Tema 3: Herencia en Java

Programación Orientada a Objetos

Curso 2009/2010

Begoña Moros Valle

DIS
Departamento de
Informática y Sistemas

Índice

- Introducción
- Herencia y creación
- Herencia y Ocultamiento de la Información
- Redefinición de características
- Polimorfismo
- Herencia y Sistema de tipos
- Ligadura dinámica
- Clase `Object`
- Genericidad y Herencia. Genericidad restringida
- Clases abstractas
- Interfaces
- Colecciones en Java (paquete `java.util`)
- Patrones de diseño:
  - Método plantilla, Patrón Estrategia y Patrón Composite
- Herencia Múltiple

Introducción

- Las clases no son suficientes para conseguir los objetivos de:

  (A) **REUTILIZACIÓN:** Necesidad de mecanismos para generar **código genérico**:

  - Capturar aspectos comunes en grupos de estructuras similares
  - Independencia de la representación
  - Variación en estructuras de datos y algoritmos

  (B) **EXTENSIBILIDAD:** Necesidad de mecanismos para favorecer:

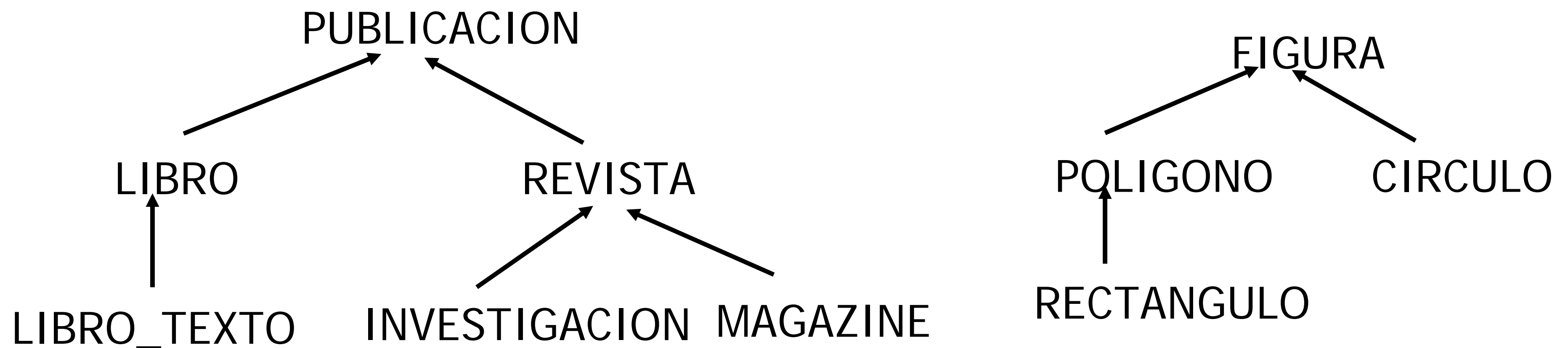
  - "Principio abierto-cerrado" y "Principio Elección Única"
  - Estructuras de datos polimórficas.

Introducción

- Entre algunas clases pueden existir relaciones conceptuales:
  - Extensión, Especialización, Combinación
- EJEMPLO:
  - Libros y Revistas tienen propiedades comunes
  - Una Pila puede definirse a partir de una Cola o viceversa
  - Un Rectángulo es una especialización de Polígono
  - Puede haber distintos tipos de Cuentas
- **¿Tiene sentido crear una clase a partir de otra?**
- La **herencia** es el mecanismo que:
  - sirve de soporte para registrar y utilizar las relaciones conceptuales existentes entre las clases
  - **posibilita la definición de una clase a partir de otra**

Jerarquías de herencia

- La herencia organiza las clases en una estructura jerárquica formando **jerarquías de clases**
- Ejemplos:

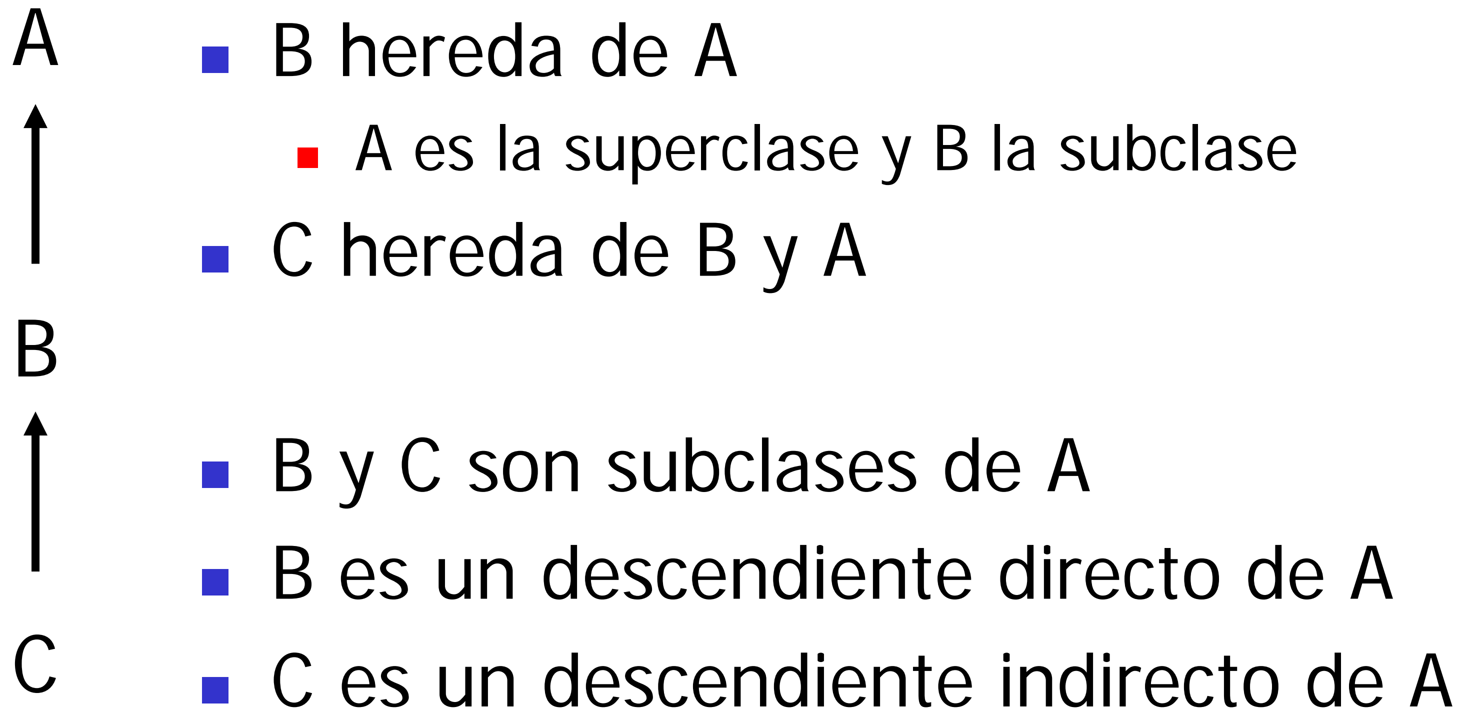


- No es tan sólo un mecanismo para compartir código
- Consistente con el sistema de tipos del lenguaje

Introducción

- Si una clase **B hereda** de otra clase **A** entonces:
  - B incorpora la estructura (atributos) y comportamiento (métodos) de la clase A
  - B puede incluir adaptaciones:
    - B puede **añadir** nuevos **atributos**
    - B puede añadir nuevos **métodos**
    - B puede **redefinir métodos**
- Las adaptaciones son dependientes del lenguaje

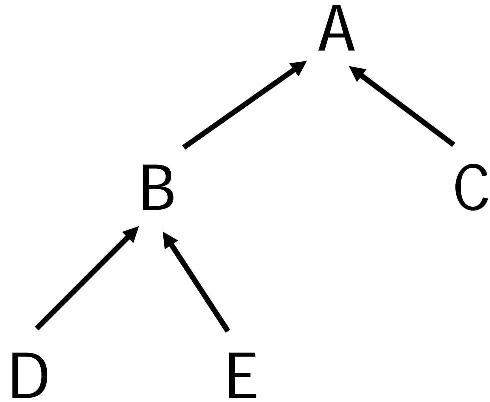
El proceso de herencia es transitivo



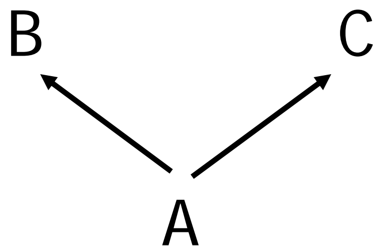
- B hereda de A
  - A es la superclase y B la subclase
- C hereda de B y A

- B y C son subclases de A
- B es un descendiente directo de A
- C es un descendiente indirecto de A

Tipos de herencia



- **Herencia simple**
  - Una clase puede heredar de una única clase.
  - Ejemplo: Java, C#



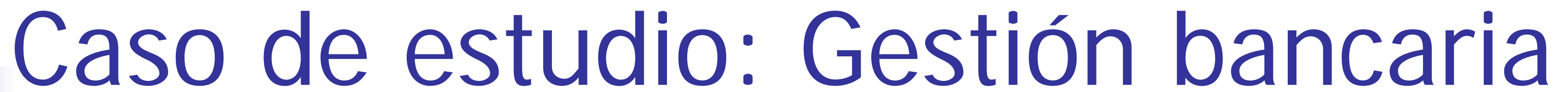
- **Herencia múltiple**
  - Una clase puede heredar de varias clases.
  - Clases forman un grafo dirigido acíclico
  - Ejemplos: Eiffel, C++

Diseño de jerarquías de herencia

- **Generalización** (**Factorización**)
  - Se detectan clases con un comportamiento común
  - Ejemplo: Libro y Revista son Publicaciones
- **Especialización** (**Abstracción**)
  - Se detecta que una clase es un caso especial de otra
  - Ejemplo: Rectangulo es un tipo de Poligono

- No hay receta mágica para crear buenas jerarquías
- Problemas con la evolución de la jerarquía

Caso de estudio: Gestión bancaria



- Además de abrir cuentas en el banco se pueden contratar depósitos.
- Un **depósito** permite a los clientes obtener una rentabilidad por su dinero
- Un depósito se caracteriza por:
  - **Estructura**: titular, capital, plazo en días, tipo de interés
  - **Comportamiento**:
    - *liquidar* el depósito una vez cumplido el plazo con lo que se devuelve al cliente el capital invertido más los intereses
    - *Consultar los intereses* producidos al final del periodo.

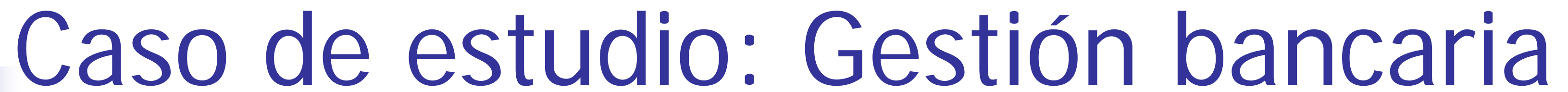
Clase Depósito (1/2)

```
public class Deposito {
  private Persona titular;
  private double capital;
  private int plazoDias;
  private double tipoInteres;

  public Deposito(Persona titular, double capital,
                  int plazoDias, double tipoInteres) {
     this.titular = titular;
     this.capital = capital;
     this.plazoDias = plazoDias;
     this.tipoInteres = tipoInteres;
   } …
```

Clase Depósito (2/2)

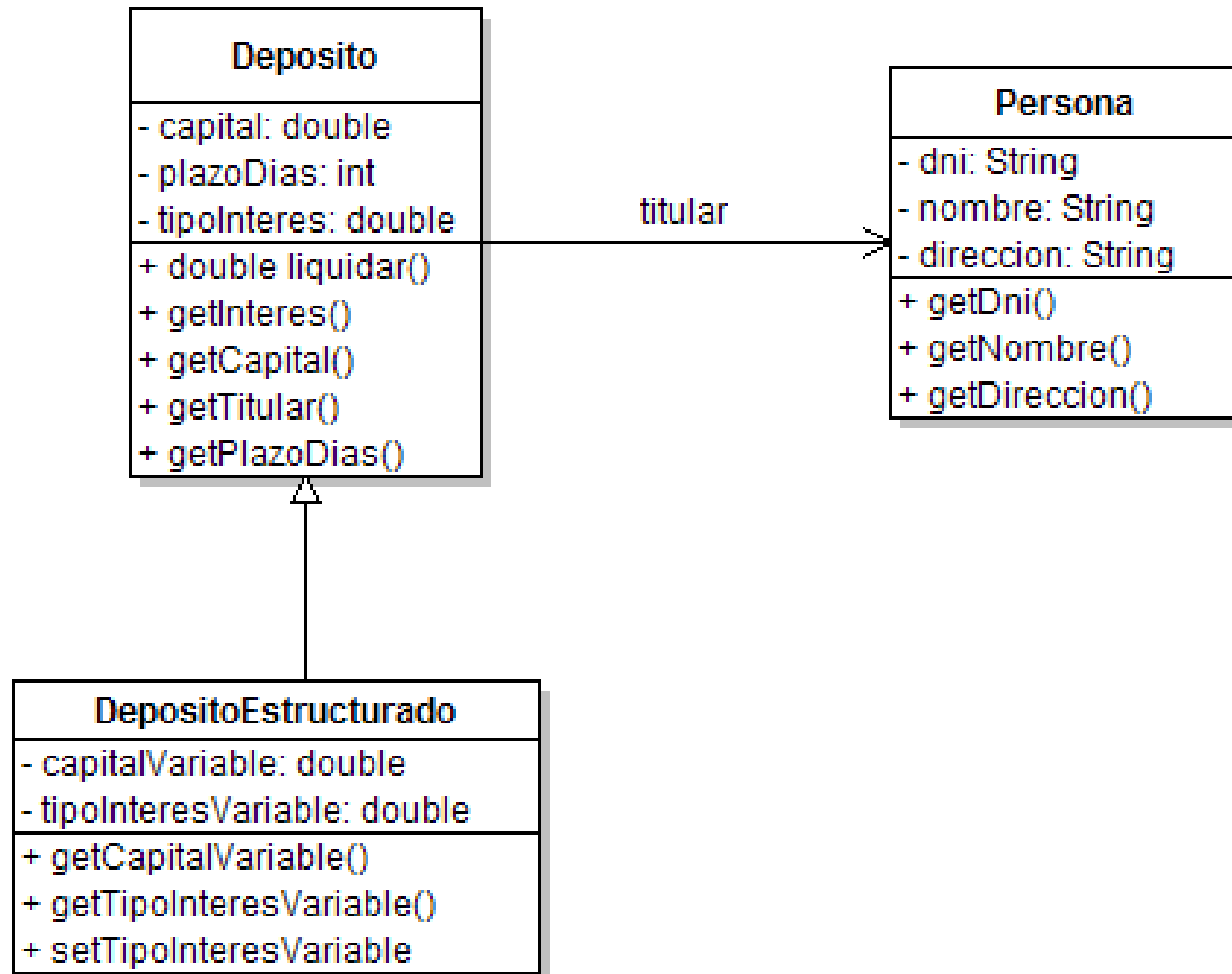
```
    public double liquidar() {
        return getCapital() + getIntereses();
    }

    public double getIntereses() {
        return (plazoDias * tipoInteres * capital)/365;
    }

    public double getCapital() {…}
    public int getPlazoDias() {…}
    public double getTipoInteres() {…}
    public Persona getTitular() {…}
}
```

Caso de estudio: Gestión bancaria



- Existe un tipo de depósito que se denomina **depósito estructurado**
- Un depósito estructurado **es un** tipo de depósito que se caracteriza por tener una parte del capital invertido a interés fijo y otra parte a interés variable
  - Comparte las características de depósito
  - Añade características nuevas
- **¿Debemos crear la clase depósito estructurado desde cero?**
- ¿Podemos aprovechar la existencia de similitudes y particularidades entre ambas clases?
  - **`DepositoEstructurado` hereda de `Deposito`**

Deposito y DepositoEstructurado



Clase DepositoEstructurado

```
public class DepositoEstructurado extends Deposito {
   private double tipoInteresVariable;
   private double capitalVariable;

   public DepositoEstructurado( … ) { … }

   public double getInteresesVariable() {
       return (getPlazoDias() * tipoInteresVariable
              * capitalVariable)/365;
   }
   public double getTipoInteresVariable() { … }
   public void setTipoInteresVariable(
              double tipoInteresVariable){ … }
   public double getCapitalVariable() { … }
}
```

Clase `DepositoEstructurado`

- Depósito estructurado hereda todas las características de depósito:
  - Hereda todos los **atributos** aunque no los vea porque se han definido como privados (Principio Ocultamiento de la Información)
  - Puede utilizar todos los **métodos** heredados como si fueran propios (por ejemplo, `getPlazoDias`)
- Añade nuevas características:
  - Atributos: `capitalVariable`, `tipoInteresVariable`
  - Métodos: `getCapitalVariable`, `setCapitalVariable`, `getTiopInteresVariable`.

Herencia y creación

- El constructor de la clase hija refina el comportamiento del padre
- En Java los constructores no se heredan
- La primera sentencia del constructor de la clase hija **SIEMPRE** es una llamada al constructor de la clase padre.
- La llamada al constructor del padre puede ser:
  - **Implícita**:
    - **Si se omite**, se llamará implícitamente al constructor por defecto
    - Equivale a poner como primera sentencia `super();`
    - Si no existe el constructor por defecto en la clase padre dará un error en tiempo de compilación
  - **Explícita**:
    - `super();` o `super(a,b);` o …
    - Dependiendo de si el constructor al que invocamos tiene o no argumentos

Clase DepositoEstructurado

```
public class DepositoEstructurado extends Deposito {
   private double tipoInteresesVariable;
   private double capitalVariable;

   public DepositoEstructurado(Persona titular, double capital,
int plazoDias, double tipoInteres, double tipoInteresVariable,
   double capitalVariable) {

      //Llamada explícita al constructor del padre
      super(titular, capital, plazoDias, tipoInteres);

      this.tipoInteresesVariable = tipoInteresVariable;
      this.capitalVariable = capitalVariable;
   }
…
}
```

Acceso protegido

- Una subclase hereda todos los atributos definidos en la superclase, pero **no puede acceder a los campos privados**.
- Para permitir que en un método de la subclase se pueda acceder a una característica (atributo/método) de la superclase, éste tiene que declararse como **`protected`**
  - Es discutible la visibilidad protegida para los atributos
  - Es útil la visibilidad protegida para los métodos
- **`Protected`:** características visibles a las subclases y al resto de clases del paquete

Resumen modificadores de acceso

- De más restrictivo a menos:
  - `private`
    - visible sólo en la clase donde se define
  - Sin modificador (por defecto)
    - visible a las clases del paquete
  - `protected`
    - visible a las subclases y al resto de clases del paquete
  - `public`
    - visible a todas las clases

Redefinición

- **¿Son válidos todos los métodos heredados de la clase depósito para un depósito estructurado?**
  - `getCapital`: debe devolver la suma del capital fijo más el capital variable en el caso del depósito estructurado
  - `getIntereses`: debe devolver la suma del interés fijo y el variable para un depósito estructurado
- Al heredar es posible redefinir los métodos para adaptarlos a la semántica de la nueva clase.
- La redefinición reconcilia la reutilización con la extensibilidad
  - Es raro reutilizar una clase sin necesidad de hacer cambios

Redefinición

- Los **atributos** no se pueden redefinir, sólo **se ocultan**
  - Si la clase hija define un atributo con el mismo nombre que un atributo de la clase padre, éste no está accesible
  - El campo de la superclase todavía existe pero no se puede acceder
- Un **método** de la subclase con la misma signatura (nombre y parámetros) que un método de la superclase lo está redefiniendo.
  - Si se cambia el tipo de los parámetros se está sobrecargando el método original

Redefinición de métodos

- Una clase hija puede redefinir un método de la clase padre por dos motivos:
  - **Reemplazo**: se sustituye completamente la implementación del método heredado manteniendo la semántica.
  - **Refinamiento**: se añade nueva funcionalidad al comportamiento heredado.
- En el refinamiento resulta útil invocar a la versión heredada del método.

Refinamiento: `super`

- La palabra reservada **`super`** se utiliza para invocar a un método de la clase padre

- Se debe utilizar para el refinamiento de métodos
  - **No se tiene que utilizar para invocar a métodos heredados**

- Se puede utilizar en el cuerpo de otros métodos:
  - `Deposito>>getCapital`: devuelve el capital fijo
  - `DepositoEstructurado>>getCapital`: devuelve el capital fijo + capital variable
  - En los métodos de `DepositoEstructurado` habrá que determinar cuál de las dos versiones del método `getCapital` es la que necesitamos.

Clase DepositoEstructurado

```
public class DepositoEstructurado extends Deposito {
    …
    @Override
    public double getCapital() {
        return super.getCapital() + getCapitalVariable();
    }

    @Override
    public double getIntereses() {
        return super.getIntereses()+getInteresesVariable();
    }
}
```

Adaptaciones al redefinir

- Se puede cambiar el **nivel de visibilidad**
  - Sólo si se relaja
    - "package" < protected < public
    - Podemos pasar de menos a más, pero no al contrario
- El **tipo de retorno** (regla covariante)
  - Siempre que el tipo de retorno del método redefinido sea **compatible** con el tipo de retorno del método original
  - Un tipo B es compatible con un tipo A si la clase B es subclase de A
  - Ejemplo: Jerarquía de Empleado
    - `Empleado >> public Empleado getColega(){…}`
    - `Jefe     >> public Jefe getColega() {…}`

Restringir la herencia y redefinición

- En Java se puede aplicar el modificador **`final`** a un **método** para indicar que no puede ser redefinido.

- Asimismo, el modificador `final` es aplicable a una **clase** indicando que no se puede heredar de ella.

- ¿El modificador `final` va contra el principio abierto-cerrado?

Ejemplo: Clase `Deposito`

```
public class Deposito {
 …
  public final double liquidar() {
      return getCapital() + getIntereses();
  }
}
```

- El algoritmo del método `liquidar` es el mismo para todos los depósitos, no se puede redefinir.

Polimorfismo

- El término **polimorfismo** significa que hay un nombre (variable, función o clase) y muchos significados diferentes (distintas definiciones).

- <u>Formas de polimorfismo</u>:
  - Polimorfismo de asignación (*variables polimorfas*)
  - Polimorfismo puro (*función polimorfa*)
  - Polimorfismo ad hoc (*sobrecarga*)
  - Polimorfismo de inclusión (*redefinición*)
  - Polimorfismo paramétrico (*genericidad*)

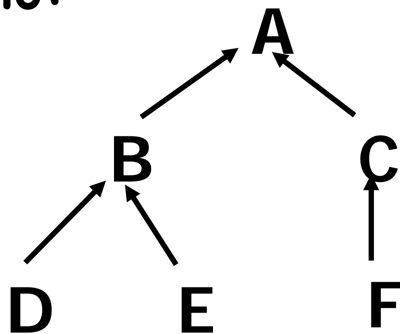
Polimorfismo de asignación

- Capacidad de una entidad de referenciar en tiempo de ejecución a objetos de diferentes clases.
- El conjunto de clases a las que se puede hacer referencia está **restringido por la herencia**
- Importante para escribir código genérico
- El polimorfismo implica que una variable tiene un **tipo estático** y un **tipo dinámico**

Tipo estático vs. tipo dinámico

- Tipo **estático**:
  - Tipo asociado en la **declaración**
- Tipo **dinámico**:
  - Tipo correspondiente a la clase del objeto conectado a la entidad en **tiempo de ejecución**
- Conjunto de tipos dinámicos (*ctd*):
  - Conjunto de posibles tipos dinámicos de una entidad

Ejemplo:




```
A oa; B ob; C oc;
```

te(oa) = A ctd(oa) = {A,B,C,D,E,F}

te(ob) = B ctd(ob) = {B, D, E}

te(oc) = C ctd(oc) = {C,F}

Polimorfismo de asignación

```
1. Deposito deposito = new Deposito(…);
2. DepositoEstructurado estructurado =
                         new DepositoEstructurado(…);
  //Asignación polimórfica
3. deposito = estructurado;
```

- El **tipo estático** de la variable `deposito` es siempre la clase `Deposito`
- El **tipo dinámico** de la variable `deposito` cambia en tiempo de ejecución:
  - En 1 es `Deposito`
  - En 3. es `DepositoEstructurado`

Polimorfismo puro

- Método que puede recibir como argumento objetos de diferentes tipos
  - El parámetro es una entidad polimórfica

```
public double indiceRentabilidad(Deposito deposito) {
 return deposito.getIntereses()/deposito.getCapital();
}
```

- El método `indiceRentabilidad` podría recibir como parámetro un objeto de la clase `Deposito` o de la clase `DepositoEstructurado`
- En tiempo de ejecución se determinará la versión de `getIntereses` y `getCapital` que debe ejecutarse (*ligadura dinámica*)

Polimorfismo puro vs. Sobrecarga

- Funciones sobrecargadas ≠ funciones polimórficas
- <u>Sobrecarga</u>:
  - Dos o mas funciones comparten el nombre y distintos argumentos (nº y tipo). **El nombre es polimórfico**.
  - Distintas definiciones y tipos (distintos comportamientos)
  - Función correcta se determina en **tiempo de compilación** según la signatura.
- <u>Funciones polimórficas</u>:
  - Una única función que puede recibir una variedad de argumentos (comportamiento uniforme).
  - La ejecución correcta se determina dinámicamente en **tiempo de ejecución**

Polimorfismo puro vs. sobrecarga

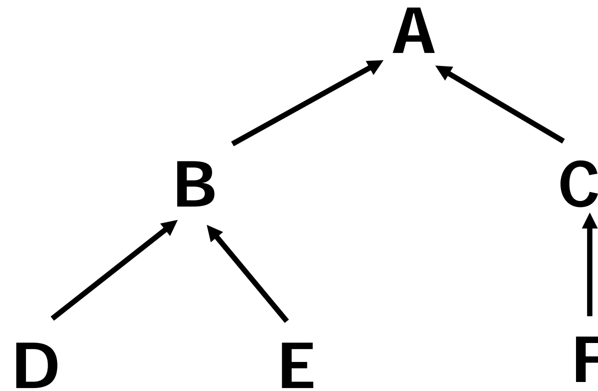
- **Polimorfismo puro:**
  - Un único código con distintas interpretaciones

```
public double indiceRentabilidad(Deposito deposito) {
   return deposito.getIntereses()/deposito.getCapital();
}
```

- **Sobrecarga:**
  - Un código para cada tipo de `Deposito`

```
public double indiceRentabilidad(Deposito deposito) {
    …
}

public double indiceRentabilidad(DepositoEstructurado deposito){
    …
}
```

Herencia y sistema de tipos



`A oa; B ob; C oc; D od;`

¿Son legales las siguientes asignaciones?

`oa = ob; oc = ob; oa = od`

¿Es legal el mensaje `od.metodo1`?

Herencia y sistema de tipos

Un lenguaje OO tiene comprobación estática de tipos si está equipado con un conjunto de reglas de consistencia, cuyo cumplimiento es controlado por los compiladores, y que si el código de un sistema las cumple se garantiza que ninguna ejecución de dicho sistema puede provocar una violación de tipos

- Reglas básicas:
  - **Regla de compatibilidad de tipos** → asignaciones válidas
  - **Regla de validez de mensajes** → mensajes válidos
- **Beneficios** esperados:
  - Fiabilidad
  - Legibilidad
  - Eficiencia

Herencia y sistema de tipos

- Inconveniente → **Política pesimista:**
  - "al tratar de garantizar que ninguna operación fallará, el compilador puede rechazar código que tenga sentido en tiempo de ejecución"

- Ejemplo: `int i; double d;`
  - `i=d;`
    - "Discrepancia de tipos: no se puede convertir de `double` a `int`"
  - `i= 0.0;` Funcionar
  - `i= -3.67;` No funcionaría
  - `i= 3.67 - 3.67;` Funcionaría

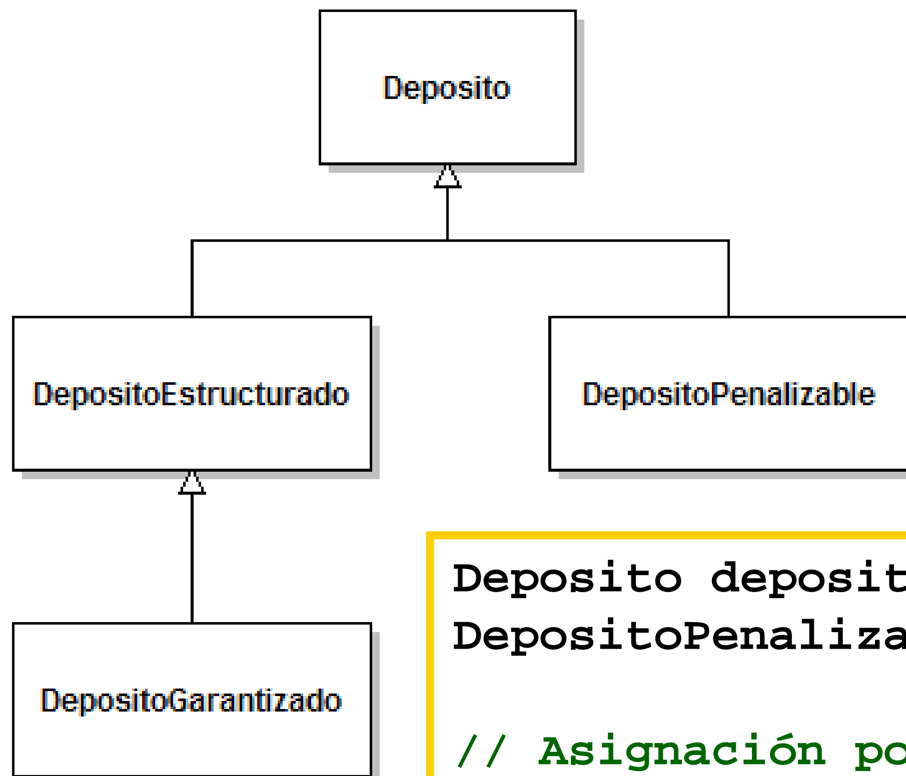
Compatibilidad de tipos

- Un tipo **B es compatible con** un tipo **A** sólo si la clase B es descendiente de la clase A.
  - `DepositoEstructurado` es compatible con `Deposito`

- Una **asignación polimórfica** es válida sólo si el tipo estático de la parte izquierda es compatible con el tipo de la parte derecha.

- El **paso de parámetros** es válido sólo si el tipo del parámetro real es compatible con el tipo del parámetro formal.

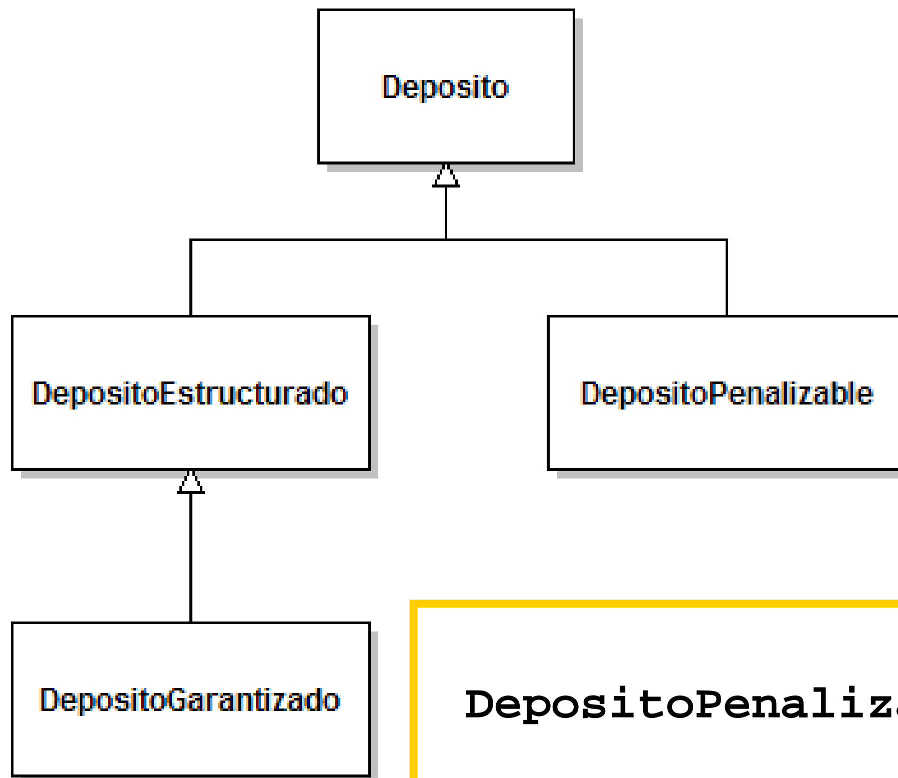
Compatibilidad de tipos



- `DepositoPenalizable` es compatible con `Deposito`
- `DepositoGarantizado` es compatible con `Deposito` y `DepositoEstructurado`
- `DepositoGarantizado` **no** es compatible con `DepositoPenalizable`

```
Deposito deposito = new DepositoEstructurado(…);
DepositoPenalizable penalizable =
                     new DepositoPenalizable(…);
// Asignación polimórfica válida
deposito = penalizable;

// Asignación polimórfica no válida
DepositoGarantizado garantizado = penalizable;
```

Compatibilidad de tipos



- El tipo del parámetro del método `indiceRentabilidad` es `Deposito`.
- Puede recibir cualquier tipo de depósito como parámetro.

```
DepositoPenalizable penalizable =
                   new DepositoPenalizable(…);

banco.indiceRentabilidad (penalizable);
```

Validez de mensajes

- Una llamada a un método `obj.met()` es válida si:
  - el método `met` está definido en la clase del tipo estático de `obj`.
  - Los parámetros reales son compatibles con los parámetros formales
  - El método es visible para la clase que invoca el mensaje
- Sobre un objeto declarado de tipo `DepositoEstructurado` se pueden invocar a todos los métodos definidos en la clase `Deposito` y la clase `DepositoEstructurado`
- Sobre un objeto declarado de tipo `Deposito` no se puede invocar a los métodos de `DepositoEstructurado`

Validez de mensajes

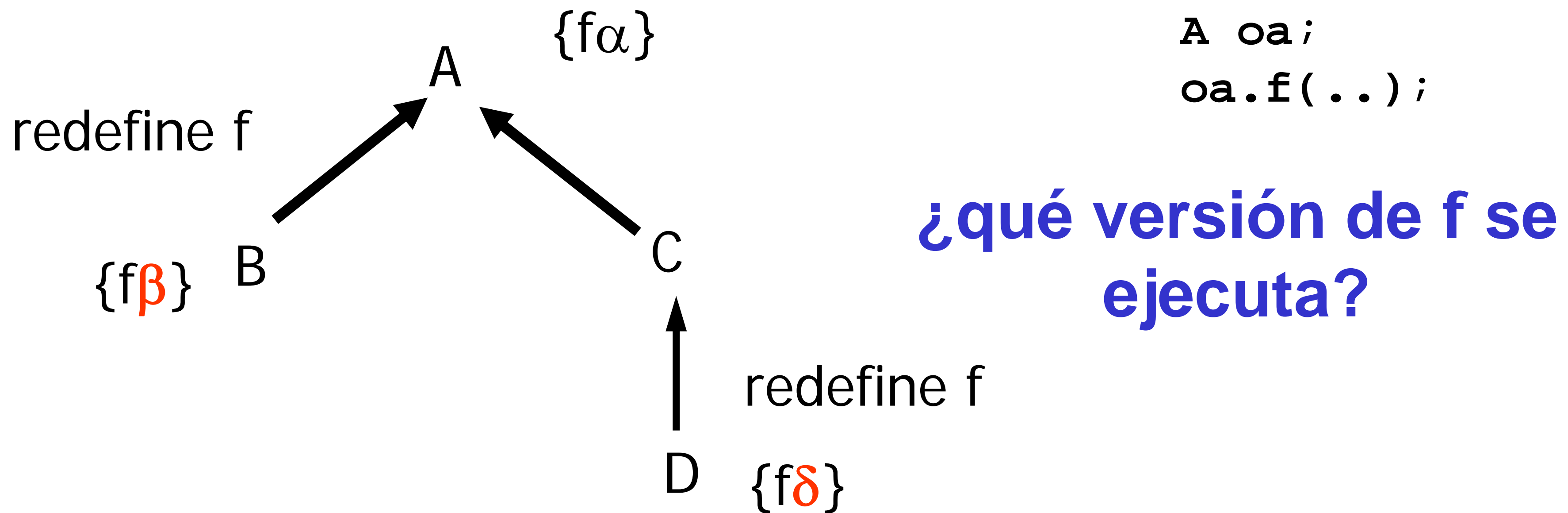
```java
Deposito deposito = new Deposito(…);
DepositoEstructurado estructurado =
                          new DepositoEstructurado(…);

estructurado.liquidar();                //OK HEREDADO DE DEPOSITO
estructurado.getCapital();              //OK HEREDADO DE DEPOSITO
estructurado.getCapitalVariable();  //OK MÉTODO PROPIO

deposito = estructurado;

deposito.getIntereses();                //OK METODO DE DEPOSITO
deposito.getCapitalVariable();      //ERROR COMPILACION!!!
```

Ligadura dinámica



Regla de la ligadura dinámica

La forma dinámica del objeto determina la versión de la operación que se aplicará.

Ligadura dinámica

- La **versión** de una rutina en una clase es la introducida por la clase (redefinición u original) o la heredada.

- **Ejemplo 1:**

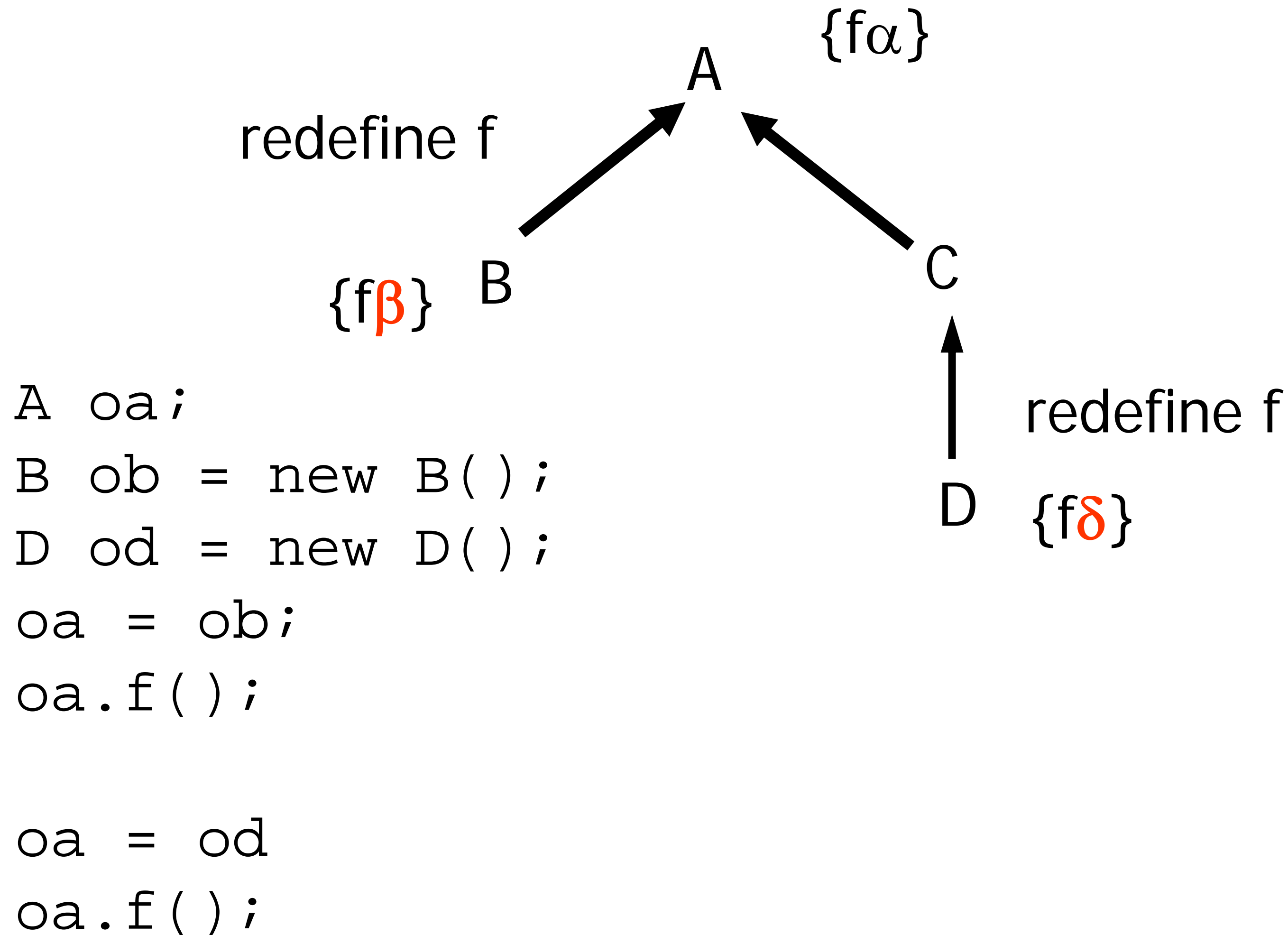
```
Deposito deposito = new Deposito(…);
DepositoEstructurado estructurado =
                           new DepositoEstructurado(…);

deposito.getCapital();       //versión de Deposito
estructurado.getCapital();   //versión de DepositoEstructurado

deposito = estructurado;

deposito.getCapital();     //versión de DepositoEstructurado
```

Ejercicio: ¿Qué versión se ejecutará?




```
A oa;
B ob = new B();
D od = new D();
oa = ob;
oa.f();

oa = od
oa.f();
```

Ligadura Dinámica

Ejemplo 2:

```
public double posicionGlobal(Deposito[] depositos) {
    double posicion = 0;
    for (Deposito deposito : depositos) {
        posicion += deposito.getCapital();
    }
    return posicion;
}
```

¿Qué sucede si aparece un nuevo tipo de deposito?

- ¿Qué relación existe entre ligadura dinámica y comprobación estática de tipos?

Sea el mensaje **x.f ()**, la **comprobación estática de tipos** garantiza que al menos existirá una versión aplicable para **f**, y la **ligadura dinámica** garantiza que se ejecutará la versión más apropiada

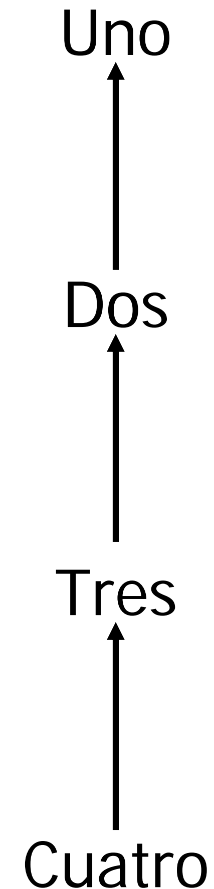
Ligadura dinámica y super

```
class Uno {
    public int test(){return 1;}
    public int result1(){return this.test();}
}

class Dos extends Uno{
    public int test(){return 2;}
}

class Tres extends Dos{
    public int result2(){return this.result1();}
    public int result3(){return super.test();}
}

class Cuatro extends Tres{
    public int test(){return 4;}
}
```



Ligadura dinámica y super

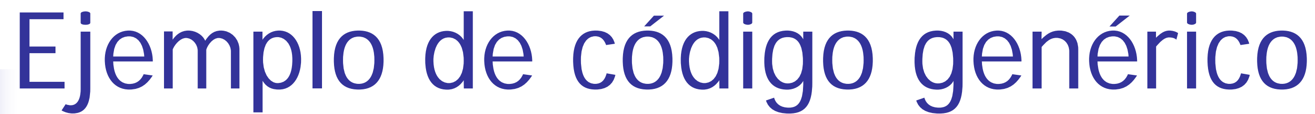
```
public class PruebaSuperThis{
    public static void main (String args[]){
        Uno ob1 = new Uno();
        Dos ob2 = new Dos();
        Tres ob3 = new Tres();
        Cuatro ob4 = new Cuatro();

        System.out.println("ob1.test = "+ ob1.test());              → 1
        System.out.println("ob1.result1 = " + ob1.result1());      → 1
        System.out.println("ob2.test = " + ob2.test());            → 2
        System.out.println("ob2.result1 = " + ob2.result1());      → 2
        System.out.println("ob3.test = " + ob3.test());            → 2
        System.out.println("ob4.result1 = " + ob4.result1());      → 4
        System.out.println("ob3.result2 = " + ob3.result2());      → 2
        System.out.println("ob4.result2 = " + ob4.result2());      → 4
        System.out.println("ob3.result3 = " + ob3.result3());      → 2
        System.out.println("ob4.result3 = " + ob4.result3());      → 2
    }
}
```

Código genérico

- Un único código con diferentes interpretaciones en tiempo de ejecución
- Es fundamental que el lenguaje soporte el polimorfismo
- El mismo código ejecutará cosas distintas en función del tipo dinámico de la entidad polimórfica (*ligadura dinámica*)
- Gracias al polimorfismo y la ligadura dinámica se satisface el criterio de reutilización de **variación de la implementación**.

Ejemplo de código genérico



- El método liquidar de la clase `Deposito` es un ejemplo de código genérico:

```
public double liquidar() {
  return getCapital() + getIntereses();
}
```

- Según el tipo de depósito sobre el que se aplique se ejecutará una versión u otra de los métodos `getCapital` y `getIntereses`

Clase `Object`

- Puede existir una **clase "raíz"** en la jerarquía de la cual heredan las demás directa o indirectamente
- En Java esta clase es la clase **`Object`**
- La clase `Object` incluye las características comunes a todos los objetos
- Una variable de tipo `Object` puede apuntar a cualquier tipo del lenguaje, incluidos los tipos primitivos (*autoboxing*)

```
Object obj = 7;
float i = (Float)obj;
```

OK!!

Métodos clase Object

- `public boolean equals(Object obj)`
  - Igualdad de objetos
- `protected Object clone()`
  - Clonación de objetos
- `public String toString()`
  - Representación textual de un objeto
- `public Class getClass()`
  - Clase a partir de la que ha sido instanciado un objeto.
- `public int hashCode()`
  - Código hash utilizado en las colecciones.

Copia de objetos

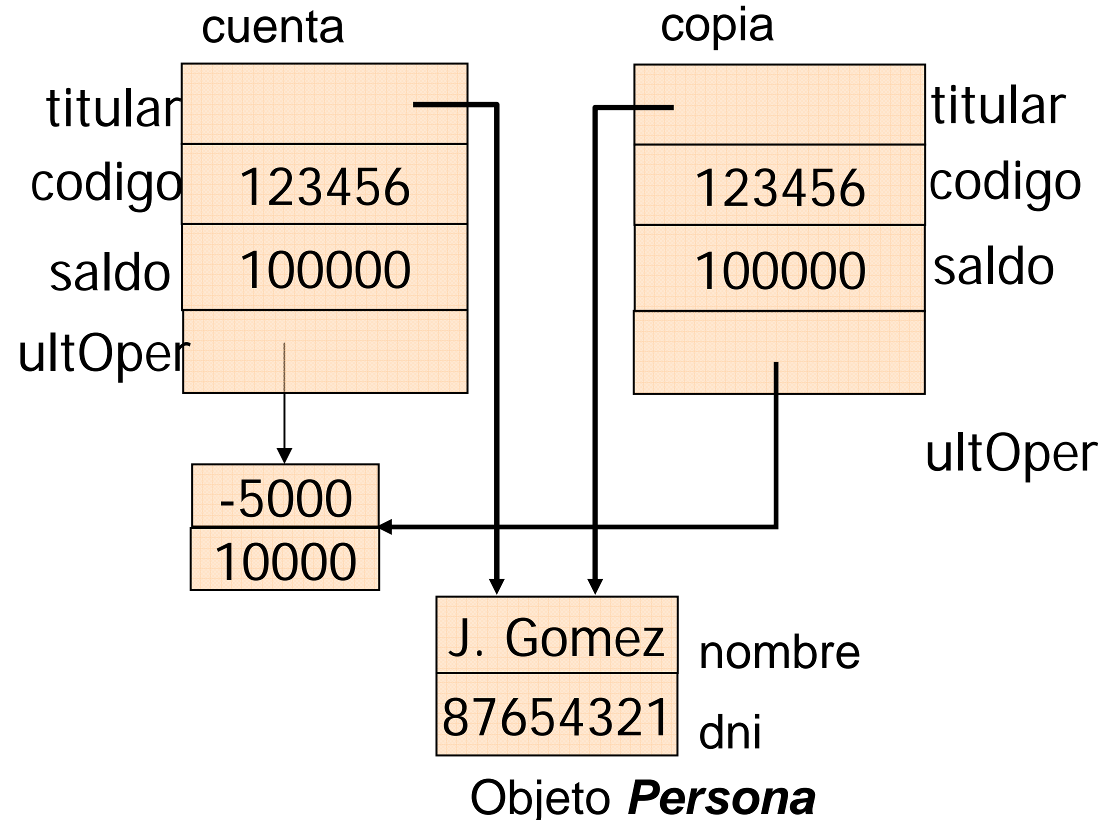
- La **asignación de referencias** (=) copia el *oid* del objeto y no la estructura de datos.
- Para obtener una copia de un objeto hay que aplicar el **método `clone`**.
- El método clone está implementado en la clase `Object` (es heredado) pero no es aplicable por otras clases (visibilidad `protected`).
- La clase debe **redefinir el método clone** para aumentar la visibilidad y crear una copia que se adapte a sus necesidades.
- La versión de la clase `Object` (`super.clone()`) construye una **copia superficial** de la instancia actual.

Tipos de copia

- **Tipos** de copia:
  - **Copia superficial**: los campos de la copia son exactamente iguales a los del objeto receptor.
  - **Copia profunda**: los campos primitivos de la copia son iguales y las referencias a objetos son copias profundas.
  - **Adaptada**: adaptada a la necesidad de la aplicación.

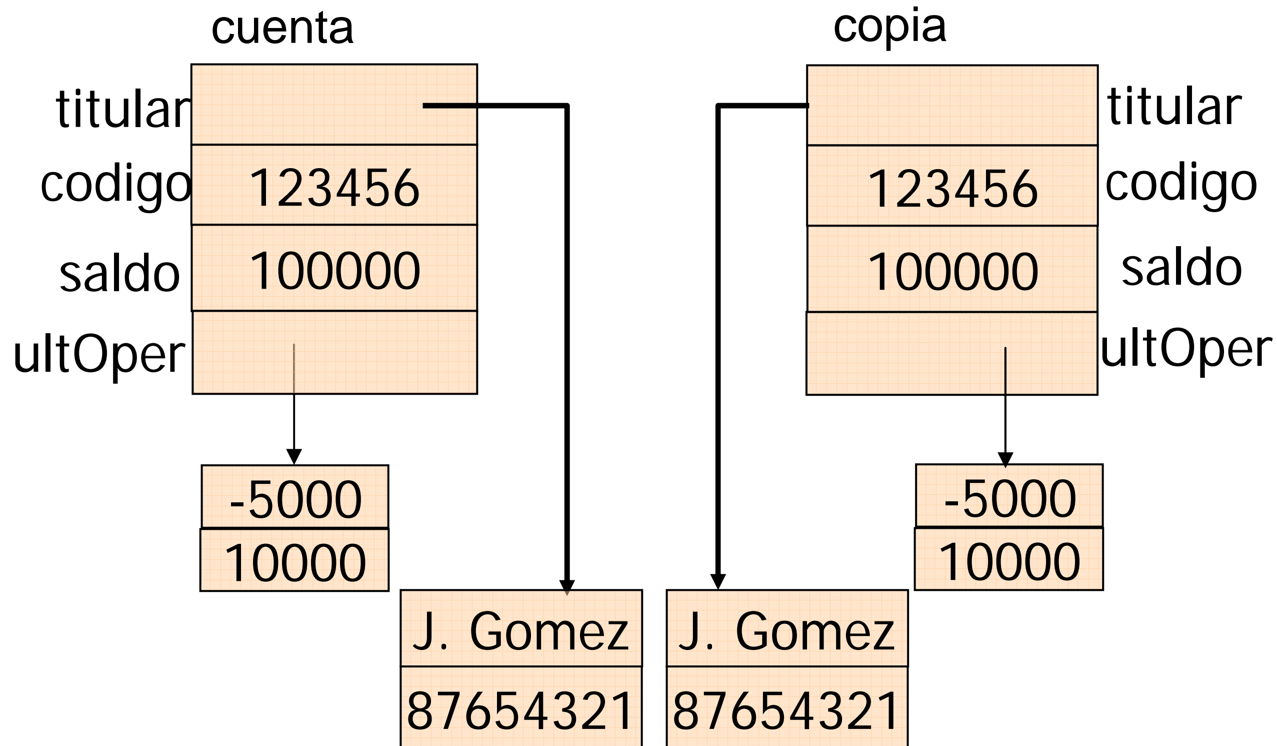
Copia superficial de Cuenta

- *Aliasing:* incorrecto al compartir las últimas operaciones.
- No deberían tener el mismo código



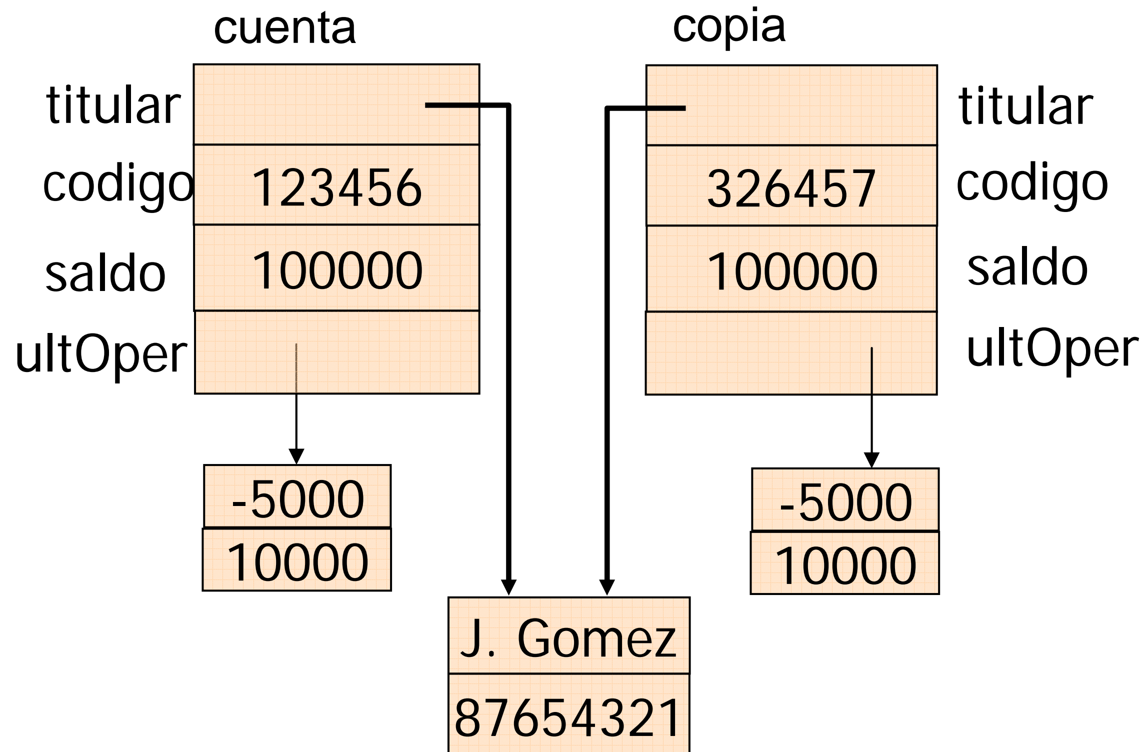
Copia profunda de Cuenta

- No tiene sentido duplicar el objeto cliente y que tengan el mismo código.



Copia correcta de Cuenta

- Una copia adaptada cumple los requisitos de la aplicación



Igualdad de objetos

- El operador de igualdad (==) compara referencias (identidad)
- El método **`equals`** permite implementar la igualdad de objetos
- La implementación en la clase `Object`:
  - ```
    public boolean equals (Object objeto){
      return this == objeto;
    }
    ```
  - Por tanto: `a.equals(b);  //false si a!=b`
- Es necesario **redefinir el método equals** en las clases donde necesitemos la operación de igualdad.
- Sin embargo, hay que elegir la **semántica** de igualdad más adecuada para la clase.
    ```

Igualdad

- **Tipos de igualdad**:
  - **Superficial**: los campos primitivos de los dos objetos son iguales y las referencias a objetos idénticas (comparten los mismos objetos).
  - **Profunda**: los campos primitivos son iguales y las referencias son iguales (`equals`) en profundidad.
  - **Adaptada**: adaptada a las necesidades de la aplicación.

- **¿Cuándo dos cuentas son iguales?**

Métodos `Object`

- En el seminario 2 de prácticas se incluye:
  - Patrón para redefinir el método `equals`.
  - Redefinición del método `clone` utilizando la copia superficial facilitada por la clase `Object`.
  - Patrón para la definición del método `toString`.
  - Consejos para la definición de los tres métodos en una jerarquía de herencia.

Determinar el tipo dinámico de los objetos en tiempo de ejecución

- **Estructuras de datos polimórficas**:
  - `Deposito[] depositos;`
  - `List<Deposito> depositos;`
  - Pueden contener referencias a objetos de cualquier tipo de depósito
- ¿Cómo puedo modificar el interés variable de todos los depósitos estructurados?
  - Sobre `depositos[i]` o `depositos.get(i)` sólo se pueden aplicar operaciones definas en la clase `Deposito`.
- ¿Cómo puedo aplicar las operaciones de la clase `DepositoEstructurado`? → casting o narrowing

Casting o narrowing

- El compilador permite hacer un casting de una variable polimórfica a uno sus posibles tipos dinámicos.

```
public void setTipoInteresVariable(double tipo,
                                   Deposito[] depositos) {
  for (Deposito d : depositos) {
     DepositoEstructurado de = (DepositoEstructurado)d;
     de.setTipoInteresVariable(tipo);
  }
}
```

- **Problema:**
  - Si la conversión no es posible falla en tiempo de ejecución
  - Salta la excepción `ClassCastException` y se aborta la ejecución.
- **Solución:**
  - Preguntar por el tipo antes de hacer el casting: `instanceof` vs. `getClass()`

Operador instanceOf

```
public void setTipoInteresVariable(double tipo,
                                   Deposito[] depositos) {
   for (Deposito d : depositos) {
     if (d instanceof DepositoEstructurado){
       DepositoEstructurado de = (DepositoEstructurado)d;
       de.setTipoInteresVariable(tipo);
     }
   }
}
```

instanceof vs. getClass

- Comprueba si el tipo de una variable es **compatible** con un tipo dado
  - Es de ese tipo o alguna de sus subclases
- No es lo mismo hace la comprobación con `instanceof` que con el método `getClass` heredado de la clase `Object`
  - `deposito.getClass()==DepositoEstructurado.class` comprueba si el tipo dinámico de la variable `deposito` **es exactamente** la clase `DepositoEstructurado`
  - Ver el seminario 2 de prácticas para más detalle

Genericidad basada en la clase `Object`

- Hasta la versión 1.4 no se incluye la genericidad como elemento del lenguaje.
- Se consigue gracias a que **toda clase es compatible con la clase `Object`.**
  - Las colecciones son contenedores de objetos
- **Problemas** => Se pierde la información del tipo:
  - Una entidad de tipo `Object` puede hacer referencia a cualquier tipo.
  - Hay que efectuar un **cast** antes de utilizar el objeto que se recupera de la entidad "genérica" (es un objeto de tipo `Object`).
  - Se detecta un objeto del tipo no deseado en tiempo de ejecución.

Genericidad basada en la clase `Object`

```
class Nodo{
   private Object valor;
   private Nodo siguiente;
   …
}
public class Lista{
   private Nodo cabeza;
   …
   public Object getFirst(){
       return cabeza.getValor();
   }
}
```

- El campo valor del Nodo de la lista puede hacer referencia a cualquier tipo de objeto.

Genericidad basada en la clase `Object`

```
public class Pila {
   private ArrayList contenido;
   //ArrayList es un contenedor de Object
   private int tope = 0;

   public void push (Object obj){
        contenido.add(obj);
        ++tope;
   }
   public Object pop () {
        Object elemento = contenido.get(tope);
        contenido.remove(elemento);
        return elemento;
   }
}
```

Genericidad basada en la clase `Object`

```
ArrayList depositos;          //quiero que sea de depósitos
Deposito d1, d2; Cuenta cta;
…
p.add(d1);
p.add(cta);                              //NO daría error el compilador
d2=depositos.get(0);                     //error asignamos un Object
d2=(Deposito)depositos.get(1);  //OK en compilación
   //error en ejecución si lo que se extrae no es un depósito
```

- **Perdemos la ventaja de la comprobación estática de tipos, las comprobaciones las hace el programador**

Genericidad en Java

- A partir de la versión 5 se incluye la definición de clases genéricas como elemento del lenguaje: **List<T>**
- Las operaciones aplicables sobre una entidad genérica (variable o atributo de tipo T) son las operaciones disponible para cualquier tipo.
- Cuando se instancia una clase genérica el conjunto de posibles tipos que puede contener viene determinado por la jerarquía de herencia
  - **List<Deposito>** contendrá cualquier tipo de depósito.
- **¿Es posible ampliar el conjunto de operaciones aplicables sobre el tipo genérico?**

¿Existe algún error en el siguiente código?

```
// Almacena elementos ordenados de menor a mayor
public class ListaOrdenada <T> {
   private List<T> contenido;
   …
   public void add (T newElement){
       int i=0;
       while (i < contenido.size() &&
              contenido.get(i).menorQue(newElement)){
          ++i;
       }
       contenido.add(i, newElement);
   }
}
```

¿Es posible que este código compile?

Solución: Genericidad Restringida

- Es posible restringir las clases a las que se puede instanciar el parámetro genérico formal de una clase genérica.

```
public class A <T extends B>
```

- Sólo es posible instanciar T con descendientes de la clase B.
- Las operaciones permitidas sobre entidades de tipo T son aquellas permitidas sobre una entidad de tipo B.
- **Ejemplos:**
  - `class ListaOrdenada <T extends Ordenable>`
  - El tipo `Ordenable` debe incluir la operación **`menorQue`**

Genericidad restringida

```java
public class CarteraAhorro<T extends Deposito>{
   private List<T> contenido;

   public void liquidar(){
       for (T deposito : contenido)
             deposito.liquidar();
   }
}
```

- Al restringir la genericidad podemos utilizar los métodos de `Deposito` sobre las entidades de tipo genérico (`T`)
- `T` sólo podrá instanciarse con la clase `Deposito` o cualquiera de sus subclases.

Genericidad Restringida

- Una clase se puede restringir a más de un tipo:
  - `class CarteraAhorro<T extends Deposito & Ordenable>`
  - Las operaciones aplicables sobre una entidad genérica son las disponibles tanto en `Deposito` como en `Ordenable`
- La genericidad no restringida equivale a restringir la genericidad a la clase `Object`
  - `List`
  - `List`**`<T extends Object>`**
- ¿Son equivalentes las declaraciones:
  - **`CarteraAhorro<Deposito>`**
  - **`CarteraAhorro<T extends Deposito>`**

  ?

¿Son equivalentes?

```
public class CarteraAhorro<T extends Deposito>{
  private T [] contenido;

  public void insertar(T elemento){
        ...
  }
  ...
}
```

```
public class CarteraAhorro{
   private Deposito[] contenido;

   public void insertar(Deposito deposito){
   ...
   }
   ...
}
```

Herencia de una clase genérica

- La subclase puede seguir siendo genérica:

```
class Pila<T> extends ArrayList<T> {…}
```

- Al heredar de una clase genérica se puede restringir el tipo:

```
class ListaOrdenada<T extends Comparable>
                    extends LinkedList<T> {…}
```

- Al heredar de una clase genérica se puede instanciar el tipo:

```
class CajaSeguridad extends LinkedList<Valorable> {…}
```

Genericidad y Sistema de tipos

```
List<Deposito> depositos;
List<DepositoEstructurado> estructurados =
            new LinkedList<DepositoEstructurado>();

depositos = estructurados; //ERROR en tiempo de compilación
```

Genericidad y sistema de tipos

- El compilador trata de asegurar que sobre una lista de depósitos estructurados no se incluyan depósitos con plazo fijo.
- Sin embargo, existe un **agujero en el sistema de tipos** de Java:

```
List lista;
lista = new LinkedList<Deposito>();                // compila
lista = new LinkedList<DepositoEstructurado>(); // compila
lista.add(“POO”);                                  // compila!!
```

- Cuando se declara una variable de un tipo genérico sin parametrizar se asigna el **tipo puro** (*raw*):
  - `Object` en el caso de una clase genérica no restringida
  - Tipo por el que se restringe la genericidad

Genericidad y sistema de tipos

- Utilizando el tipo puro podemos saltar la seguridad de tipos:

```
List<DepositoEstructurado> estructurados = new ...;
List<Deposito> depositos;

//depositos = estructurados error en tiempo de compilación
//Sin embargo …
List lista;

lista = estructurados;
depositos = (List<Deposito>)lista;  //COMPILA!!
```

Genericidad y tipos comodín

- ¿Es posible pasar como parámetro un colección de depósitos estructurados al método `posicionGlobal`?

```
public double posicionGlobal(List<Deposito> depositos) {
      double posicion = 0;
      for (Deposito deposito : depositos) {
            posicion += deposito.getCapital();
      }
      return posicion;
  }
```

- `List<DepositoEstructurado>` no es compatible con el parámetro del método!!!

Genericidad y tipos comodín

- Solución: **Tipo comodín**

```
double posicionGlobal (List<? extends Deposito> depositos)
```

- Significa: "permite cualquier lista genérica instanciada a la clase Deposito o a cualquiera de sus subclases"
- Si pasamos como parámetro un objeto List<DepositoEstructurado>, éste será el tipo reconocido dentro del método.
- No hay riesgo de inserciones ilegales dentro de la colección.

Genericidad y máquina virtual

- **La máquina virtual no maneja objetos de tipo genérico**

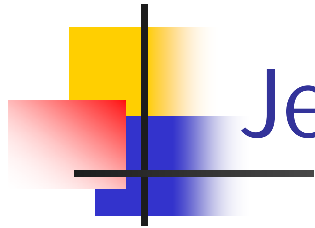
- Todo tipo genérico se transforma en un *tipo puro*
  - La información del tipo genérico se “borra” en tiempo de ejecución

- Todas las consultas sobre el tipo dinámico siempre devuelven el tipo puro:

```
(lp instanceof List<Deposito>) //no compila
(lp instanceof List)           //si compila
```

Clases abstractas

- En el banco existen cuentas y depósitos con propiedades en común:
  - Ambos tienen un `titular`
  - Ambos tienen un método para calcular los impuestos (`getImpuestos`) → 18% del beneficio
- Podemos generalizar (*factorizar comportamiento común*) y definir la clase `ProductoFinanciero`
  - ¿Tiene sentido crear instancias de la clase `ProductoFinanciero`?
  - ¿Cómo se implementa el método `getBeneficio` en `ProductoFinanciero`?
  - ¿Tienen sentido definir el método `getBeneficio` como un método vacío en `ProductoFinanciero`?



Jerarquía de productos financieros

ProductoFinanciero*
+ getTitular()
+ getImpuestos()
+ getBeneficio()*

titular

Persona
- dni: String
- nombre: String
- direccion: String
+ getDni()
+ getNombre()
+ getDireccion()

Deposito
- capital: double
- plazoDias: int
- tipoInteres: double
+ double liquidar()
+ getInteres()
+ getCapital()
+ getPlazoDias()

Cuenta
- saldo: double
+ void ingreso(cantidad: double)
+ void reintegro(cantidad: double)

DepositoEstructurado
- capitalVariable: double
- tipoInteresVariable: double
+ getCapitalVariable()
+ getTipoInteresVariable()
+ setTipoInteresVariable

- `ProductoFinanciero` es una clase abstracta
- El método `getBeneficio` es un método abstracto

#
Clase ProductoFinanciero

```java
public abstract class ProductoFinanciero {
    private Persona titular;

    public ProductoFinanciero(Persona titular) {
        this.titular = titular;
    }

    public Persona getTitular() { return titular; }

    public double getImpuestos() {
        return getBeneficio() * 0.18;
    }

    public abstract double getBeneficio();
}
```

Clases abstractas

- Toda clase que contenga algún método abstracto (heredado o no) es abstracta.
- Una clase puede ser abstracta y no contener ningún método abstracto.
- Especifica una **funcionalidad que es común** a un conjunto de subclases aunque no es completa.
- Puede ser total o parcialmente abstracta.
- **No es posible crear instancias** de una clase abstracta, pero si declarar entidades de estas clases.
  - Aunque la clase puede incluir la definición del constructor.
- Las clases abstractas **sólo tienen sentido en un lenguaje con comprobación estática de tipos**.

Clases parcialmente abstractas

- Contienen métodos abstractos y efectivos
  - Ejemplo: Clase `ProductoFinanciero`
- Los métodos efectivos pueden hacer uso de los abstractos.
  - Ejemplo: método `getImpuestos()`
  - **Método Plantilla!!**
- Importante mecanismo para incluir **código genérico**.
- Incluyen comportamiento abstracto común a todos los descendientes.
- Se satisface el requisito de reutilización de **factorizar comportamiento común.**

Clases totalmente abstracta

- Sólo declara métodos abstractos
- Equivale a la definición de un TAD
  - No incluye implementación
- **¿Tendría sentido que una clase pudiera heredar de más de una clase totalmente abstracta?**
  - Evita los problemas de heredar más de una implementación
  - Permite ampliar el conjunto de tipos que puede representar un objeto de una clase
- Solución: **Interfaces**

Interfaces en Java

- Por defecto, toda la definición de una interfaz es pública
- Sólo contiene definiciones de métodos y constantes
  - Los métodos son abstractos, no es necesario especificarlo explícitamente
- No se puede crear un objeto del tipo de la interfaz, pero si utilizarlo en la declaración de variables.
  - Una interfaz es un tipo

Ejemplo de Interfaz

- Un depósito puede ser amortizable lo que permite rescatar parte del capital antes del vencimiento.

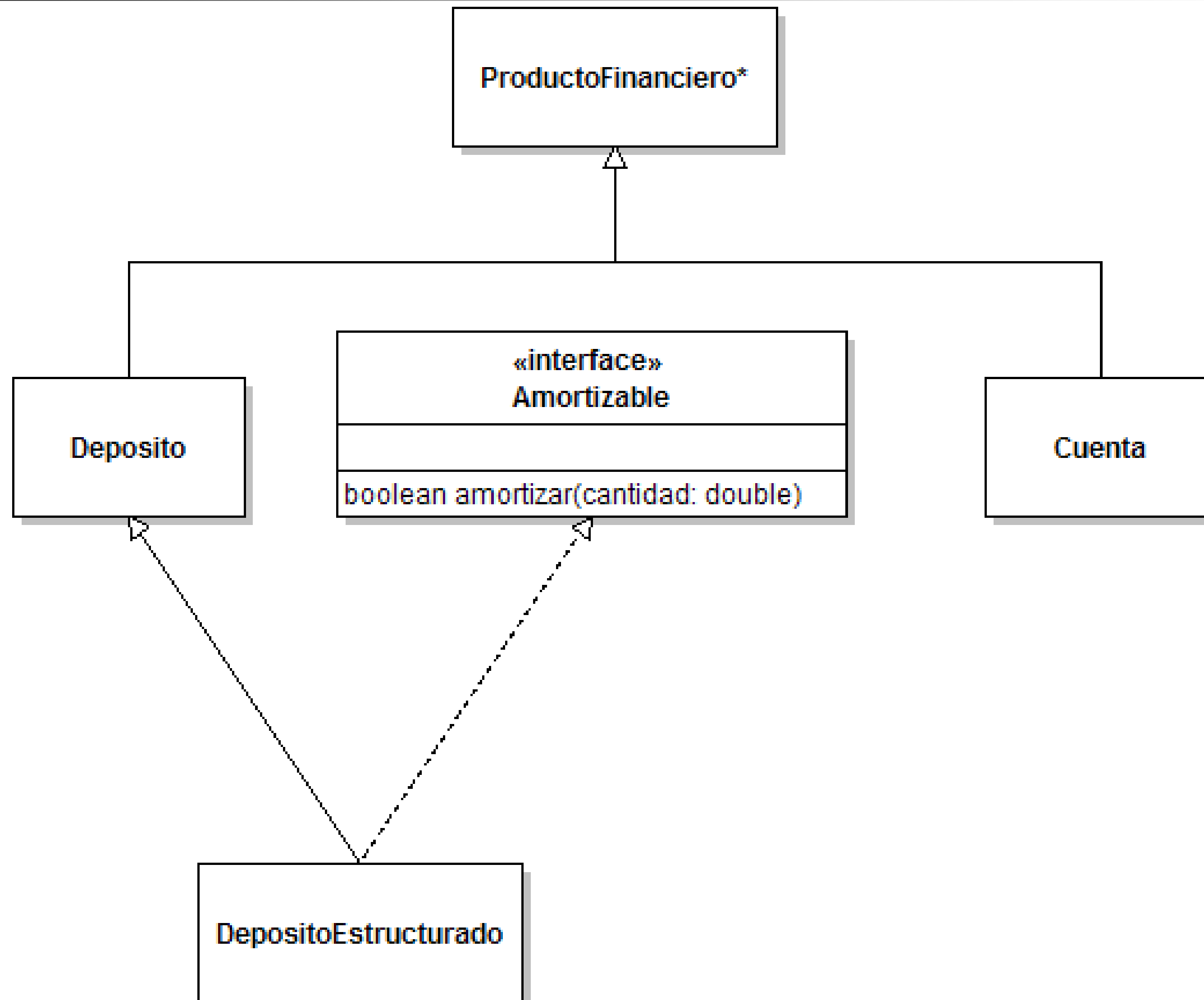
```
public interface Amortizable {
      double LIMITE = 10000.0;
      boolean amortizar(double cantidad);
}
```

- El método `amortizar` devuelve un valor booleano para indicar si se pudo realizar la amortización.

Implementación de una interfaz

- Se dice que una clase **implementa** una interfaz
  - La clase debe implementar los métodos definidos en la interfaz
  - Si una clase no implementa todos los métodos de una interfaz debe declararse como abstracta
- Útiles para dar soporte a la **herencia múltiple** en Java
  - Una clase puede heredar de una única clase
  - Puede implementar más de una interfaz
  - Una interfaz puede extender más de una interfaz

Interfaz `Amortizable`



Una clase implementa una interfaz

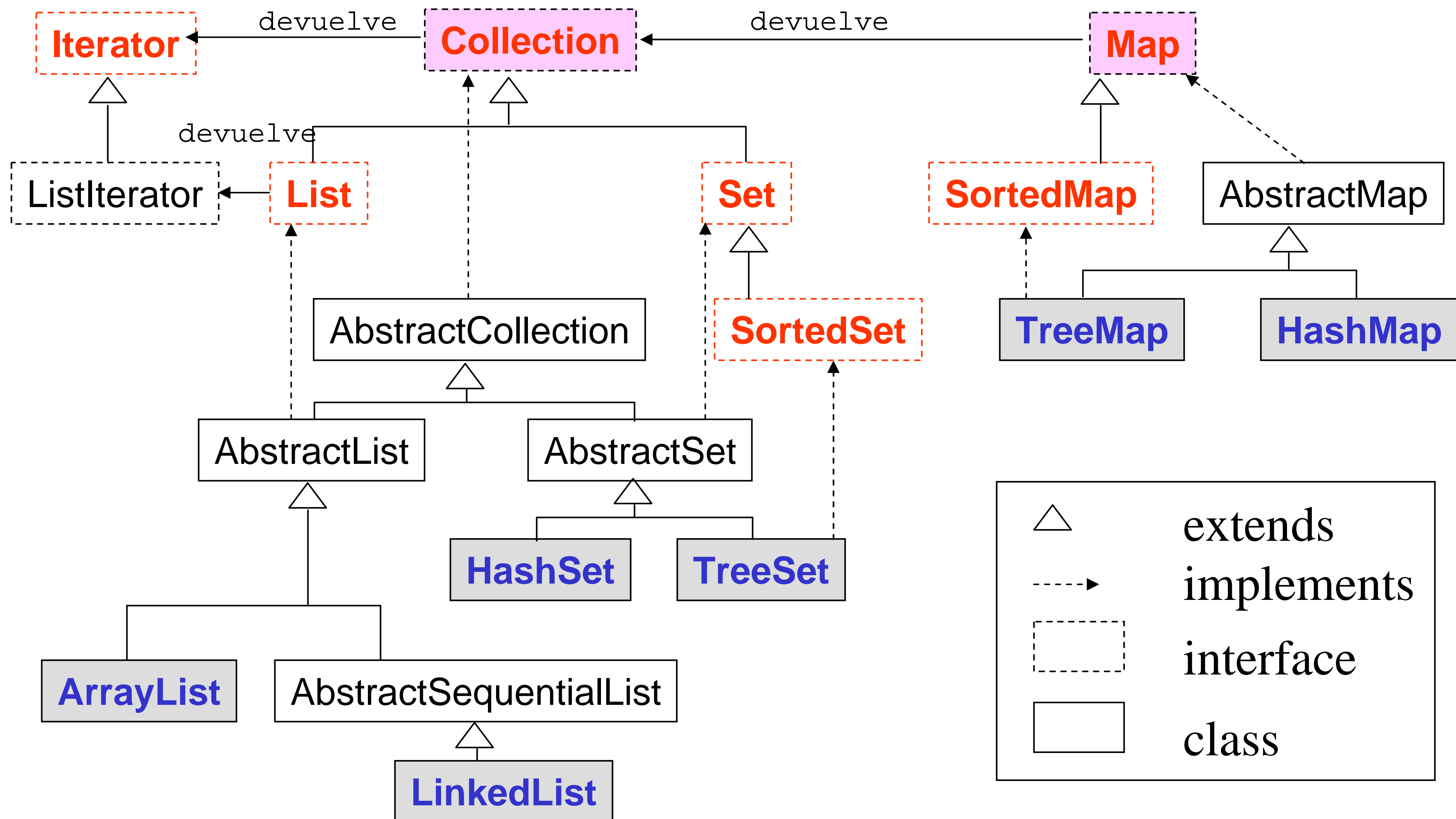
```java
public class DepositoEstructurado extends Deposito
                                        implements Amortizable{
…

  public boolean amortizar(double cantidad) {
      if (cantidad > capitalVariable)
            return false;
      capitalVariable = capitalVariable - cantidad;
      return true;
  }
}
```

```java
Amortizable a = new DepositoEstructurado(…);
a.amortizar(600.0);
```

Colecciones en Java

- Las colecciones en Java son un ejemplo destacado de implementación de código reutilizable utilizando un lenguaje orientado a objetos.
- Todas las colecciones son genéricas e iterables.
- Los tipos abstractos de datos se definen como interfaces.
- Se implementan clases abstractas que permiten factorizar el comportamiento común a varias implementaciones.
- Un mismo TAD puede ser implementado por varias clases → `List: LinkedList, ArrayList`

Colecciones en Java (java.util)



Iteradores en Java

- Las colecciones de Java son iterables.
- El recorrido **for each** sólo es aplicable sobre objetos **iterables**

  → Evita tener que manejar explícitamente el objeto iterador.
- La **implementación de un iterador** requiere crear una clase responsable de llevar el estado de la iteración → **Clases internas**

Iteradores en Java

- Java proporciona la interfaz **Iterable<T>** que debe ser implementada por aquellas clases sobre las que se pueda iterar:

```
public interface Iterable<T> {

        Iterator<T> iterator();
}
```

- A los objetos iterables se les exige que creen objetos iterador (*iterator*) para realizar la iteración.

Iteradores en Java

- Interfaz **Iterator<T>**:
  - **hasNext**(): indica si quedan elementos en la iteración.
  - **next**(): devuelve el siguiente elemento de la iteración.
  - **remove**(): elimina el último elemento devuelto por el iterador.

```
public interface Iterator<E> {
        boolean hasNext();
        E next();
        void remove();
}
```

Implementación del iterador

```
public class Lista<T> implements Iterable<T>{
    …
      public Iterator<T> iterator() {
          return new Itr();
       }
      private class Itr implements Iterator<T> {
          int cursor = 0;
          public boolean hasNext() {
                return cursor != size();
          }

          public T next() {
               return get(cursor);
           }

          public void remove() { …}
       }
}
```

Clases internas

- Clase que se declara dentro de otra
- Los objetos de esta clase están ligados al objeto de la clase contenedora donde se crean
- La clase interna tiene visibilidad sobre las declaraciones de la clase contenedora
- Los objetos de la clase interna pueden acceder a los atributos y métodos del objeto contenedor como si fueran propios.
  - Ejemplo: llamada a `size` y `get` en el iterador
- Los objetos de la clase interna se crean dentro de un método de instancia de la clase que los contiene
  - Ejemplo: método `iterator`

Uso del iterador

- Utilizamos un iterador para imprimir los morosos:

```
List<Cuenta> cuentas = new LinkedList<Cuenta>();
...
public void imprimeMorosos(){
      Iterator<Cuenta> iterador = cuentas.iterator();

      while (iterador.hasNext()){
            Cuenta cta = iterador.next();
            if (cta.isEnRojo())
                  System.out.println(cta.getTitular());

      }
}
```

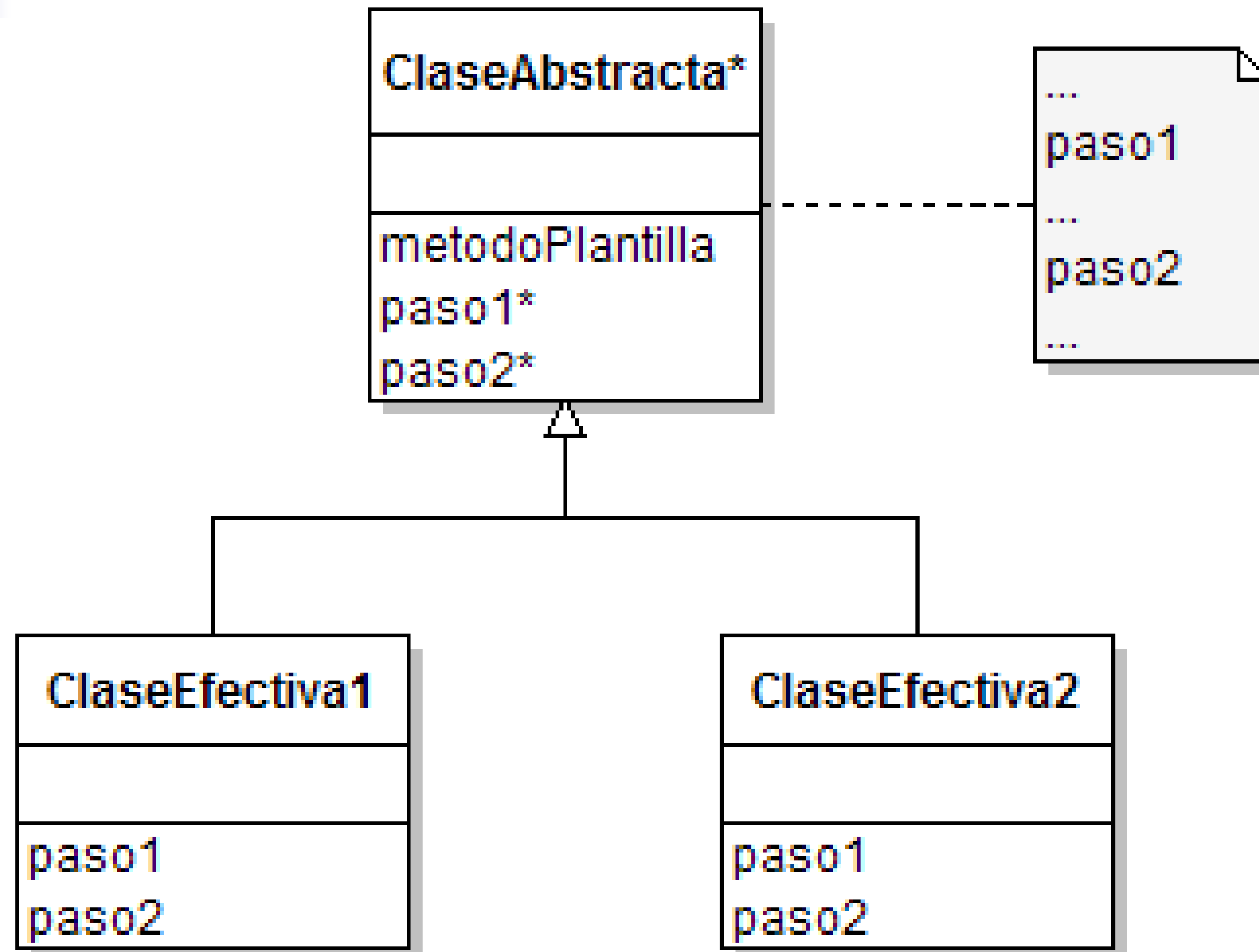
Patrones de diseño

- Soluciones a problemas que aparecen recurrentemente en la POO.
- **Diseño de clases reutilizable** que se puede adaptar a diferentes situaciones.
- La definición de un patrón de diseño es independiente del lenguaje que se utilice para implementarlo.
- Vamos a estudiar:
  - Método Plantilla
  - Patrón Estrategia
  - Patrón Composite

Método Plantilla

- Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos.
- Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.
- El algoritmo factoriza un comportamiento común a un conjunto de subclases.
- El algoritmo se implementará en un método efectivo que hará uso de otros métodos abstractos de la clase (clases parcialmente abstractas).

Método Plantilla



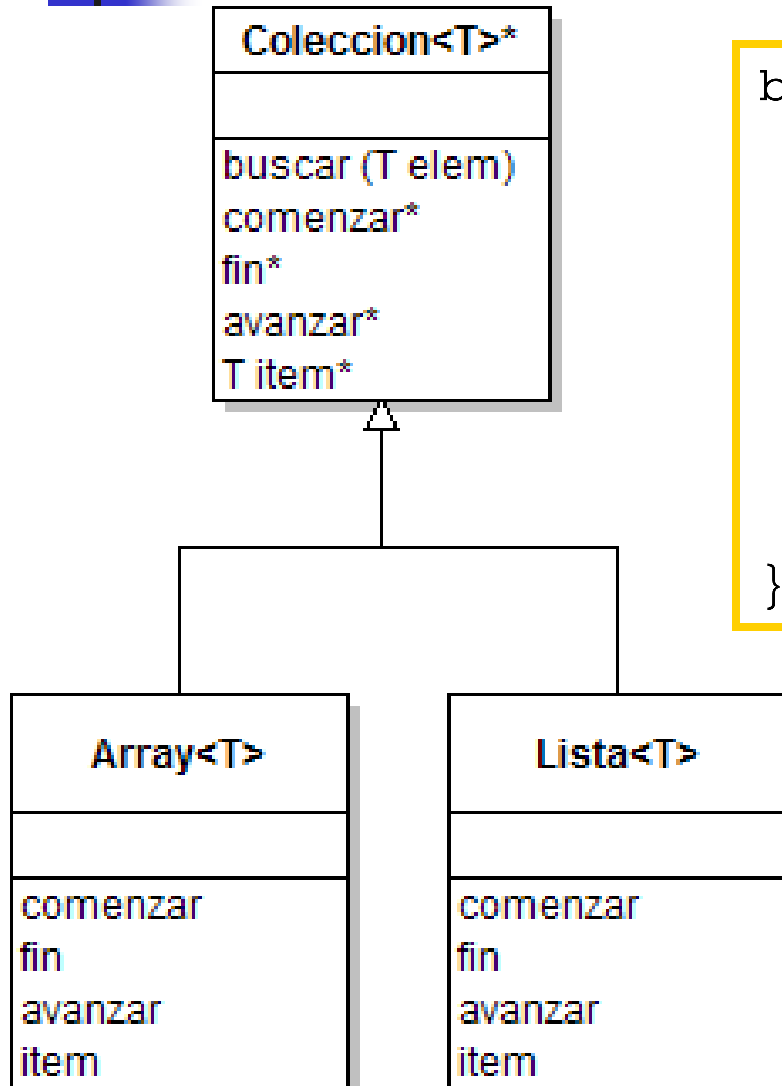
Método Plantilla

- En la aplicación bancaria a los titulares de los productos financieros se les manda la información fiscal
- La información fiscal de un producto financiero contiene: el nombre del titular, el producto financiero contratado, el beneficio obtenido y el valor de los impuestos cobrados.
- El método que genera el informe fiscal es una plantilla de código
  - Factoriza el comportamiento común a todos los productos financieros

Método plantilla

```
public abstract class ProductoFinanciero {

   …

   public double getImpuestos() {
      return getBeneficio() * 0.18;
   }

   public String getInforme(){
     return "Titular:" + titular +
            "producto contratado: "+getTipo()+
             "beneficios: "+ getBeneficio()+
             "impuestos: "+ getImpuestos();
    }

    public abstract String getTipo();
    public abstract double getBeneficio();
}
```

Ejemplo de Método Plantilla




```
boolean buscar (T x){
   comenzar();
   while (!fin() && !encontrado){
          if (x.equals(item()))
                encontrado = true;
          avanzar();
   }
   return encontrado;
}
```

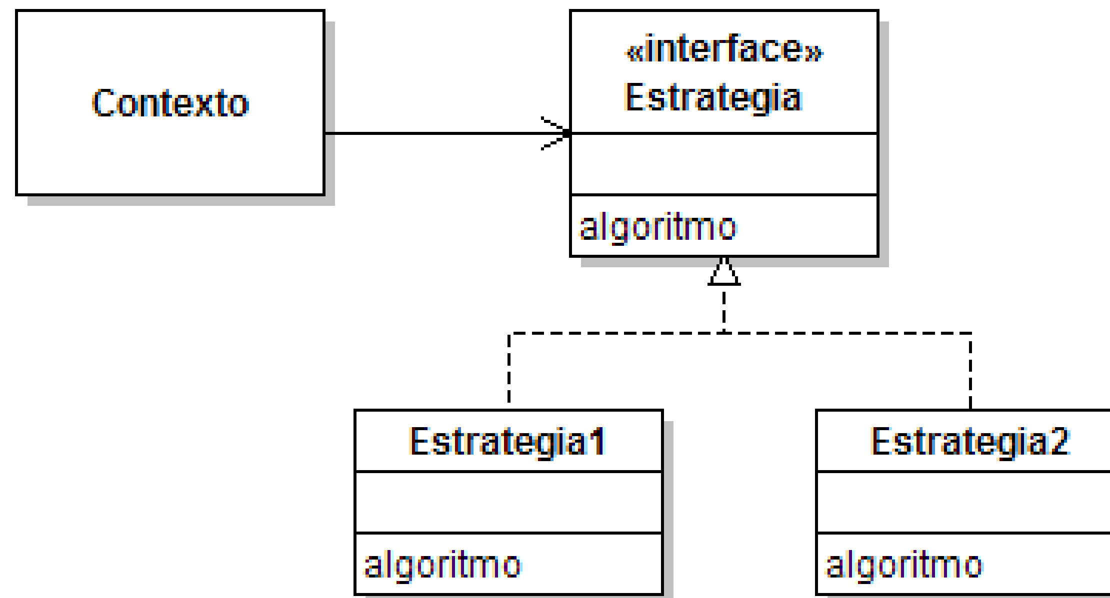
NOTA:
- Ejemplo de código reutilizable del Tema1
- Recordad los criterios de reutilización

Método Plantilla

```
public abstract class Coleccion<T> {

   public boolean buscar (T x){
        comenzar();
        while (!fin() && !encontrado){
              if (x.equals(item()))
                     encontrado = true;
              avanzar();
        }
        return encontrado;
   }
   public abstract void comenzar();
   public abstract boolean fin();
   public abstract T item();
   public abstract void avanzar();
}
```

Patrón Estrategia

- Permite definir una familia de algoritmos que se pueden intercambiar en tiempo de ejecución
  - Por ejemplo: algoritmos de ordenación, criterios de selección, etc.

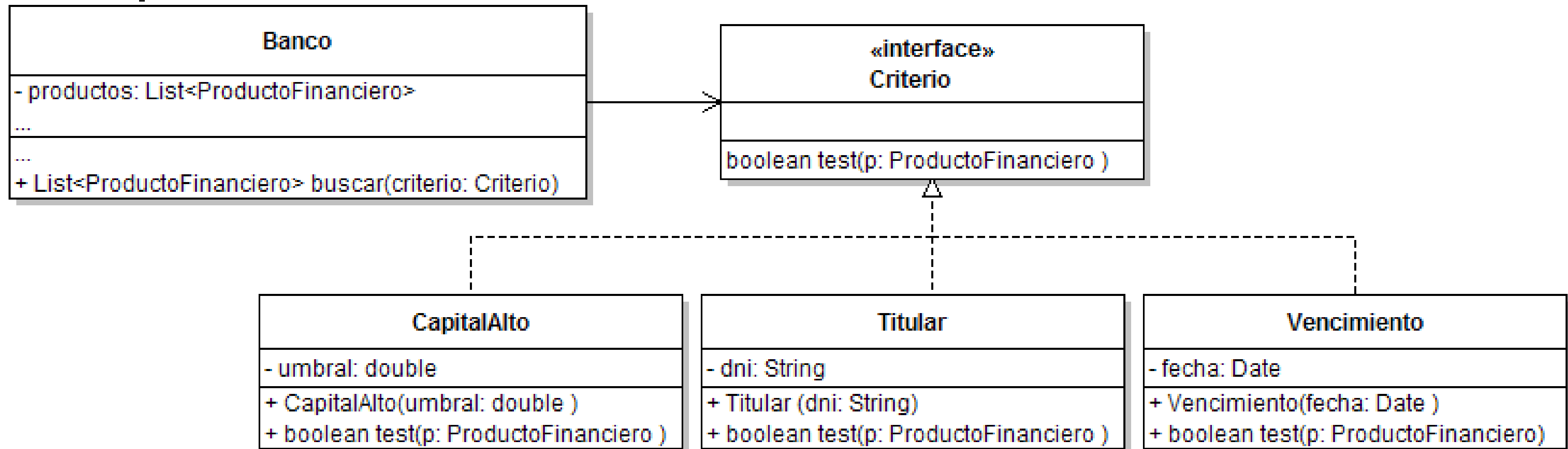


- Favorece la extensibilidad (definición de nuevos algoritmos)

Ejemplo Patrón Estrategia

- En la aplicación bancaria, supongamos que la clase `Banco` almacena la lista de todos los productos financieros
- ¿Cómo podemos implementar un método de búsqueda que devuelva todos los productos financieros que cumplen una determinada condición?
- La condición de búsqueda puede cambiar de una búsqueda a otra
  - Todos los depósitos con el capital mayor de 10000
  - Todos los productos contratados por "Pepito Pérez"
  - Todos los depósitos que vencen en una determinada fecha
  - ...
- En Java no es posible pasar una función como parámetro.

Aplicación del Patrón Estrategia



- La interfaz `Criterio` encapsula el algoritmo que representa el criterio de búsqueda
  - El método `test` evalúa si el `ProductoFinanciero` que se le pasa como parámetro cumple o no el criterio de búsqueda
- Cada criterio de búsqueda se implementa en una clase que implementa la interfaz `Criterio`.

Implementación Patrón Estrategia

```
public interface Criterio {
    boolean test(ProductoFinanciero producto);
}
```

```
public class Banco {
   private List<ProductoFinanciero> productos;
   …
   public List<ProductoFinanciero> buscar (Criterio criterio){
       List<ProductoFinanciero> result =
                          new LinkedList<ProductoFinanciero>();

       for (ProductoFinanciero p: productos)
             if (criterio.test(p))
                   result.add(p);

       return result;
   }
}
```

Implementación Patrón Estrategia

```
public class CapitalAlto implements Criterio {
   private double umbral;

   public CapitalAlto(double umbral){
        this.umbral = umbral;
   }

   public boolean test(ProductoFinanciero producto) {

        if (producto instanceof Deposito){
               Deposito deposito = (Deposito)producto;
               return deposito.getCapital() > umbral;
        }
        else
               return false;

   }
}
```

Implementación Patrón Estrategia

- Para buscar los productos que cumplen un determinado criterio de búsqueda habrá que pasar como parámetro al método `buscar` un objeto de una clase que implemente el criterio.

```
Banco banco = new Banco();
List<ProductoFinanciero> depositosAltos;

Criterio criterio = new CapitalAlto(10000.0);
depositosAltos = banco.buscar(criterio);
```

Estrategias anónimas

- ¿Tiene sentido crear una clase por cada criterio?
- Solución basada en clases anónimas

```
banco.buscar(new Criterio(){
   public boolean test (ProductoFinanciero p){
      if (p instanceof Deposito){
         Deposito deposito = (Deposito)p;
         return deposito.getCapital() > 10000.0;
      }
      else
         return false;
   }
});
```

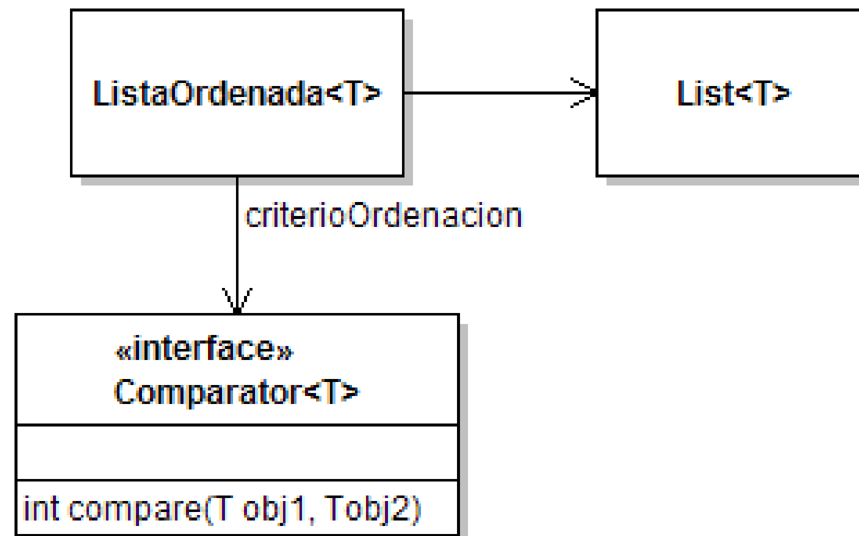
Estrategias anónimas

- Solución basada en clases anónimas que necesita un parámetro

```
public List<ProductoFinanciero> buscarUmbral
                                    (final double umbral){
    return buscar(new Criterio(){
        public boolean test(ProductoFinanciero producto) {
            if (producto instanceof Deposito){
                Deposito deposito = (Deposito)producto;
                return deposito.getCapital() > umbral;
            }
            else
                return false;
        }
    });
}
```

Ejercicio: Lista ordenada

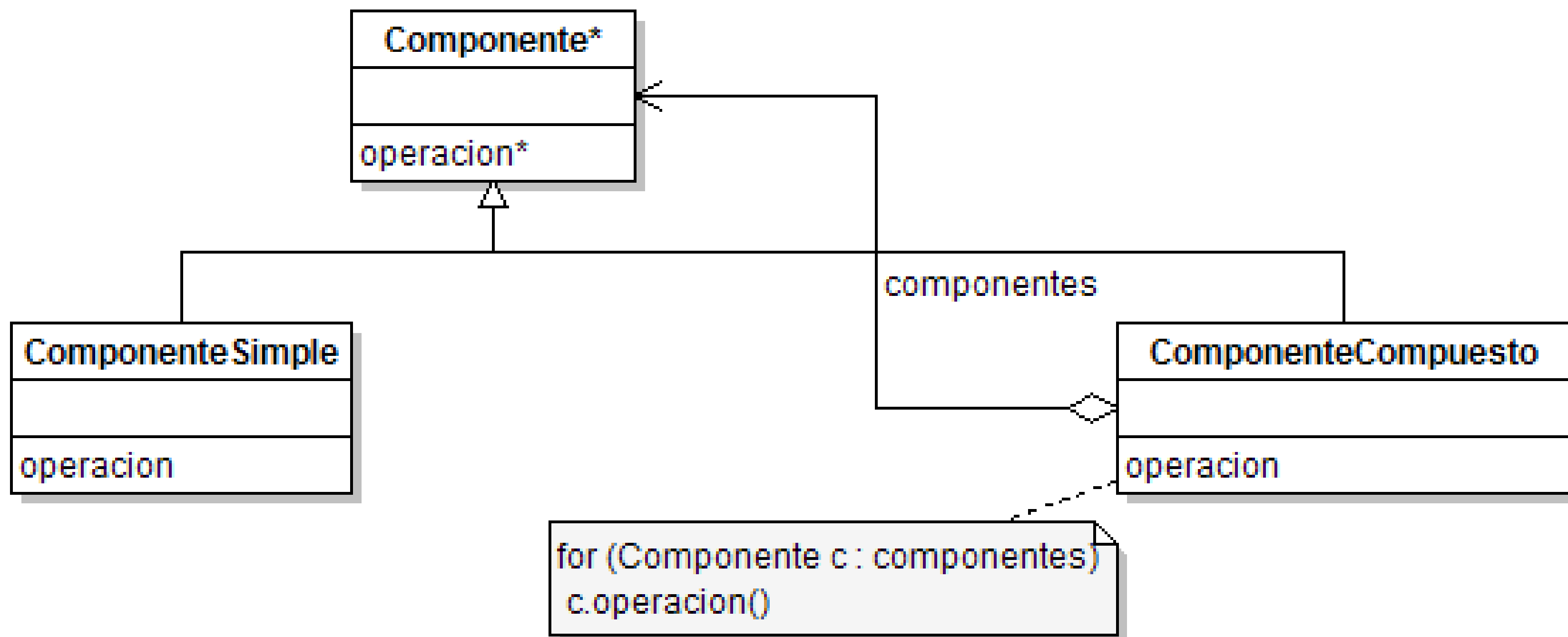
- Implementar una clase que represente una lista ordenada
- El criterio de ordenación se establece en el momento de la creación de la lista
- El método `add` inserta los elementos de acuerdo al orden establecido



- Aplicarlo para construir una lista ordenada de productos financieros por orden alfabético de los titulares.

Patrón Composite

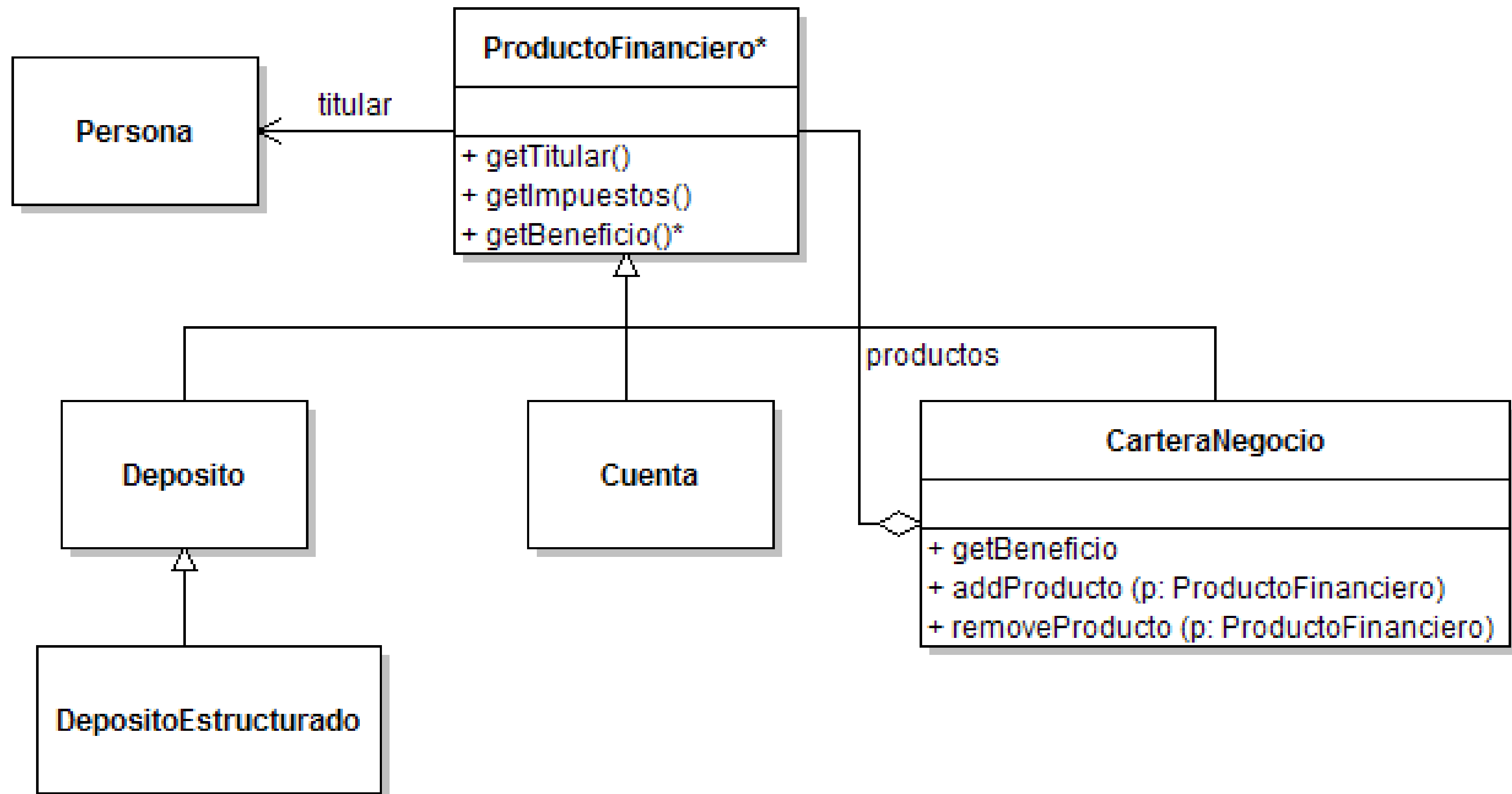
- Permite representar objetos compuestos.
- Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.



Ejemplo Patrón Composite

- Supongamos que en el banco existen carteras de negocios.
- Una cartera de negocios **es un** producto financiero
- Una cartera de negocios **está formada por** otros productos financieros incluidas otras carteras
- Una cartera de negocios es un producto financiero compuesto
  - El beneficio es la suma de los beneficios de todos los productos que contiene

Ejemplo Patrón Composite



Clase CarteraNegocio

```
public class CarteraNegocio extends ProductoFinanciero {
   private List<ProductoFinanciero> productos;

   public CarteraNegocio(Persona titular){
        super(titular);
        productos = new LinkedList<ProductoFinanciero>();
   }

    public void addProducto(ProductoFinanciero p) { … }

    @Override
   public double getBeneficio() {
        double total = 0;

        for (ProductoFinanciero producto : productos)
              total+=producto.getBeneficio();

        return total;
   }
}
```

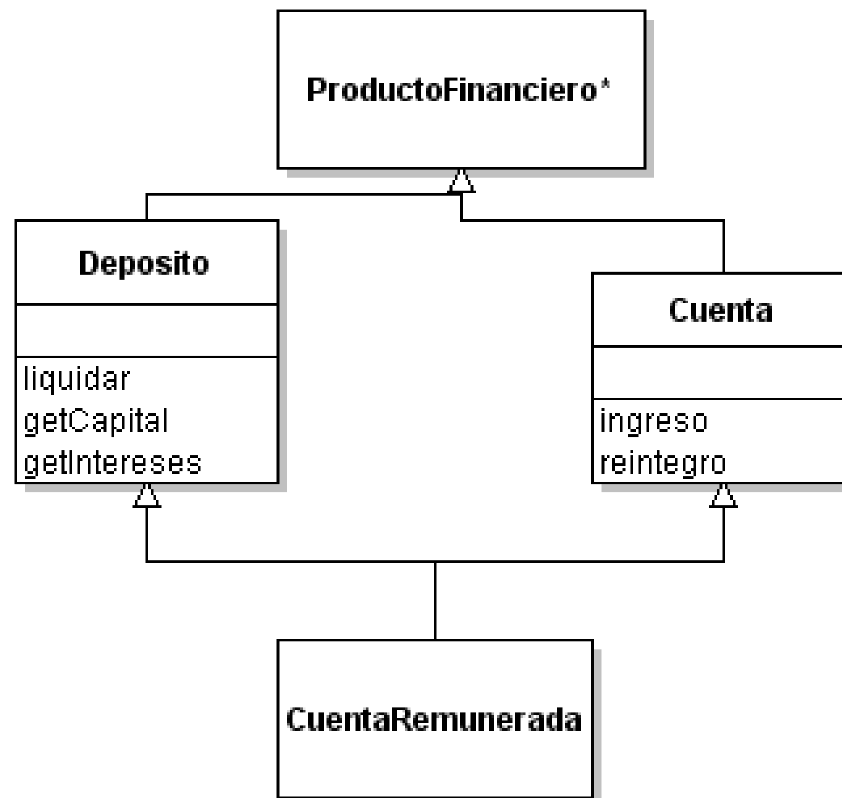
Herencia múltiple

- ¿Qué proporciona la herencia?
  - Posibilidad de reutilizar el código de otra clase (perspectiva módulo).
  - Hacer que el tipo de la clase sea compatible con el de otra clase (perspectiva tipo).

- En Java la **herencia es simple**.
- La limitación de tipos impuesta por la herencia es superada con el uso de interfaces.
- Sin embargo, sólo es posible reutilizar código de una clase.

Herencia múltiple

- **Motivación**:
  - **Cuenta Remunerada**: es una cuenta bancaria que también ofrece rentabilidad como un depósito.



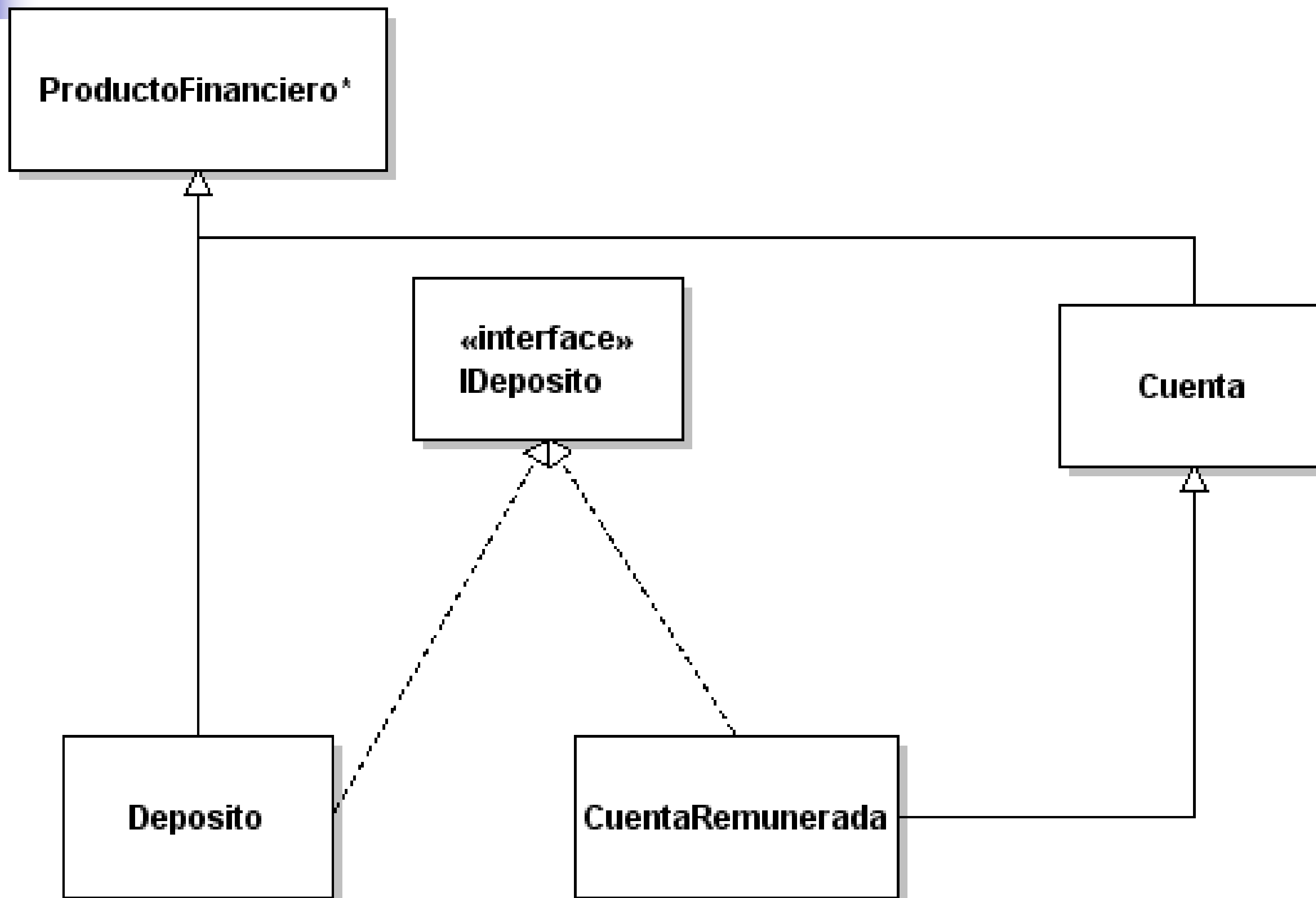
Herencia múltiple

- En Java sólo podemos heredar de una clase:
  - Elegimos la clase `Cuenta` como clase padre.

- ¿Cómo conseguimos que `CuentaRemunerada` sea compatible con el tipo `Deposito`?
  - Definimos la **interfaz** **`IDeposito`** y hacemos que la clase `Deposito` implemente la interfaz.

- `CuentaRemunerada` también implementa la interfaz `IDeposito`.

Interfaz IDeposito



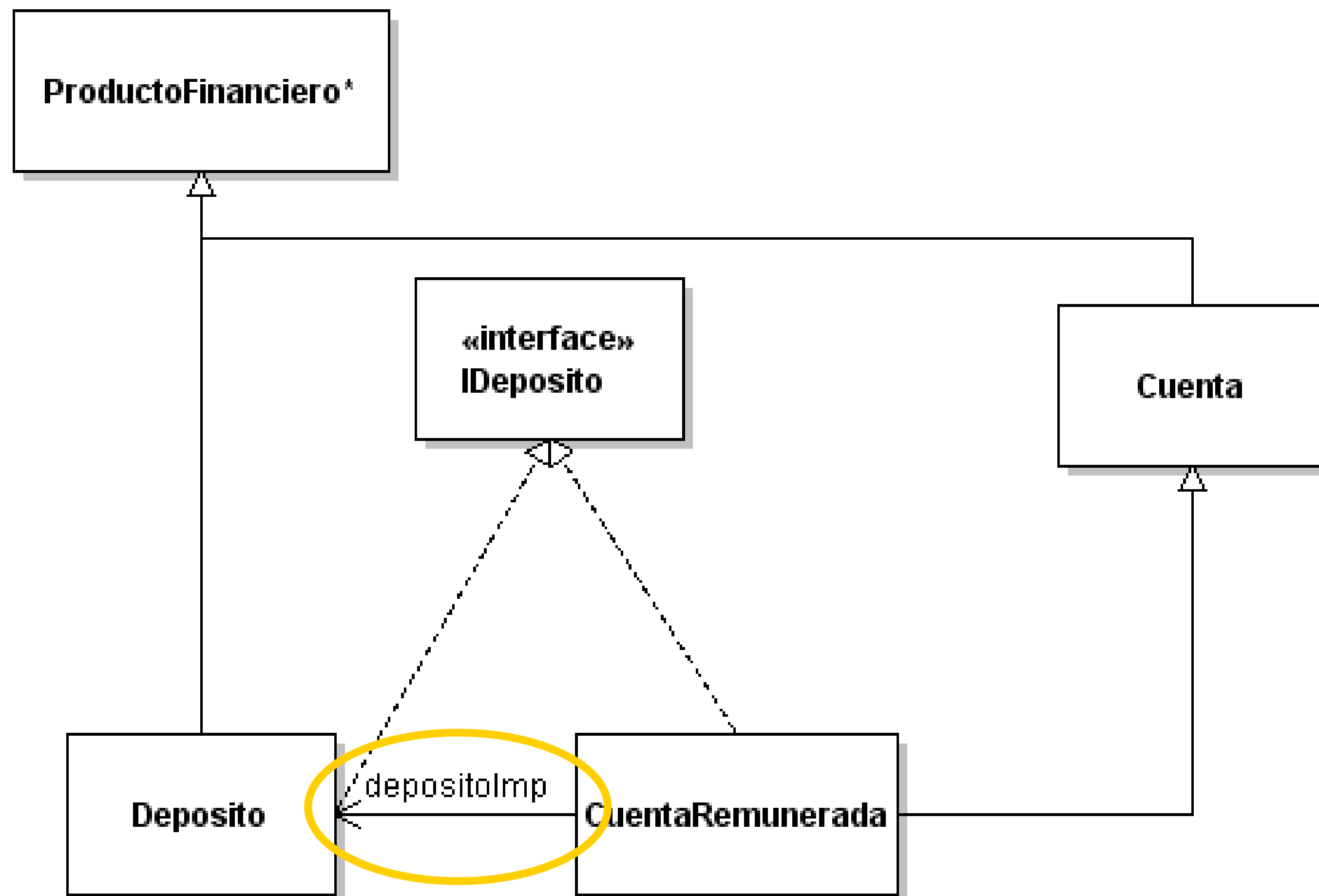
Herencia múltiple en Java

```
public interface IDeposito
{
    double liquidar();
    double getIntereses();
    double getCapital();
    int getPlazoDias();
    double getTipoInteres();
    Persona getTitular();
}
```

- `CuentaRemunerada` es compatible con el tipo depósito definido por la interfaz `IDeposito`
- ¿Cómo podemos reutilizar la implementación de la clase `Deposito`?

Solución 1: Relación clientela

- Si no es necesario extender la definición de la clase `Deposito`, establecemos una relación de clientela



Solución 1: Relación de clientela

```
public class CuentaRemunerada extends Cuenta
                                        implements IDeposito {
  …
  private Deposito depositoImpl;

  public CuentaRemunerada(Persona titular,
                  int saldoInicial, double tipoInteres) {

     super(titular, saldoInicial);
       // Liquidacion mensual de intereses
     depositoImpl =
       new Deposito(titular,saldoInicial,30,tipoInteres);

  }
```

Solución 1: Relación de clientela

```
    // Implementación IDeposito
    @Override
    public double getCapital() {
        return depositoImpl.getCapital();
    }
    @Override
    public double getIntereses() {
        return depositoImpl.getIntereses();
    }
    …
}//fin de la clase CuentaRemunerada
```

- La clase `CuentaRemunerada` delega la implementación de los métodos de la interfaz `IDeposito` en la clase `Deposito`

Solución 2: clases internas

- Puede que necesitemos reutilizar la implementación de la clase `Deposito` pero extendiendo la definición original

- Definimos una clase interna que herede de `Deposito`:
  - La clase interna puede redefinir métodos de la clase `Deposito`
  - La clase interna puede aplicar métodos de la clase contenedora (`CuentaRemunerada`)

Herencia Múltiple

```
public class CuentaRemunerada extends Cuenta
                                  implements IDeposito {
     …
   private double saldoMedioUltimoMes() { … }
   private class DepositoImpl extends Deposito {

       DepositoImpl(Persona titular, double capital,
                    int plazoDias, double tipoInteres) {

           super(titular, capital, plazoDias, tipoInteres);
       }

       @Override
       public double getCapital() {
           return saldoMedioUltimoMes();
       }
   }
```

Solución 2: clases internas

```
public class CuentaRemunerada extends Cuenta
                                          implements IDeposito {
  …
  private Deposito depositoImpl;

  public CuentaRemunerada(Cliente titular,
                    int saldoInicial, double tipoInteres) {

     super(titular, saldoInicial);
       // Liquidacion mensual de intereses
     depositoImpl =
       new DepositoImpl(titular,saldoInicial,30,tipoInteres);
   }
```

Solución 2: clases internas

```
    // Implementación IDeposito
    @Override
    public double getCapital() {
        return depositoImpl.getCapital();
    }
    @Override
    public double getIntereses() {
        return depositoImpl.getIntereses();
    }
    …
}//fin de la clase CuentaRemunerada
```

- La clase `CuentaRemunerada` delega la implementación de los métodos de la interfaz `IDeposito` en la clase interna

Consejos de diseño de herencia

- Hay que poner las operaciones y campos comunes en la superclase
- No se deben utilizar atributos protegidos
- Hay que utilizar la herencia para modelar la relación "es_un"
- No se debe utilizar la herencia salvo que todos los métodos heredados tengan sentido
- No hay que modificar la semántica de un método en la redefinición
- Hay que utilizar el polimorfismo y no la información relativa al tipo