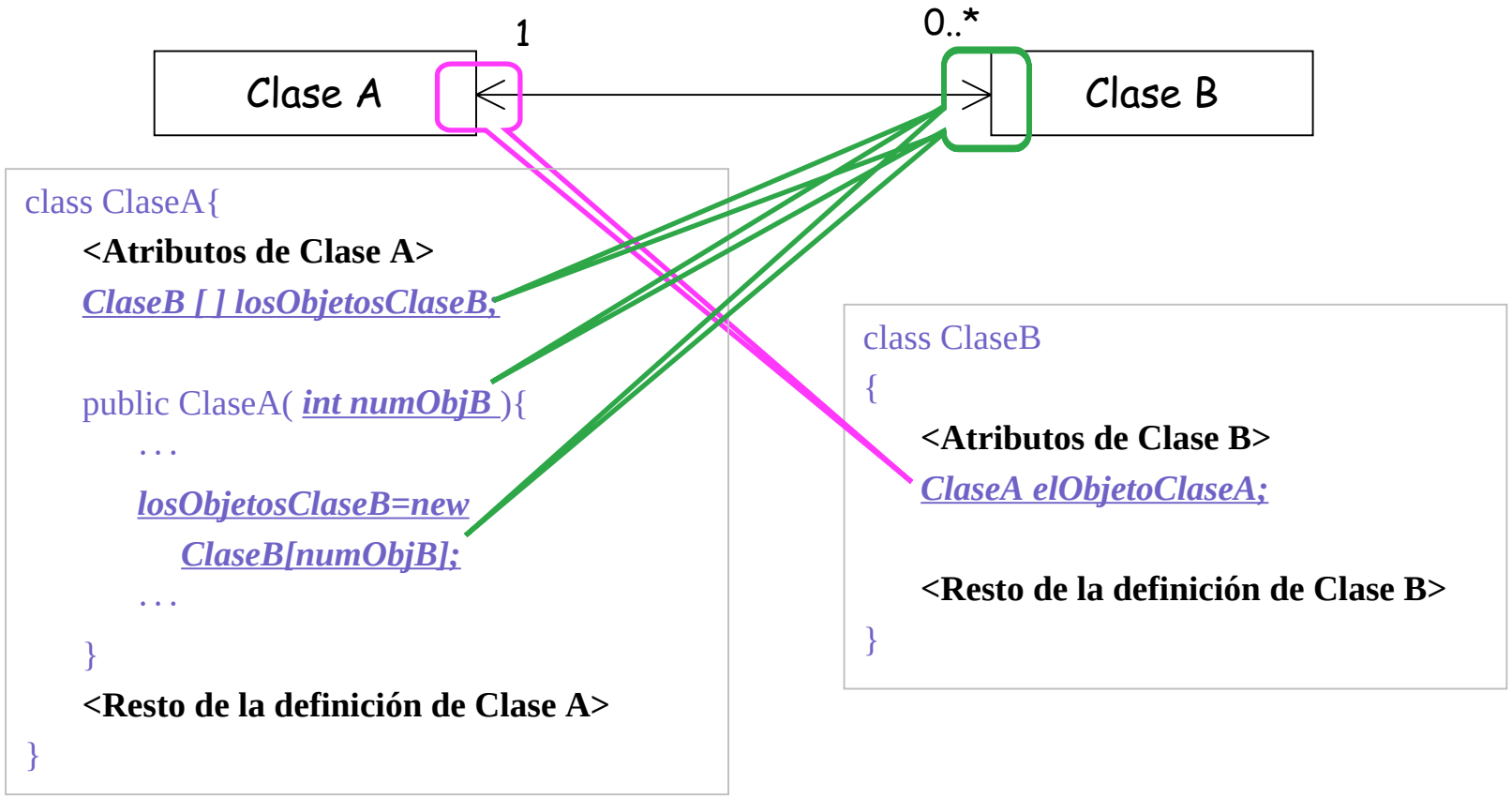
Relaciones no de composición: 1-n (I)

- Aunque se muestre los dos extremos sólo es nuevo el de multiplicidad exactamente n (en este caso 13): se tiene que guardar los 13 objetos relacionados en cualquier colección (aquí array) que se inicializa en el constructor de la otra clase
- Gráficamente



Relaciones no de composición: 1-0..* (I)

- Parecido al caso anterior. Como no se sabe en principio cuántos objetos de un tipo están relacionados con cada uno del otro, el constructor de la otra clase recibe ese número como parámetro
- Gráficamente



Relaciones no de composición: n-m (I)

- Este caso no aporta ninguna novedad pues puede reducirse a dos extremos de tipo n o 0..* de los analizados anteriormente.

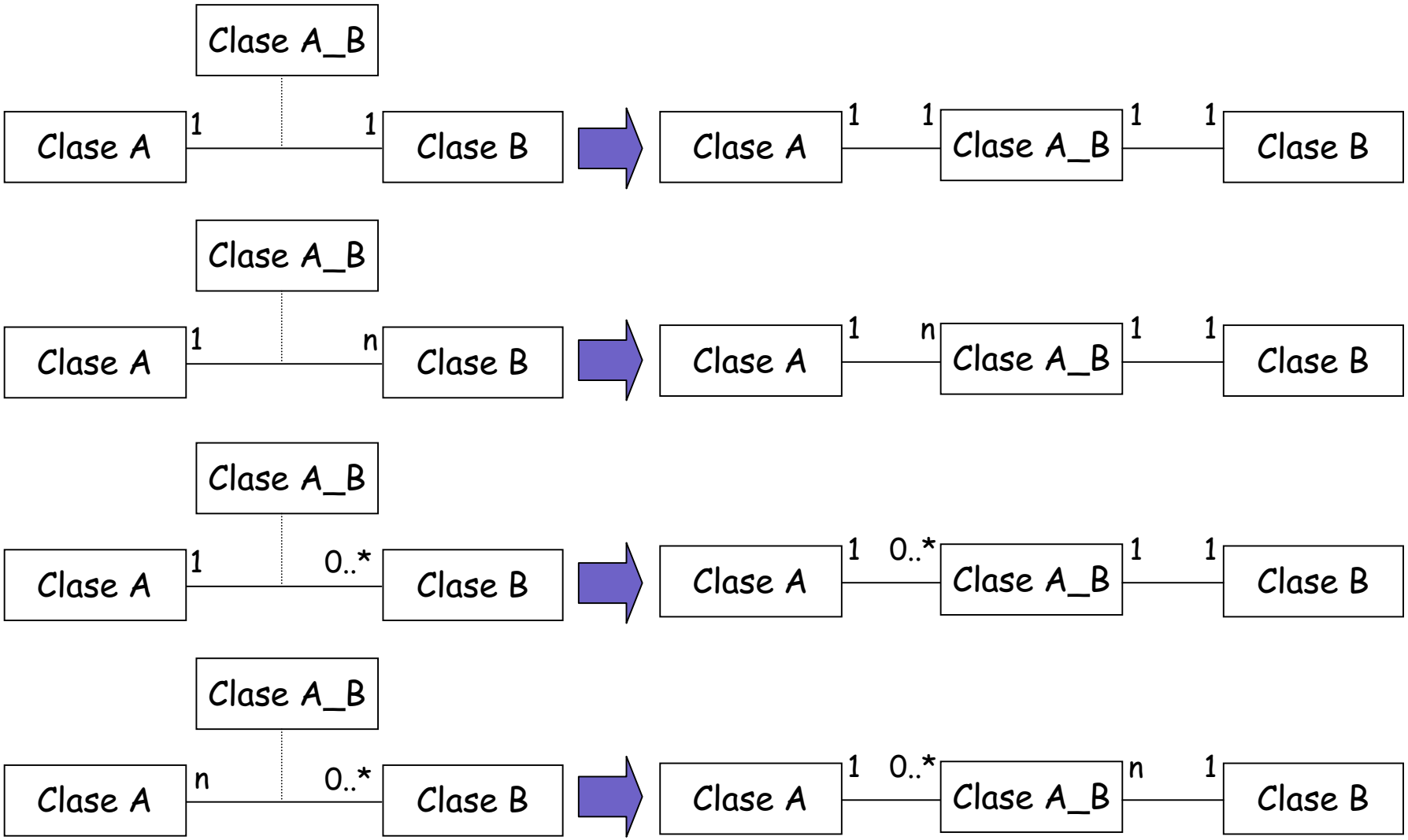

```
class ClaseA{
    <Atributos de Clase A>
    ClaseB [ ] losObjetosClaseB;

    public ClaseA( int numObjB ){
        . . .
        losObjetosClaseB=new
            ClaseB[numObjB];
        . . .
    }
    <Resto de la definición de Clase A>
}
```

```
class ClaseB
{
    <Atributos de Clase B>
    ClaseA [ ] losObjetosClaseA;

    public ClaseB(){
        . . .
        losObjetosClaseA=new ClaseA[7];
        . . .
    }
    <Resto de la definición de Clase B>
}
```

El uso de clases de asociación (I)

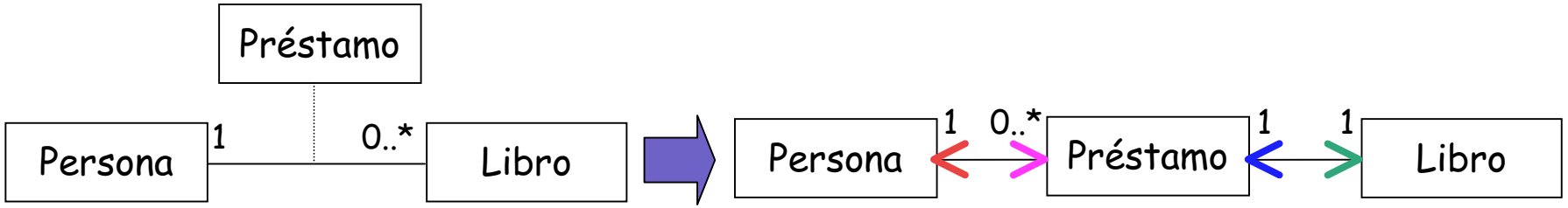
- Ya se explicó que las clases de asociación aparecen cuando la asociación tiene atributos propios.
- Es fácil imaginar un diagrama de clases UML equivalente al de partida y que se puede codificar con las reglas sugeridas en las transparencias anteriores
- A continuación se muestran esas transformaciones, la clase de asociación (que en los gráficos se llama siempre ClaseA_B) se trataría como cualquiera de las clases que han aparecido anteriormente.

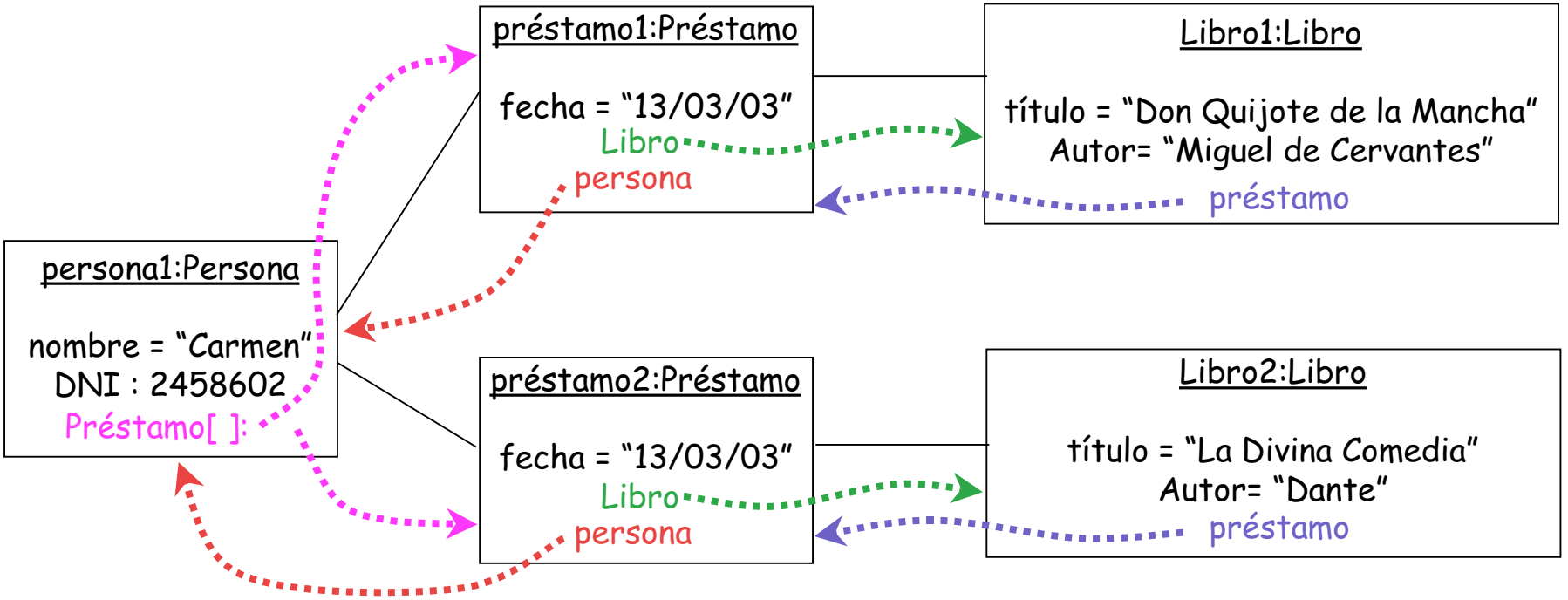
El uso de clases de asociación (II)



El uso de clases de asociación (II)

- Veamos con detalle uno de los casos.



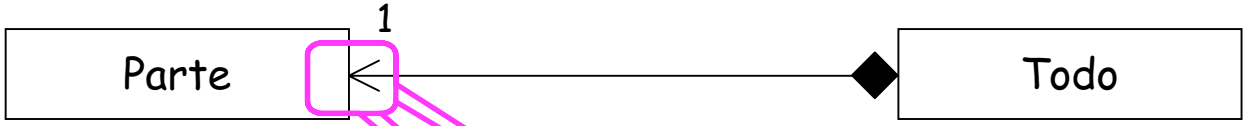


Relaciones de no composición: ejemplo

- El caso de las relaciones de composición (para el que se aporta un ejemplo completo) es más complicado e incluye el de las de no composición. Se remite al alumno a ese ejemplo.

Relaciones composición: 1-1 (I)

- Sólo añade a las de no composición que la responsabilidad de la creación de las partes reside en el todo
- Es acumulativo (si hay más de una clase parte, sus referencias y construcción se acumula de la misma manera)

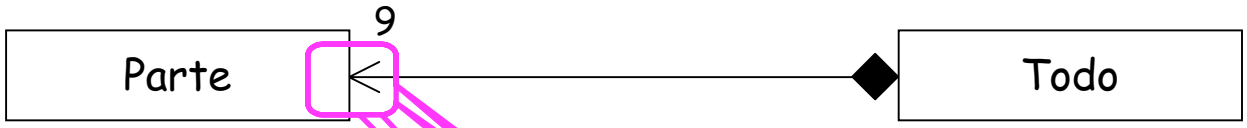



```
class Parte
{
    <Atributos de la clase Parte>

    public Parte(<args de construcción de Parte>)
    {
        <Código del constructor>
    }
    <Resto de la definición de la clase Parte>
}
```

```
class Todo{
    <Atributos de la clase Todo>
    Parte laParte;

    public Todo(<args de construcción de Parte>){
        laParte=new
                Parte(<args de construcción de Parte>);
    }

    <Resto de la definición de la clase Todo>
}
```

Relaciones composición: 1-n (I)

- Análogo al anterior. En el ejemplo se utiliza n=9

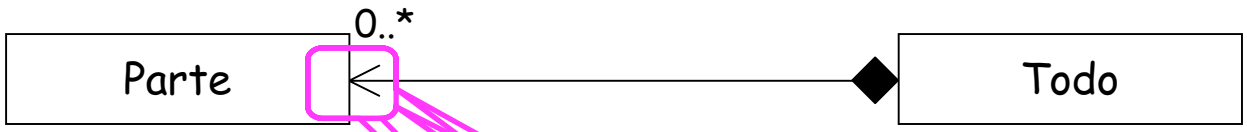



```
class Parte
{
    <Atributos de la clase Parte>

    public Parte(<args de construcción de Parte>)
    {
        <Código del constructor>
    }
    <Resto de la definición de la clase Parte>
}
```

```
class Todo{
    <Atributos de la clase Todo>
    Parte [ ] lasPartes ;

    public Todo(<args de construcción de las 9 Partes>){
      lasPartes=new Parte [9];
      for (int i=0; i<9; i++)
        lasPartes[i] (<args de construcción de Parte i>);
    }

    <Resto de la definición de la clase Todo>
}
```

Relaciones composición: 1-0..* (I)

- Análogo al anterior. Al constructor de la clase Todo hay que proporcionarle, además de los parámetros de construcción de todas su partes, el número de ellas.

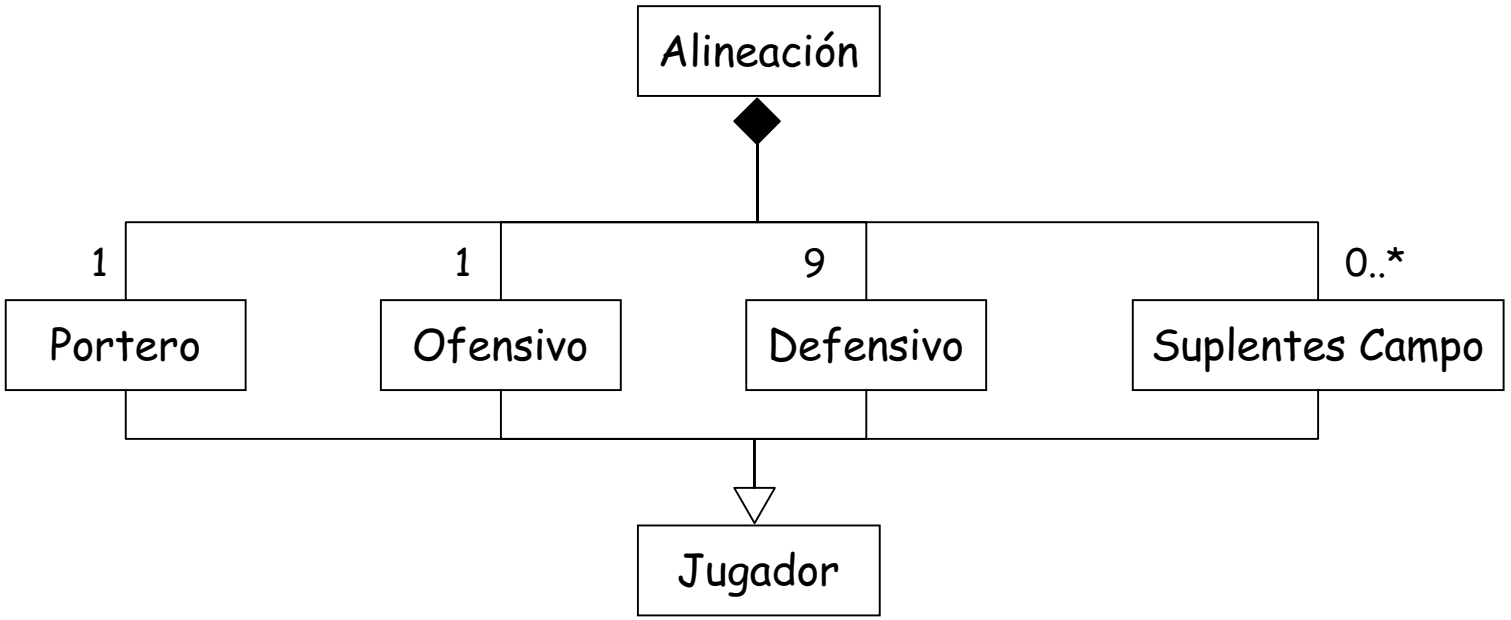



```
class Parte
{
    <Atributos de la clase Parte>

    public Parte(<args de construcción de Parte>)
    {
        <Código del constructor>
    }
    <Resto de la definición de la clase Parte>
}
```

```
class Todo{
    <Atributos de la clase Todo>
    Parte [ ] lasPartes ;

    public Todo(
            int numPartes ,
            <args de construcción de todas las Partes>){
        lasPartes=new Parte [numPartes];
        for (int i=0; i<numPartes; i++)
          lasPartes[i] (<args de construcción de Parte i>);
    }

    <Resto de la definición de la clase Todo>
}
```

Relaciones composición: ejemplo (I)

- **Ejemplo:** suponga el siguiente diagrama UML de clases cuya semántica se ha explicado como un ejercicio de clase



- A continuación se muestra una posible codificación Java para este diagrama de clases.

Relaciones composición: ejemplo (II)

- **Ejemplo: (cont.)**

```
class Jugador {
   protected String nombre;
   public String getNombre() {return nombre;}
}
```

Definición de la superclase Jugador (no aporta nada al ejemplo)

```
class Portero extends Jugador {
   public Portero(){}
   public Portero(String n) { nombre = n;}
}
```

Definición de la parte Portero

```
class Ofensivo extends Jugador {
   public Ofensivo(){}
   public Ofensivo(String n) { nombre = n;}
}
```

Definición de la parte Ofensivo

Relaciones composición: ejemplo (III)

- **Ejemplo: (cont.)**

```
class Defensivo extends Jugador {
   public Defensivo(){}
   public Defensivo(String n) {nombre = n;}
}
```

→ Definición de la parte Defensivo

```
class SuplenteCampo extends Jugador {

   public SuplenteCampo(){}
   public SuplenteCampo(String n) { nombre = n;}
}
```

→ Definición de la parte SuplenteCampo

Relaciones composición: ejemplo (IV)

- **Ejemplo: (cont.)**

```
class Alineacion {
```
→ Definición del todo

```
private Portero elPortero;
```
→ El único portero de la alineación

```
private Ofensivo elOfensivo;
```
→ El único jugador ofensivo

```
private Defensivo[] losDefensivos;
```
→ Array para guardar los 9 defensivos

```
private SuplenteCampo[] losSuplentesCampo;
```
→ Para guardar los posibles suplentes de campo

```
public Alineacion(){};
```
→ Constructor por defecto

Relaciones composición: ejemplo (V)

- **Ejemplo: (cont.)**

```
public Alineacion(   String nombrePortero,
                     String nombreOfensivo,
                     String [] nombresDefensivos,
                     int numero_suplentes_campo,
                     String [] nombresSuplementesCampo
                   )
{
   elPortero = new Portero( nombrePortero );
   elOfensivo = new Ofensivo(nombreOfensivo);
   losDefensivos = new Defensivo[9];
   for (int i = 0; i < 9; i++) {
      losDefensivos[i]=new Defensivo( nombresDefensivos[i]);
   }
   losSuplentesCampo = new
  SuplenteCampo[numero_suplentes_campo];
   for (int i =0; i < numero_suplentes_campo; i++) {
      losSuplentesCampo[i]=new
  SuplenteCampo(nombresSuplementesCampo[i]);
   }
}
```

Arg constructor de Portero

Arg constructor de Ofensivo

Args de los constr. de los 9 defensivos

Número de los suplentes

Args de constructor de todos los suplentes (son numero_suplentes_campo)

Creación del Portero

Creación del Ofensivo

Creación de los 9 Defensivo

Creación de los numero_suplentes_campo suplentes de campo

Relaciones composición: ejemplo (VI)

- **Ejemplo: (cont.)**

Método auxiliar

```
public void mostrarAlineacion() {
        System.out.print("BAJO LOS PALOS...");
        System.out.println(elPortero.getNombre());
        System.out.println();
        System.out.print("EN PUNTA...");
        System.out.println(elOfensivo.getNombre());
        System.out.println();
        System.out.println("EN EL RESTO DEL TERRENO:");
        for (int i=0; i < losDefensivos.length; i++) {
           System.out.println(losDefensivos[i].getNombre());
        }
        System.out.println();
        System.out.println("EN EL BANQUILLO:");
        for (int i=0; i < losSuplentesCampo.length; i++) {
           System.out.println(losSuplentesCampo[i].getNombre());
        }
    }
}
```

Relaciones composición: ejemplo (VII)

- **Ejemplo: (fin)**

Posible main de prueba

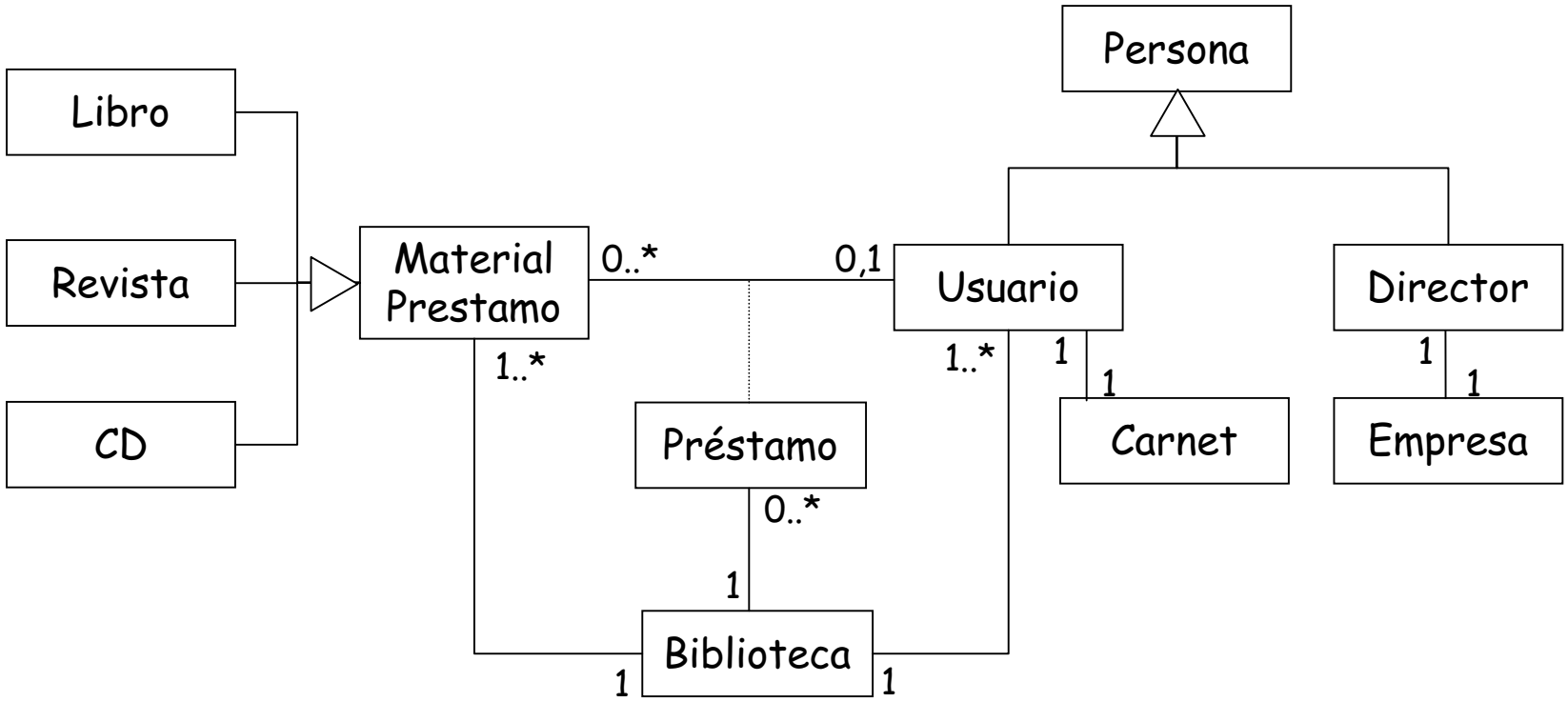
```
public class EjComposicion {

   public static void main(String [] args ) {
      Alineacion equipoTitular = null;
      String [] nombres_defensas = { "Juan Olivas", "Perico Lopez", "Luca D'an ti", "Roberto Locomotora Pepo",
                                     "Santos Romera", "Olivio Juantorenea", "Olivier Saint Perignisois",
                                     "Manu el loco Panti", "Salvador Perdido" };
      String [] nombres_suplentes_campo= {"Chavi el perro loco Santonja", "Richard Pichardo"};

      equipoTitular = new Alineacion(
                                     "Ramiro Man Oplas",
                                     ”Julian Motos",
                                     nombres_defensas,
                                     nombres_suplentes_campo.length,
                                     nombres_suplentes_campo);
      equipoTitular.mostrarAlineacion();
   }
}
```

Establecimiento de los enlaces

- Descripción del problema:
  - Una vez que se tiene las clases creadas con suficiente información como para implementar sus relaciones queda sólo, en el código de la aplicación, especificar cómo se establecen y utilizan los enlaces
  - Uno de los enlaces suele tener que establecerlo una clase ajena.
  - Para el otro se puede optar entre:
    - Que lo establezca una clase ajena
    - Que lo establezca directamente el objeto involucrado en el enlace anterior

Relaciones: ejemplo (I)

- **Ejemplo:** suponga el siguiente diagrama UML y que interesan todas las navegabilidades



- A continuación se muestra fragmentos de una posible aplicación Java para este diagrama de clases.

Relaciones : ejemplo (II)

- **Ejemplo: (cont.)** clase *MaterialPrestamo*

```
class MaterialPrestamo
{
   String titulo;
     . . .
   Prestamo el_prestamo;

     . . .

   public void realizarPrestamo(Prestamo p)
   {
      el_prestamo=p;
   }
}
```

→ De la relación entre MaterialPrestamo y Préstamo

→ Método para el establecimiento de este enlace

Relaciones : ejemplo (III)

- **Ejemplo: (cont.)** clases *Persona* y *Usuario*

```
class Persona{
   String nombre;
   int edad; }

class Usuario extends Persona
{
   ArrayList los_prestamos;
   Carnet su_carnet;
   Empresa su_empresa;

   public Usuario()
   {
      los_prestamos=new ArrayList();
   }
   public void realizarPrestamo(Prestamo p)
   {
      los_prestamos.add(p);
      p.material_prestado.realizarPrestamo(p);
   }
}
```

Clase persona

Clase Usuario

De la relación entre Usuario y Préstamo

De la relación entre Usuario y Préstamo

Establecimiento de enlaces entre Préstamo y MaterialPrestado (al dar de alta un préstamo) desde Usuario (clase implicada en el enlace)

Relaciones : ejemplo (IV)

- **Ejemplo: (cont.)** clase *Préstamo*

```
class Prestamo
{
   String fecha_inicio;
   MaterialPrestamo material_prestado;
   Usuario usuario_beneficiario;

     . . .
}
```

→ De la relación entre Préstamo, Usuario y MaterialPrestamo

Relaciones : ejemplo (V)

- **Ejemplo: (cont.)** clase *Biblioteca*

```
class Biblioteca{
   public ArrayList prestamos_bibliotecarios;
   public ArrayList usuarios_biblioteca;
   public ArrayList materiales_prestamo;

   public Biblioteca(){
      prestamos_bibliotecarios = new ArrayList();
      usuarios_biblioteca = new ArrayList();
      materiales_prestamo = new ArrayList();}

   public void altaUsuario(Usuario nuevo_usuario) {···}
   public void altaMaterialPrestamo(MaterialPrestamo nuevo_material){···}
   public MaterialPrestamo buscarMaterialPrestamo(MaterialPrestamo mp){···}
   public Usuario buscarUsuario(Usuario u)   {···}
   public ArrayList buscarPrestamosUsuario(Usuario u) {···}
```

De la relación entre Biblioteca y Préstamos

De la relación entre Biblioteca y Usuarios

De la relación entre Biblioteca y MaterialPrestamo

De la relación entre Biblioteca y Préstamos

De la relación entre Biblioteca y Usuarios

De la relación entre Biblioteca y MaterialPrestamo

Métodos para la funcionalidad de la aplicación

Relaciones : ejemplo (VI)

- **Ejemplo: (cont.)** clase *Biblioteca*, método *altaPrestamo*

```
public void altaPrestamo(Prestamo nuevo_prestamo) {
    ...
    aux_prestamo=new Prestamo(aux_usuario, aux_material);

    aux_material.el_prestamo=aux_prestamo;
    aux_usuario.los_prestamos.add(aux_prestamo);

    aux_usuario.realizarPrestamo(aux_prestamo);

    prestamos_bibliotecarios.add(aux_prestamo);
    ...
  }
}
```

Si es Biblioteca (clase ajena) la que establece los (dos) enlaces

Si es una de las clases involucradas en el enlace (Usuario) la que establece los enlaces

Relaciones : ejemplo (VII)

- **Ejemplo: (cont.)** clase *Biblioteca*, método *altaPrestamo*

```
public class GestionBiblioteca
{

   public static void main (String [] args){···}

}
```