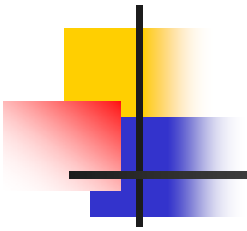
Tema 3: Herencia en Java

Programación Orientada a Objetos

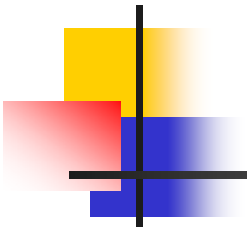
Curso 2008/2009

Begoña Moros Valle



Índice

- Introducción
- Herencia y Ocultamiento de la Información
- Redefinición de características
- Herencia y creación
- Polimorfismo
- Herencia y Sistema de tipos
- Ligadura dinámica
- Clase `Object`
- Genericidad y Herencia
  - Genericidad restringida
- Clases abstractas
  - Clases comportamiento: Iteradores
- 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:

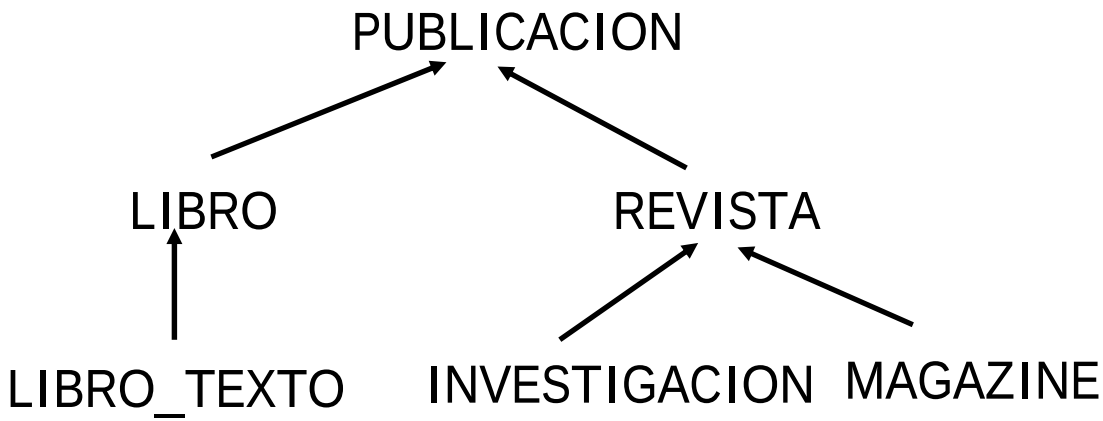
  - "Principio abierto-cerrado" y "Principio Elección Única"
  - Estructuras de datos polimórficas.

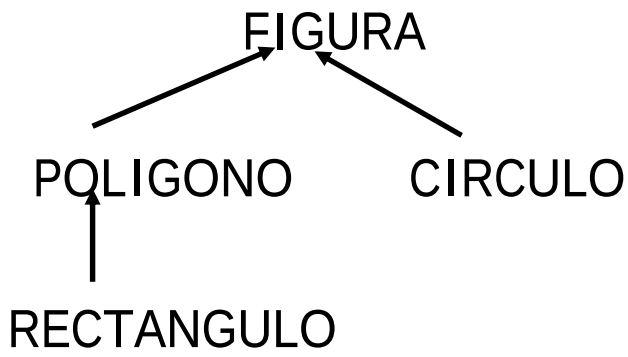
Introducción

- 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**

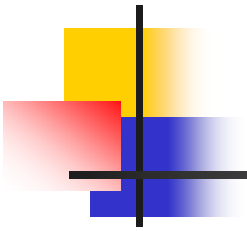
Jerarquías de herencia

- 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

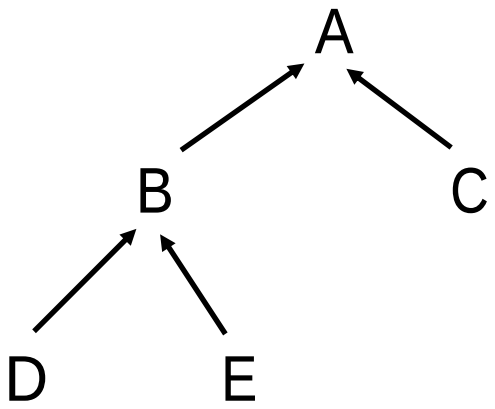
- Si una clase **B hereda** de otra clase **A** entonces:
  - B incorpora la estructura (atributos) y comportamiento (métodos) de la clase A
  - B puede incluir adaptaciones:
    - B puede **añadir** nuevos **atributos**
    - B puede añadir nuevos **métodos**
    - B puede **redefinir métodos**
   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

El proceso de herencia es transitivo

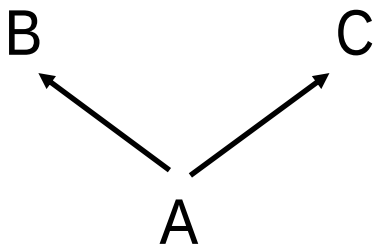
- B hereda de A
  - A es la superclase y B la subclase
- C hereda de B y A

- 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

Polígonos y Rectángulos

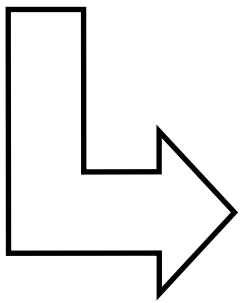
- Tenemos la clase **`Poligono`** y necesitamos representar rectángulos:

¿Debemos crear la clase **`Rectangulo`** partiendo de cero?

Podemos aprovechar la existencia de similitudes y particularidades entre ambas clases

Polígonos y Rectángulos

- Un rectángulo tiene muchas de las características de un polígono (*rotar*, *trasladar*, *vértices*,..)
- Pero tiene características especiales (*diagonal*) y propiedades especiales (*4 lados, ángulos rectos*)
- Algunas características de polígono pueden implementarse más eficientemente *(perímetro)*



```
class Rectangulo extends Poligono{

      ...Características específicas

}
```

Clase Polígono 1/3

```
public class Poligono {
   //Un polígono se implementa como una lista de
    //puntos sucesivos
   private List<Punto> vertices;
   private int numVertices;

   public Poligono(List<Punto> puntos) {
       vertices = new LinkedList<Punto>(puntos);
       numVertices = puntos.size();
   }

   public int getNumVertices(){
       return numVertices;
   }

   public void rotar (Punto centro,
                              double angulo){…}
   public void trasladar (double a, double b){…}
   public void visualizar(){…}
   public double perimetro(){…}
   …

}
```

Clase Polígono 2/3

```
/**
 * Desplaza a horizontalmente y b verticalmente
 */
public void trasladar (double a, double b){
   for (Punto pto : vertices)
       pto.trasladar(a,b);
}


/**
 * Rota el ángulo alrededor del centro
 */
public void rotar (Punto centro, double angulo){
   for (Punto pto : vertices)
       pto.rotar(centro, angulo);
}
```

Clase Polígono 3/3

```
/**
 * Suma las longitudes de los lados
 */

public double perimetro(){
   double total = 0;
   Punto anterior;
   Punto actual = vertices.get(0);

   for (int index = 1; index < numVertices; index++){
       anterior = actual;
       actual = vertices.get(index);
       total = total + actual.distancia(anterior);
   }

   total=total+actual.distancia(vertices.get(0));

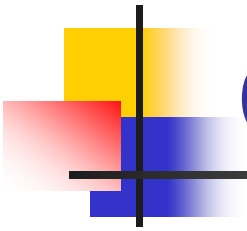
   return total;
}
```

Clase Rectangulo

```
public class Rectangulo extends Poligono{
    private double lado1, lado2;     //Nuevos atributos
    private double diagonal;

    public Rectangulo(List<Punto> vertices,
                      double lado1, double lado2){
        super(vertices);
        this.lado1 = lado1;
        this.lado2 = lado2;
    }

    @Override //Redefine perímetro
    public double perimetro(){
        return (2*(lado1 + lado2 ));
    }
    public double diagonal() {//Nuevos métodos
       return Math.sqrt(lado1*lado1 + lado2*lado2);
    }
}
```

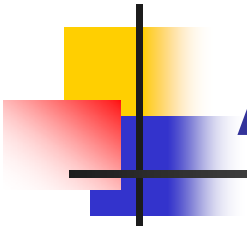


Clase Rectángulo

- Rectángulo hereda todos los métodos definidos en Polígono, no hay que repetirlos:
  - `Rectangulo.trasladar(0.0,0.0);`
- Añade nuevas características:
  - Atributos: `lado1, lado2`
  - Constructor que invoca al constructor de `Poligono`
  - Métodos: `diagonal`
- Redefine características heredadas:
  - Reemplaza la implementación de `perímetro`

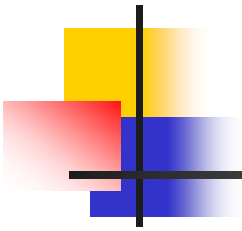
Herencia y creación

- 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



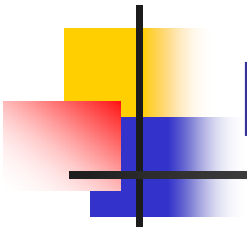
Acceso protegido

- 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



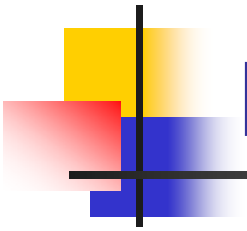
Resumen modificadores de acceso

- 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

- La redefinición reconcilia la reutilización con la extensibilidad:
  - “Es raro reutilizar un componente software sin necesidad de cambios”
- 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**:
  - Mejorar implementación.
    - Ej: redefinir `perímetro` en la clase `Rectangulo`.
  - Otra diferente (aunque con la misma semántica).
    - Ej: el método `dibujar` en la jerarquía de `Figura`
- **Refinamiento**:
  - Método del padre + acciones específicas
    - Ej: `ingreso` en una `CuentaAhorro` calcula un beneficio

Refinamiento: `super`

- Si un método redefinido refina el comportamiento del método original puede necesitar hacer referencia a este comportamiento

- **`super`** se utiliza para invocar a un método de la clase padre:
  - `super.met();`
  - Se “rompe” la ligadura dinámica
  - No sólo se utiliza en los métodos redefinidos

Ejemplo super

```
public class CuentaAhorro extends Cuenta{
   …
   @Override //Refina el comportamiento heredado

   public void ingresar(double cantidad){

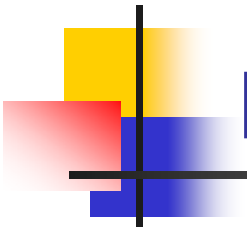
//Hace lo mismo que el método de la clase padre
       super.ingresar(cantidad);

   //Además hace cosas propias de la CuentaAhorro
       beneficio = cantidad * PORCENTAJE_BENEFICIO;
   }
}
```

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** (tipo 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() {…}`

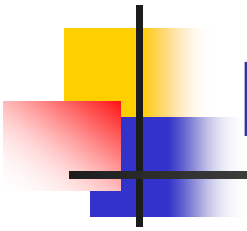


Restringir la herencia y redefinición

- 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?



Polimorfismo

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

- Formas de polimorfismo:
  - 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 instancias de diferentes clases.
- El conjunto de clases a las que se puede hacer referencia está **restringido por la herencia**
- Importante para escribir código genérico
- Sea las declaraciones:

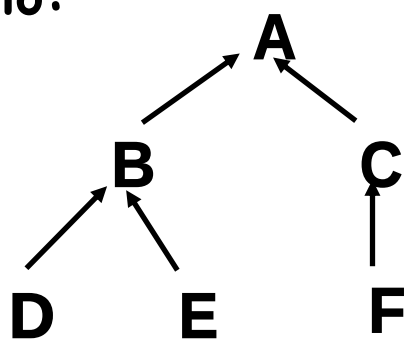
```
X ox; metodo(Y oy);
```

  - En un lenguaje con monomorfismo (Pascal, Ada, ..) en tiempo de ejecución `ox` y `oy` denotarán valores de los tipos `X` e `Y`, respectivamente.
  - En un lenguaje con polimorfismo (Java, ..) en tiempo de ejecución `ox` y `oy` podrán estar asociados a objetos de varios tipos diferentes:
    - **tipo estático** vs. **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:
  - Conjunto de posibles tipos dinámicos de una entidad

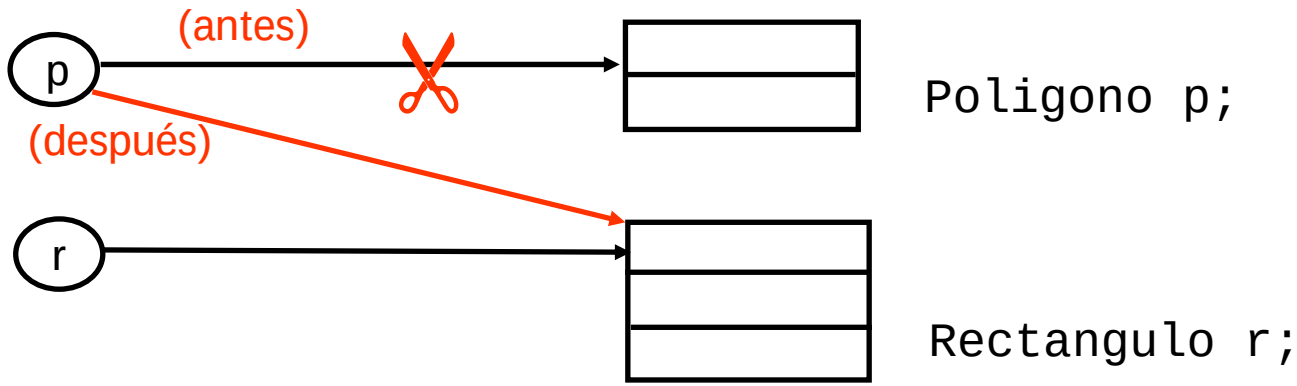
Ejemplo:




```
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}
```

Entidades y rutinas polimorfas

Conexión polimorfa: el origen y el destino tiene tipos diferentes



a) asignación:

```
p = r;
```

-- p es una **entidad polimorfa**
(*polimorfismo de asignación*)

b) paso de parámetros:

```
void f (Poligono p){
...
}
```

-- f es un **método polimórfico**
(*polimorfismo puro*)

- **Sólo** se permite para entidades destino de **tipo referencia**

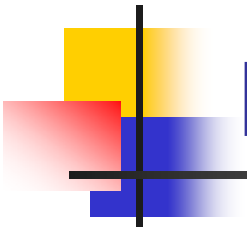
Polimorfismo de asignación

```
1. Poligono poligono = new Poligono(…);
2. Rectangulo rectangulo = new Rectangulo(…);
3. poligono = rectangulo;
  //Asignación polimórfica
```

- El tipo estático no cambia:
  - 1. El tipo estático de `poligono` es la clase `Poligono`
  - 1. El tipo dinámico de `poligono` es `Poligono`
- El tipo dinámico cambia en tiempo de ejecución:
  - 3. El tipo dinámico de `poligono` es `Rectangulo`
  - 3. El tipo estático de `poligono` sigue siendo `Poligono`

Polimorfismo puro vs. Sobrecarga

- Funciones sobrecargadas ≠ funciones polimórficas
- **Sobrecarga**:
  - 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.
- **Funciones polimórficas**:
  - 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

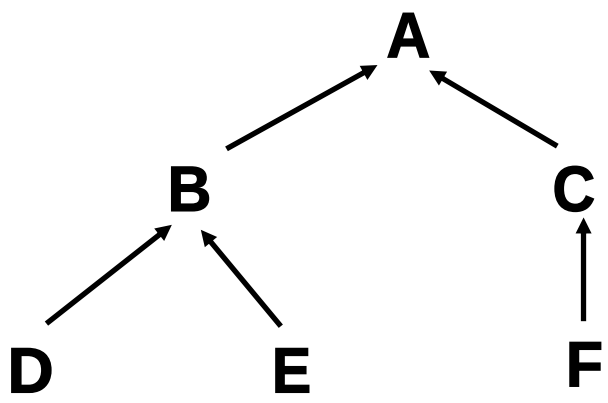
- **Polimorfismo puro:**
  - Un único código con distintas interpretaciones

```
public void recortar (Poligono p){
        …
}
```

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

```
public void recortar (Poligono p){
… }
public void recortar (Rectangulo r){
… }
```

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

Reglas básicas de consistencia

Definición: compatibilidad o conformidad de tipos

Un tipo **`U`** es compatible o conforme con un tipo **`T`** sólo si la clase base de `U` es un descendiente de la clase base de `T`.

- Por ejemplo, `Rectangulo` es compatible con `Poligono`

Regla de compatibilidad de tipos

Una conexión con origen **`y`** y destino **`x`** (esto es, una asignación **`x=y`**, o invocación **`m(..,y,..)`** a un método **`m(.., T x,..)`** es válido solamente si el tipo de **`y`** es compatible con el tipo de **`x`**.

Reglas básicas de consistencia

Regla de llamada a una característica

En una llamada a una característica **`x.f`** donde el tipo de `x` se basa en una clase `C`, la característica `f` debe estar definida en uno de los antecesores de `C`.

Luego, sean las declaraciones

```
Poligono p ; Rectangulo r;
```

Mensajes legales:	**Mensajes ilegales:**

`p.perimetro(); p.getVertices() ;`
`p.trasladar(..); p.rotar (..);`
`r.getDiagonal();r.getLado1();`
`r.getLado2();`
`r.getVertices(); r.trasladar(..);`
`r.rotar (..); r.perimetro();`
	`p.getLado1();`
	`p.getLado2();`
	`p.getDiagonal();`

Reglas básicas de consistencia

Regla de validez de un mensaje

Un mensaje **`ox.met(y),`** supuesta la declaración **`X ox;`** será legal si:

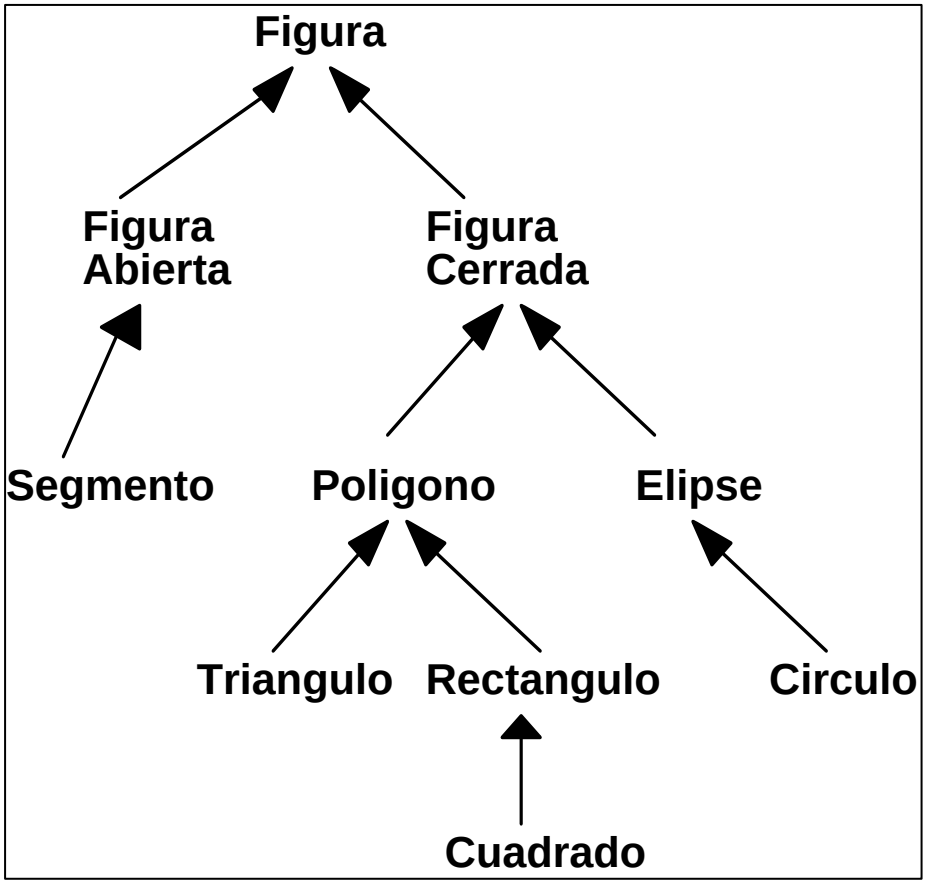
i) **`X`** incluye un método con nombre **`met`**,

ii) los argumentos son compatibles con los parámetros y coinciden en número

iii) **`met`** está disponible (es visible) para la clase que incluye el mensaje.

Ejemplo

```
Poligono p; Rectangulo r; Triangulo t;... double x;
```



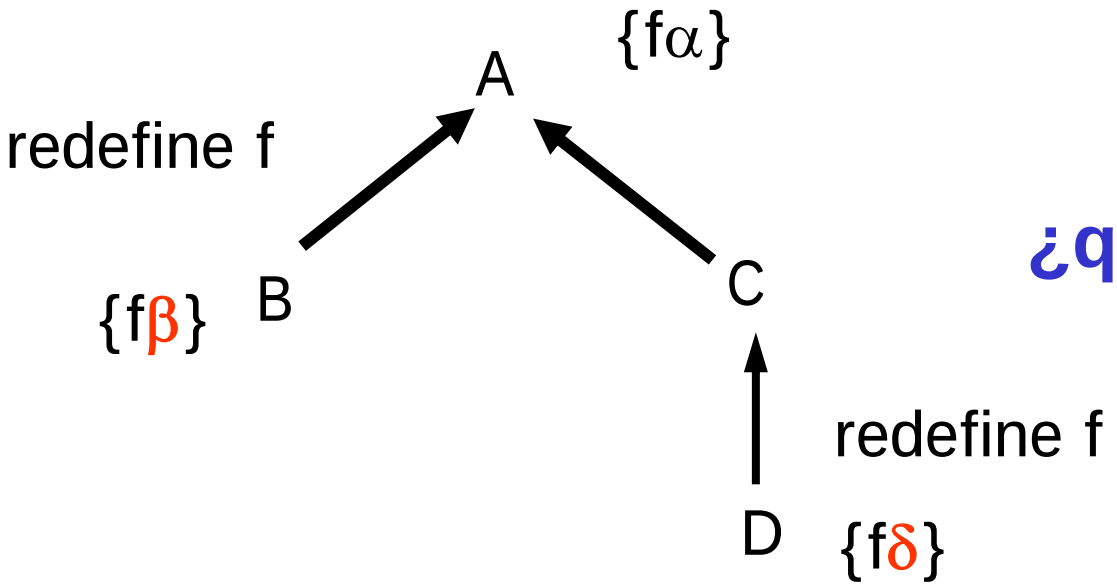
SERÍA CORRECTO EL CODIGO

```
x = p.perimetro();
x = r.perimetro();
x = r.getDiagonal();
if (test) p = r
else p = t
x = p.perimetro();
```

SERÍA INCORRECTO

```
x = p.getDiagonal();
r = p;
```

Ligadura dinámica




```
A oa;
oa.f(..);
```

¿qué versión de f se ejecuta?

Regla de la ligadura dinámica

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

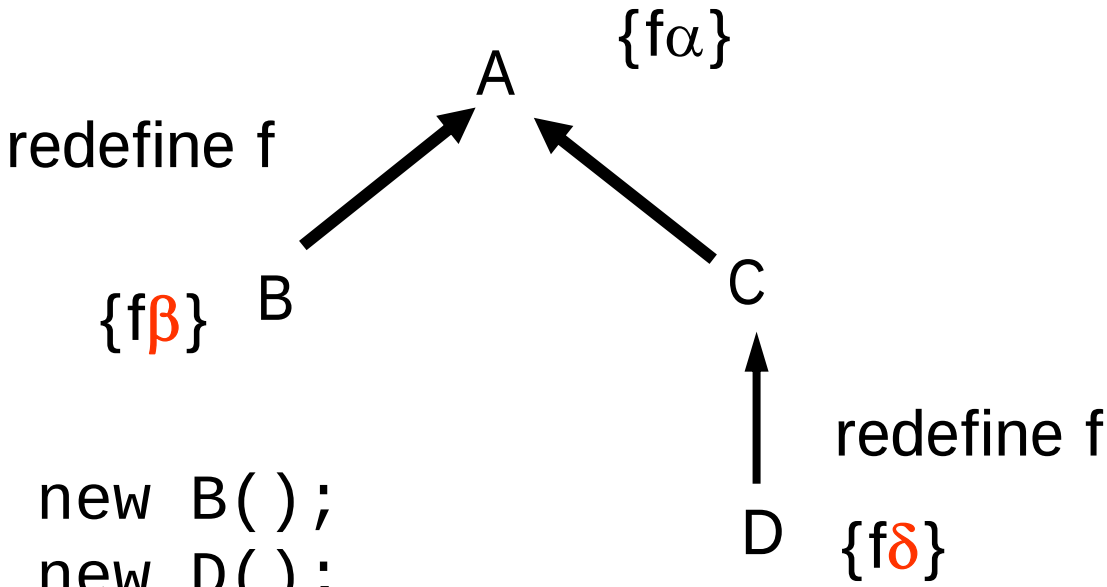
Ligadura dinámica

- La **versión** de una rutina en una clase es la introducida por la clase (redefinición u original) o la heredada.
- **Ejemplo 1:**

```
Poligono p = new Poligono();
Rectangulo r = new Rectangulo();

x = p.perimetro() → perimetro_POLIGONO
x = r.perimetro() → perimetro_RECTANGULO
p = r;
x = p.perimetro() → perimetro_RECTANGULO
```

Ejercicio: ¿Qué versión se ejecutará?



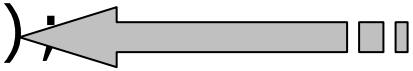

```
A oa;
B ob = new B();
D od = new D();
oa = ob;
oa.f();

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

Ligadura Dinámica

Ejemplo 2:

```
void visualizar (Figura [] figuras){
        for (Figura figura : figuras)
            figura.dibujar();
}
```



¿Qué sucede si aparece un nuevo tipo de figura?

- ¿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

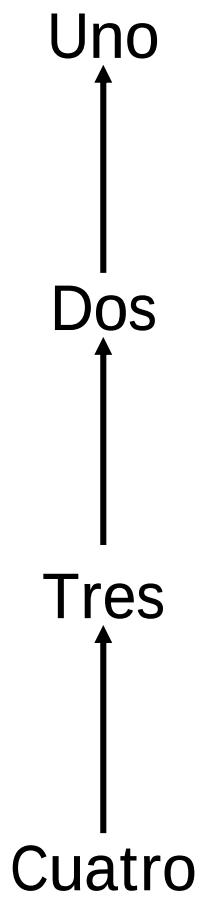
Ligadura dinámica y super

```
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();}
}

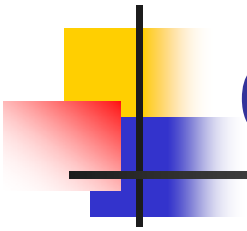
class Cuatro extends Tres{
    public int test(){return 4;}
}
```



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();

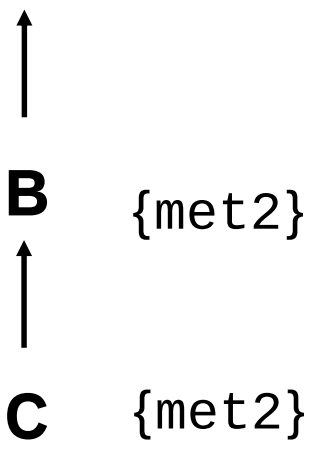
        System.out.println("ob1.test = "+ ob1.test());                  ---> 1
        System.out.println("ob1.result1 = " + ob1.result1());           ---> 1
        System.out.println("ob2.test = " + ob2.test());                 ---> 2
        System.out.println("ob2.result1 = " + ob2.result1());           ---> 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**.

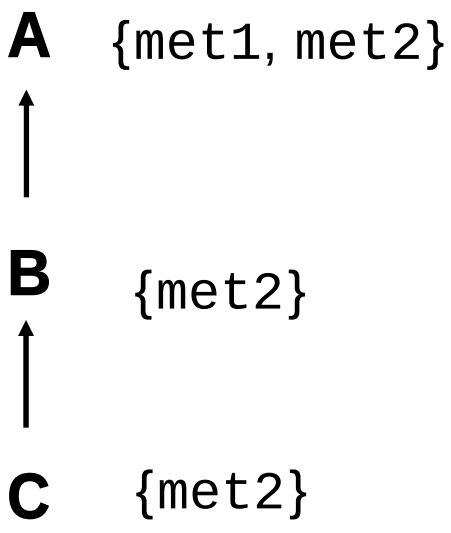
Patrones de código genérico




```
1) void met1(){
      ...
      this.met2();
      -- this es una entidad polimorfa
      ...
   }

2) void met3 (A p){
      ...
      p.met2();
      ...
   }
```

Patrones de código genérico




```
3) void met1(A p){
       ...
    this.met2();
    p.met2();
      ...
   }

4) void met3 (A[] tabla){
      for (A a: tabla)
            a.met2();
   }
```

Ejemplo: código genérico

```
public double precioVallado(Poligono parcela){

      return parcela.perimetro()*precioMetro;

}
```

- Calcular el precio del vallado de una parcela con forma de polígono
- El parámetro `parcela` puede referenciar en tiempo de ejecución a un polígono o a un rectángulo
- `parcela.perimetro()` ejecutará versiones diferentes en función del tipo dinámico del parámetro.

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;
```

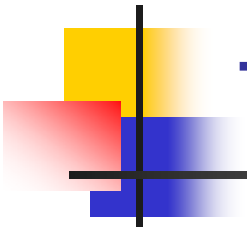
OK!!

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.

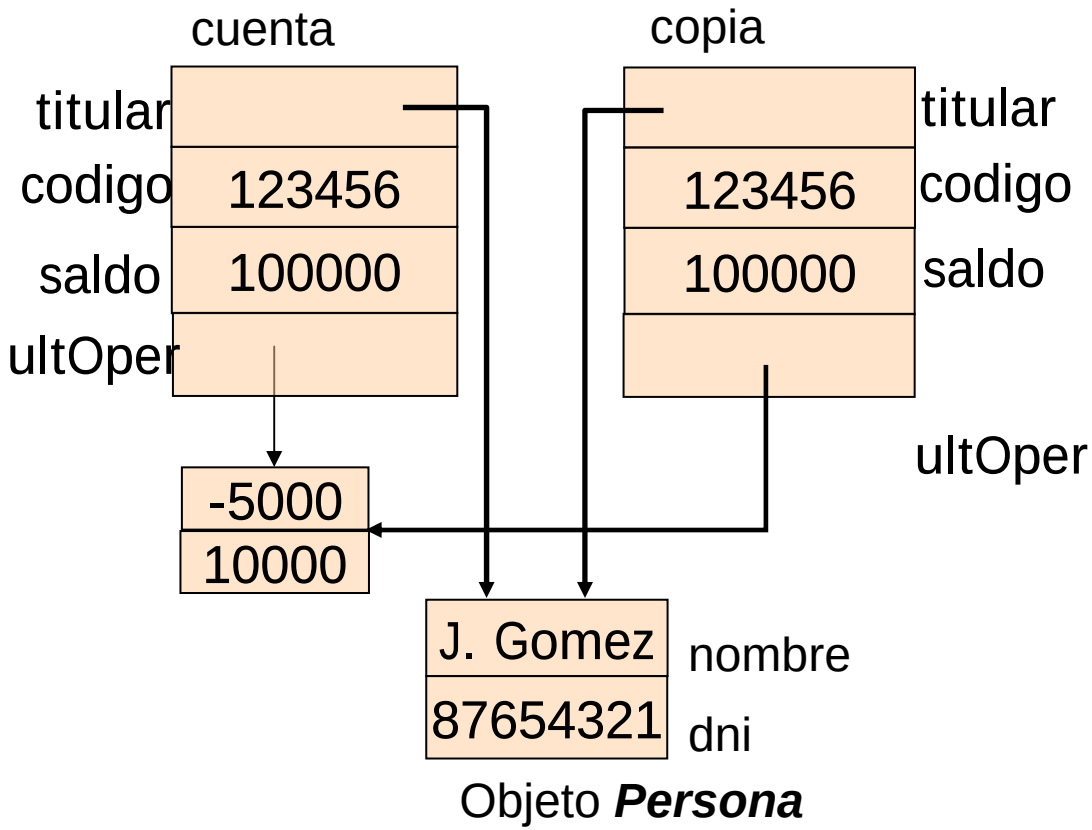


Tipos de copia

- **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.

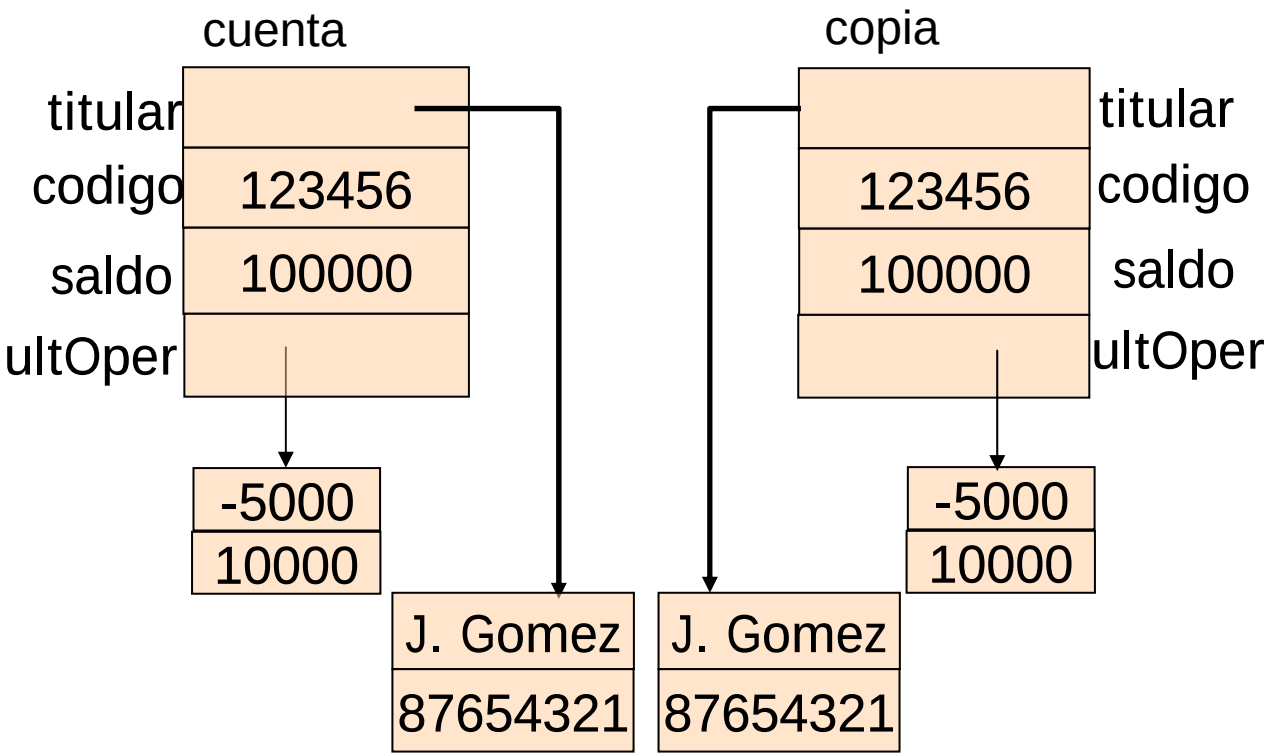
Copia superficial de Cuenta

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



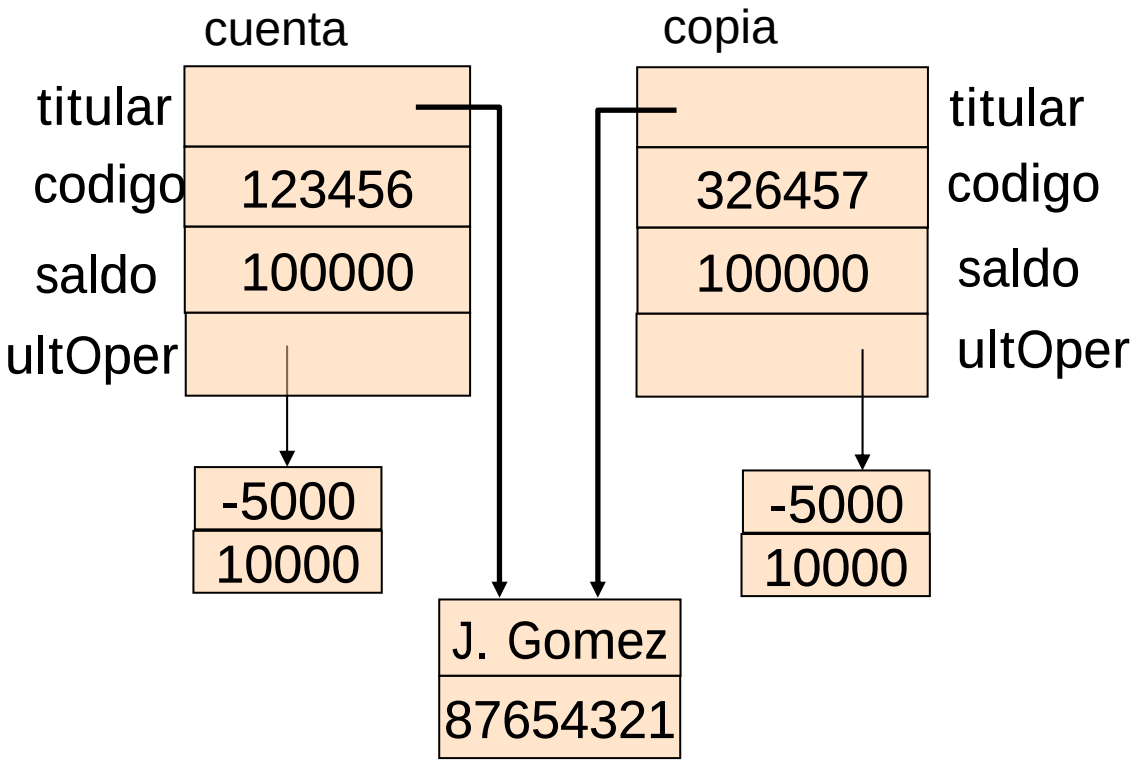
Copia profunda de Cuenta

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



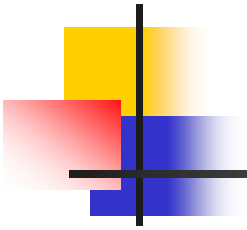
Copia correcta de Cuenta

- Una copia adaptada cumple los requisitos de la aplicación



Igualdad de objetos

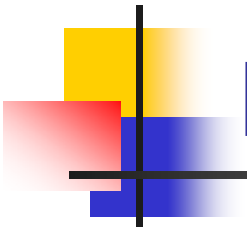
- El operador de igualdad (==) compara referencias (identidad)
- El método **`equals`** permite implementar la igualdad de objetos
- La implementación en la clase `Object`:
  - ```
    public boolean equals (Object objeto){
      return this == objeto;
    }
    ```
  - Por tanto: `a.equals(b);  //false 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.

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

```
Pila p; //quiero que sea de Movimientos de Cuenta
Movimiento m; Animal a;
… //creación de los objetos
p.push(m);
p.push(a);                    //NO daría error el compilador
m=p.pop();                    //error asignamos un Object
m=(Movimiento)p.pop()         //OK en compilación
                              //error en ejecución si lo que se
                        // extrae no es compatible con Movimiento
```

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

Estructuras de datos polimórficas

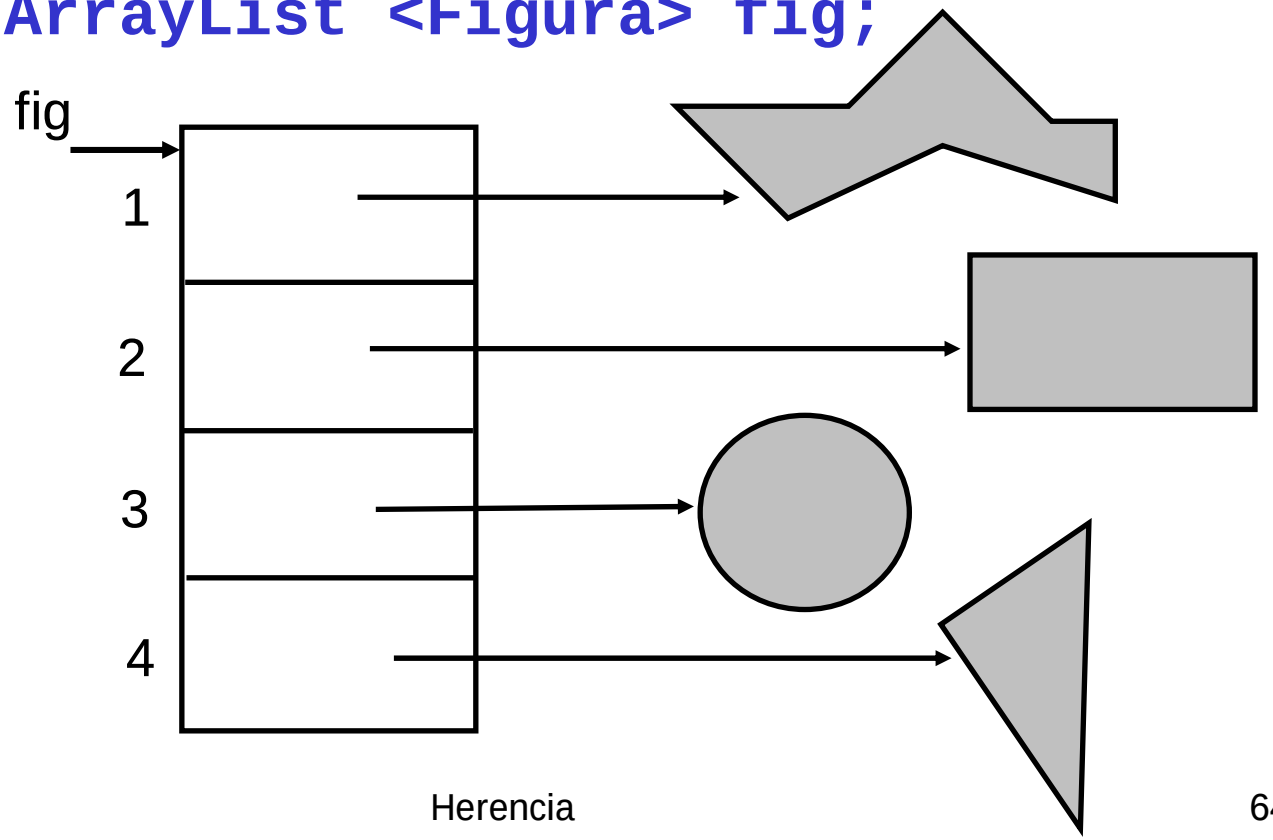
- Son aquellas estructuras de datos que pueden contener instancias de una jerarquía de clases.
- Ejemplo: **`Figura[] fig;`**

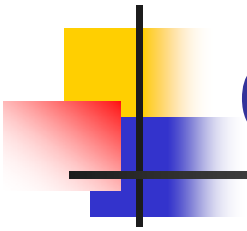
`ArrayList <Figura> fig;`

```
Poligono p;
Rectangulo r;
Circulo c;
Triangulo t;

//se crean …

  fig.add(0,p);
  fig.add(1,r);
  fig.add(2,c);
  fig.add(3,t);
  ...
```





Genericidad y polimorfismo

- Cuando instanciamos una clase genérica las entidades genéricas de la clase podrán hacer referencia a objetos compatibles con el tipo de la instanciación.

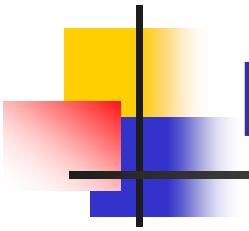
- **`ArrayList <Figura> figuras;`**
  - Significa que en la colección de `figuras` podremos almacenar cualquier tipo compatible con la clase `Figura: Rectangulo, Triangulo`, etc.
  - La colección NO va a almacenar SÓLO objetos de tipo `Figura`.

Tipo dinámico de los objetos

- Las estructuras condicionales en las que se pregunta por el tipo de los objetos van en contra de los principios de Elección Única y Abierto-Cerrado.

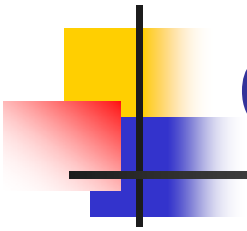
```
if “f es de tipo RECTANGULO” then
      …
elseif “f es de tipo CIRCULO” then
      ...
```

- Sin embargo, como consecuencia del manejo de estructuras de datos polimorfas puede ser inevitable tener que **preguntar por el tipo dinámico de los objetos → `instanceof`**.



Ejemplo

```
public float maxDiagonal (LinkedList<Figura> listaFig) {
   Figura f;
   double actual,result=-1;
   for (Figura figura : listaFig){
        if (figura instanceof Rectangulo){
             actual = (Rectangulo)figura.getDiagonal();
             if (actual>result) result = actual;
        }
    }
    return result;
}
```



Operador `instanceof`

- Comprueba si el tipo de una variable es **compatible** con un tipo dado
  - Es de ese tipo o alguna de sus subclases
- Si no se hace la comprobación, en el caso de que fallara el casting (en tiempo de ejecución) se abortaría el programa
- No es lo mismo hace la comprobación con `instanceof` que con el método `getClass` heredado de la clase `Object` (ver el seminario 2 de prácticas para más detalle)

¿Qué ocurre si quiero sumar vectores?

```
public class Vector <T> {
   private int count;
   public T item (int i){
       …
   }
   public void put(T v, int index){
    …
   }
   public Vector<T> suma (Vector<T> otro){
       Vector<T> result = new Vector<T>();
       for (int i=1; i <= count; i++)
             result.put(this.item(i) + other.item(i),i);
       return result;
   }
}
```

¿Se pueden sumar dos objetos de tipo T?

¿Es posible ampliar el nº de operaciones?

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 Vector <T extends Number>
class Dictionary <G, H extends Comparable>
class ListaOrdenada <T extends Comparable &
                                Serializable>
```

Genericidad restringida

```
public FiguraCompueta<T extends Poligono>{
   private List<T> contenido;

   public void dibujar(){
       for (T poligono : contenido)
             poligono.dibujar();
   }
}
```

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

Genericidad Restringida

- La genericidad no restringida equivale a:

```
<T extends Object>
```

- ¿Sería legal la declaración:

```
Vector<Vector<Number>> vectorDeVectores;
```
?

- ¿Son equivalentes las declaraciones:
  - `Vector<Number>`
  - `Vector<T extends Number>`

  ?

¿Son equivalentes?

```
class FiguraCompuesta<T extends Poligono>{
  private T [] contenido;

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

```
public class FiguraCompuesta{
   private Poligono[] contenido;

   public void insertar(Poligono p){
   ...
   }
   ...
}
```

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<Poligono> lp;
List<Rectangulo> lr = new …;
lp = lr; //ERROR en tiempo de compilación
```

Genericidad y sistema de tipos

- El compilador trata de asegurar que sobre un lista de polígonos no se incluyan rectángulos
- Sin embargo, existe un **agujero en el sistema de tipos** de Java:

```
List lista;
lista = new LinkedList<Poligono>();   // compila
lista = new LinkedList<Rectangulo>(); // compila
lista.add("POO");                     // compila!!
```

- Cuando se declara una variable de un tipo genérico sin parametrizar se asigna el **tipo puro** (*raw*) que corresponde a `Object`.

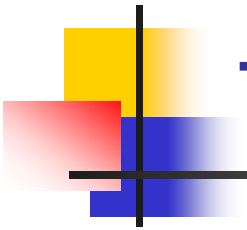
Genericidad y sistema de tipos

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

```
List<Rectangulo> rectangulos = new ...;
List<Poligono> poligonos;

//poligonos = rectangulos error en tiempo de compilación
//Sin embargo …
List lista;

lista = rectangulos;
poligonos = (List<Poligono>)lista;  //COMPILA!!
```



Tipos comodín

- Dado el código de la clase `Paint`:

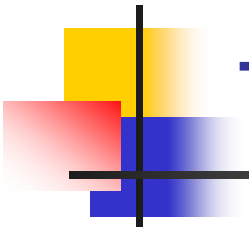
```
public static void imprimir (List<Poligono> poligonos){
      for (Poligono p : poligonos)
            System.out.println(p);
}
```

- El siguiente código daría error:

```
List<Rectangulo> lr = new LinkedList<Rectangulo>(); …
Paint.imprimir(lr); //Error no es una List<Poligono>
```

- Para que no de error hay que usar el comodín en el argumento:

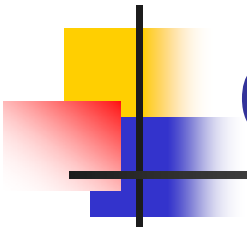
```
void imprimir (List<? extends Poligono> poligonos)
```



Tipos comodín

```
void imprimir (List<? extends Poligono> poligonos)
```

- Significa: “permite cualquier lista genérica instanciada a la clase `Poligono` o a cualquiera de sus subclases”
- Si pasamos como parámetro un objeto `List<Rectangulo>`, 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<Poligono>) //no compila`
  - `(lp instanceof List)  //si compila`

Clases abstractas

Sea la declaración

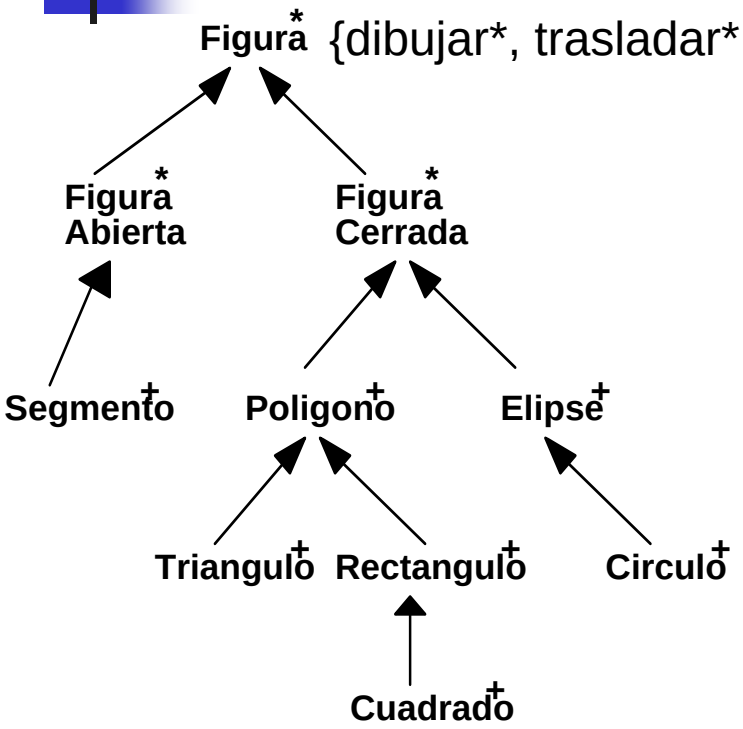
```
Figura f; Poligono p;
```

y el código

```
p = new Poligono(…);
f = p;
f.dibujar();    ¿Sería legal?
```

- ¿Cómo se implementa **`dibujar`** en la clase **`Figura`**?
- La rutina **`dibujar`** no puede ser implementada en **`Figura`** pero **`f.dibujar`** es **¡dinámicamente correcto!**
- ¿Tendría sentido incluir **`dibujar`** en **`Figura`** como una rutina que no hace nada?

Solución: Métodos abstractos



En `Figura`:

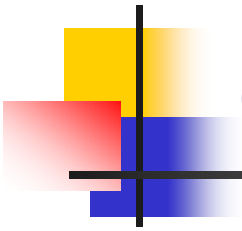
```
abstract void dibujar();
```

Clases abstractas

```
public abstract class Figura {...}
```

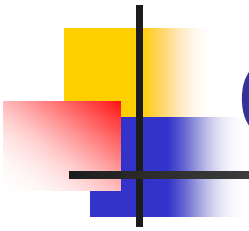
- Una subclase de una clase abstracta puede seguir siendo abstracta

```
public abstract class FiguraAbierta extends Figura { ... }
public abstract class FiguraCerrada extends Figura { ... }
```



Clases abstractas

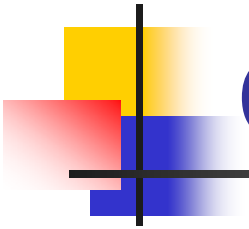
- Toda clase que contenga algún método abstracto (heredado o no) es abstracta.
- Una clase puede ser abstracta y no contener ningún método abstracto.
- Especifica una **funcionalidad que es común** a un conjunto de subclases aunque no es completa.
- Puede ser total o parcialmente abstracta.
- **No es 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.
- Los métodos efectivos pueden hacer uso de los abstractos.
- Importante mecanismo para incluir código genérico.
- Incluyen comportamiento abstracto común a todos los descendientes.

"***programs with holes***"

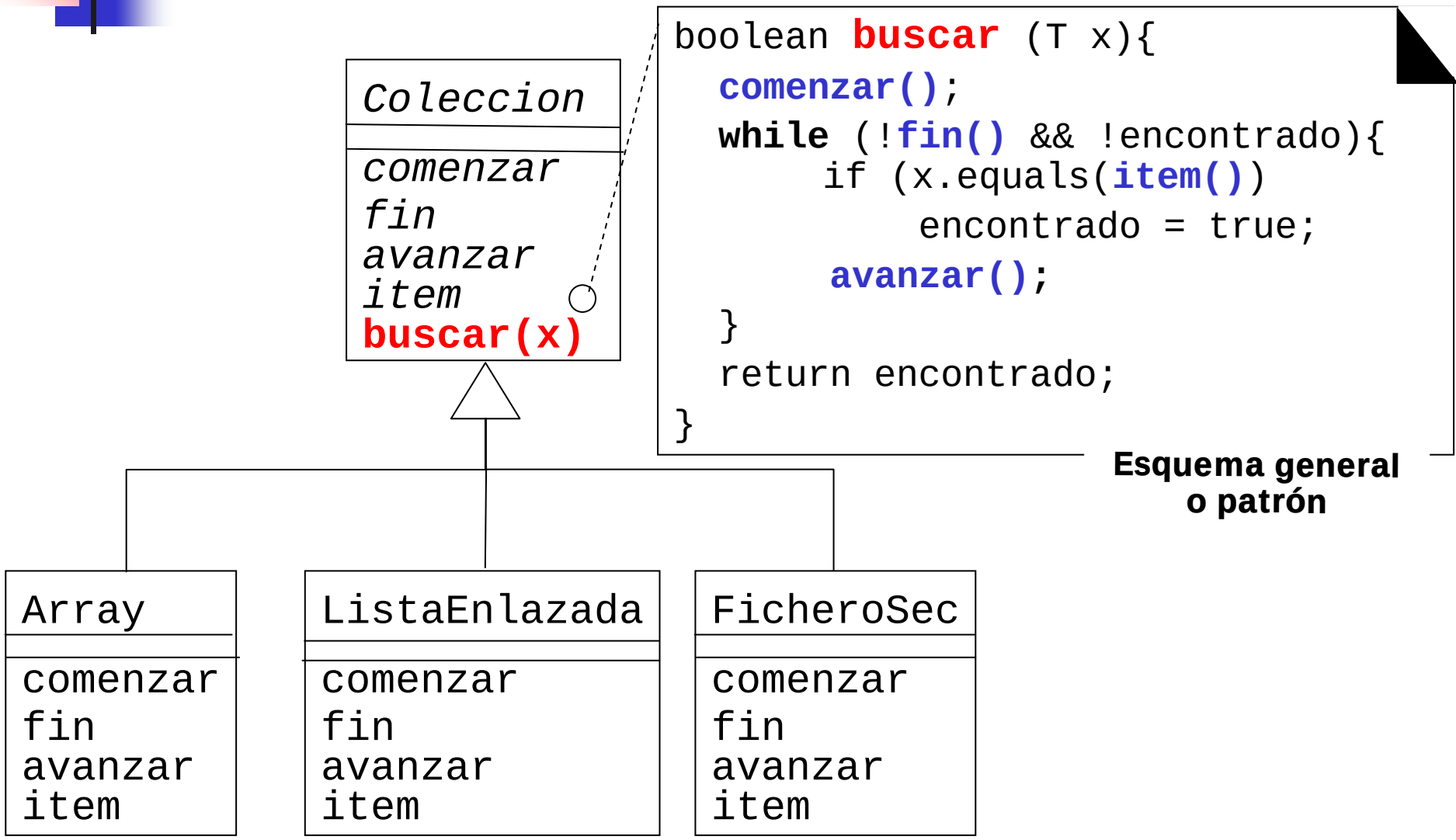


Clases parcialmente abstractas

- "Permiten **capturar lo conocido** sobre el comportamiento y estructuras de datos que caracterizan a cierta área de aplicación, **dejando una puerta abierta a la variación**"

- Se satisface el requisito de reutilización de **factorizar comportamiento común**

Clases parcialmente abstractas

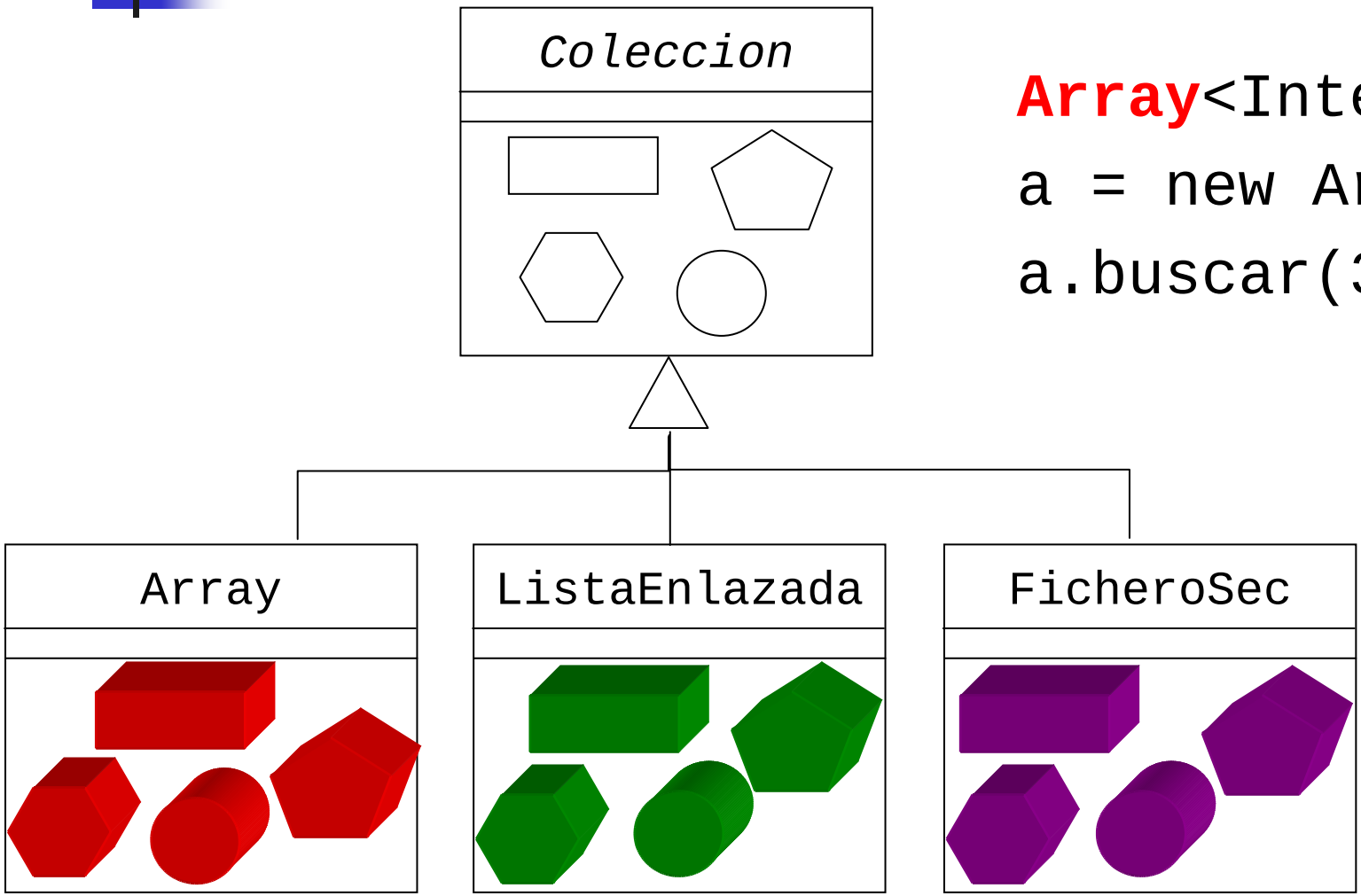


Método Plantilla

```
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();
}
```

“No nos llame, nosotros le llamaremos”




```
Array<Integer> a;
a = new Array();
a.buscar(31);
```

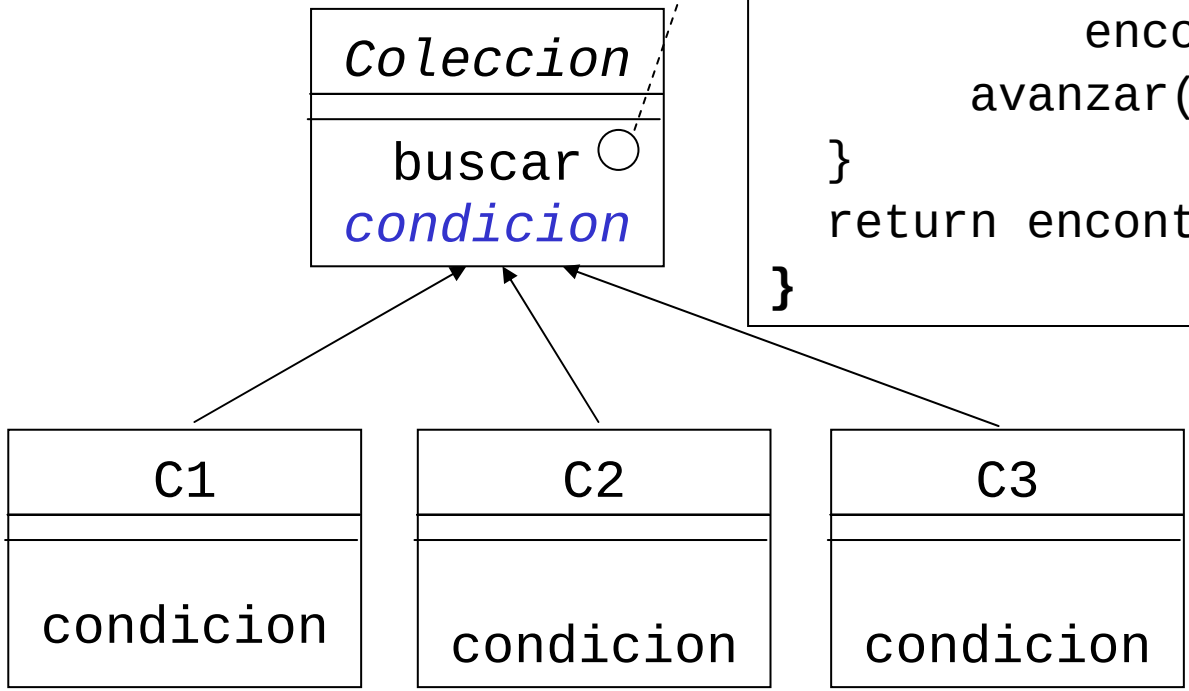
Implementación de Acciones

- ¿Cómo podemos buscar si existe un elemento que cumple una condición?

```
buscar(condicion)
```

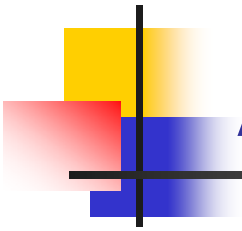
- Dos posibles soluciones:
  1) Herencia → Método plantilla
  2) Composición

Acciones con Método Plantilla




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

No!?

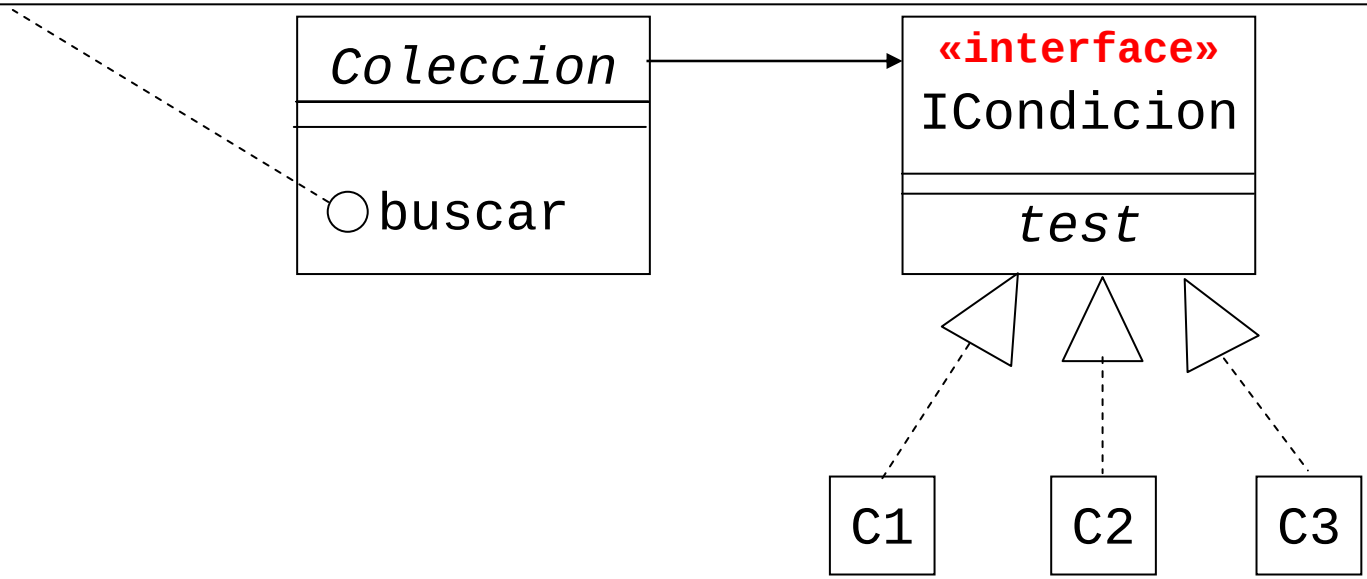


Acciones con Método Plantilla

- **Problemas de la solución**:
  - La condición de búsqueda debe ser implementada por los subtipos: `Array`, `ListaEnlazada`, etc.
  - Habría que crear un subtipo de Array (o de cualquier colección) por cada condición de búsqueda que queramos hacer
  - La condición depende de la aplicación y no del tipo de colección.
- **Alternativa**:
  - Establecer la **condición como parámetro**.
  - Problema: en Java no es posible definir punteros a funciones.

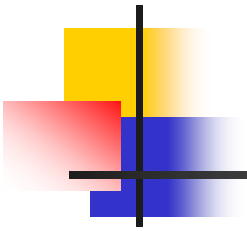
Acciones mediante composición

```
public boolean buscar (ICondicion c){
      boolean encontrado = false;
     for (int i=0; (i<size() && !encontrado); ++i){
           if (c.test(get(i))) encontrado = true;
      return encontrado;
}
```



Acciones mediante composición

- La condición podría ser una clase totalmente abstracta
  - Equivale a la definición de un TAD
- Java proporciona una construcción para la definición de tipos (sin implementación)→ **`interface`**
- Se dice que una clase **implementa** una interfaz
- 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



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.
- Si una clase no implementa todos los métodos de una interfaz debe declararse como abstracta

Interfaz ICondicion

- Una condición debe tomar un elemento y evaluarlo

```
public interface ICondicion<T>{
    boolean DEFAULT = false;  //cte

    boolean test(T objeto);
}
```

- Sólo están permitidos los modificadores: `public`, `abstract` y `final`

Implementación de la Interfaz

- Buscamos si existe una cuenta con saldo superior a un determinado umbral

```
public class CondicionSaldoSuperior implements
                                        ICondicion<Cuenta>{
   private double umbral;

   public CondicionSaldoSuperior(double cantidad){
        umbral = cantidad;
   }

   @Override
   public boolean test (Cuenta cuenta){
        return cuenta.getSaldo() > umbral;
   }
}
```

Pasar una acción como parámetro

- Para pasar la condición al método de búsqueda habría que crear un objeto de la clase que implementa la colección:

```
Coleccion<Cuenta> cuentas;
…
cuentas.buscar(new CondicionSaldoSuperior(60000));
```

Acciones anónimas

- ¿Tiene sentido crear una clase por cada condición?

- Solución basada en clases anónimas

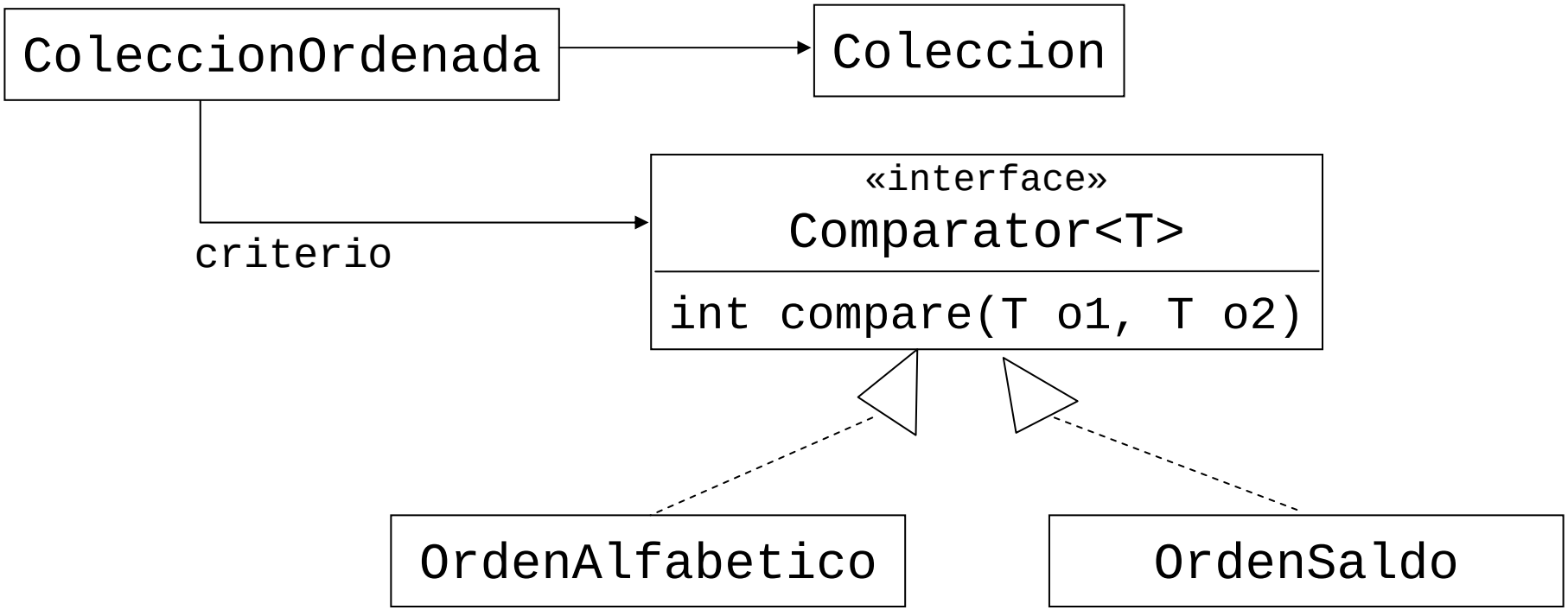
```
Coleccion<Cuenta> cuentas;
…
cuentas.buscar(new ICondicion<Cuenta>(){
  @Override
  public boolean test (Cuenta cuenta){
     return cuenta.getSaldo() > 60000;
  }
});
```

Acciones anónimas

- Solución basada en clases anónimas que necesita un parámetro

```
Coleccion<Cuenta> cuentas;
…
public boolean buscarUmbral (final double umbral){
  return cuentas.buscar(new ICondicion<Cuenta>(){
      @Override
      public boolean test (Cuenta cuenta){
            return cuenta.getSaldo() > umbral;
      }
  });
}
```

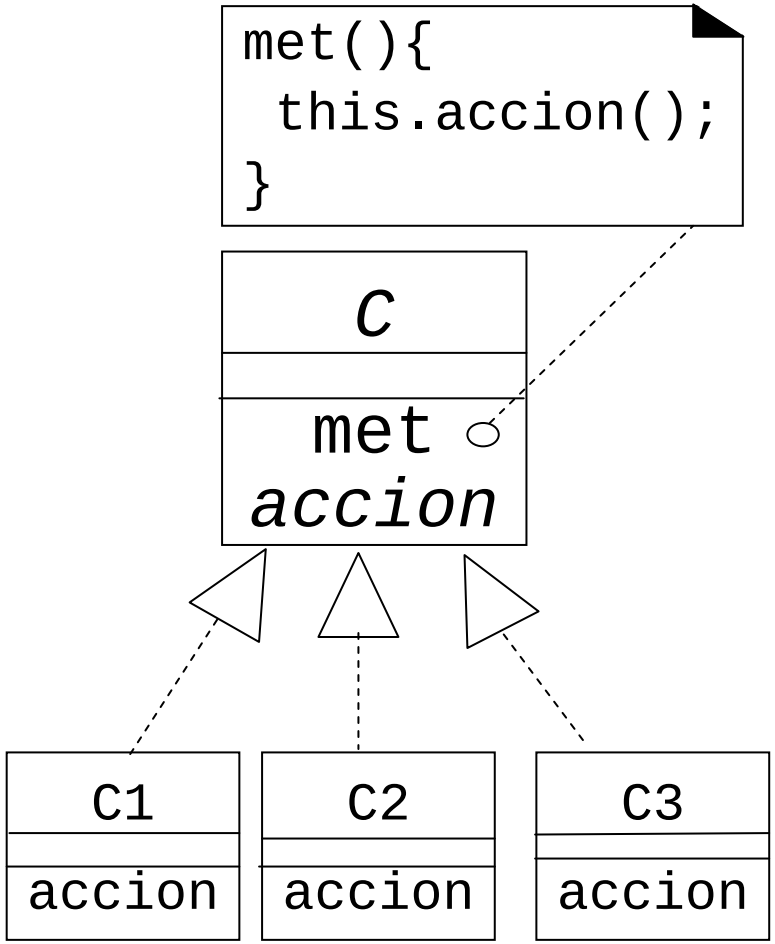
Ejercicio: Colección ordenada



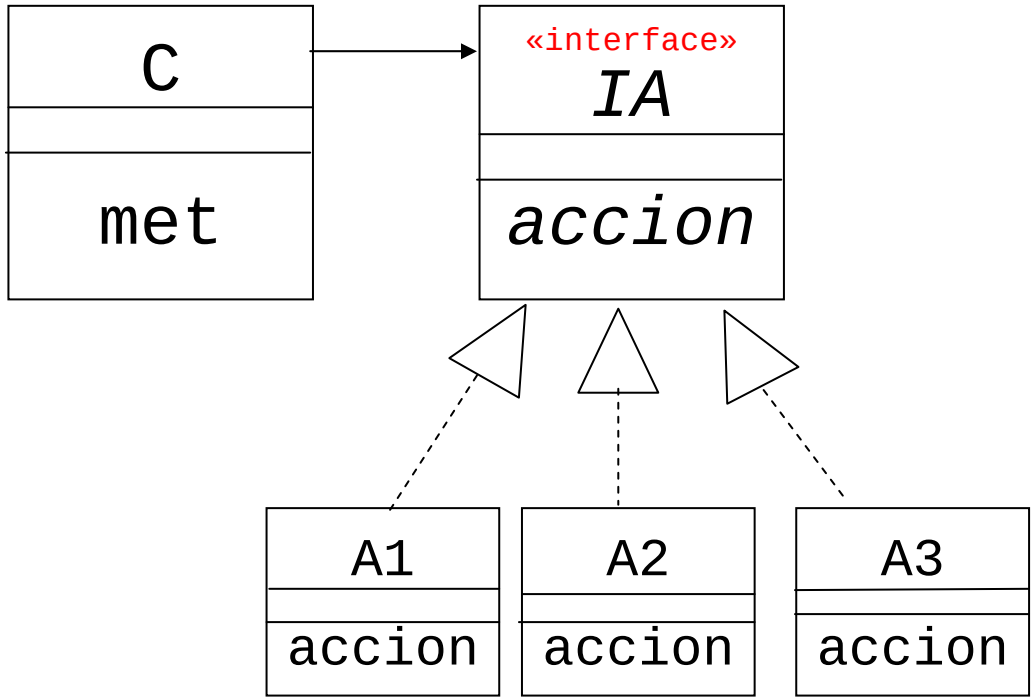

```
ColeccionOrdenada<Cuenta> cuentas;
cuentas = new ColeccionOrdenada<Cuenta> (new OrdenAlfabetico());
```

- **Ejercicio**: implementar **`add`** en **`ColeccionOrdenada`**.

Resumen: Implementación de acciones

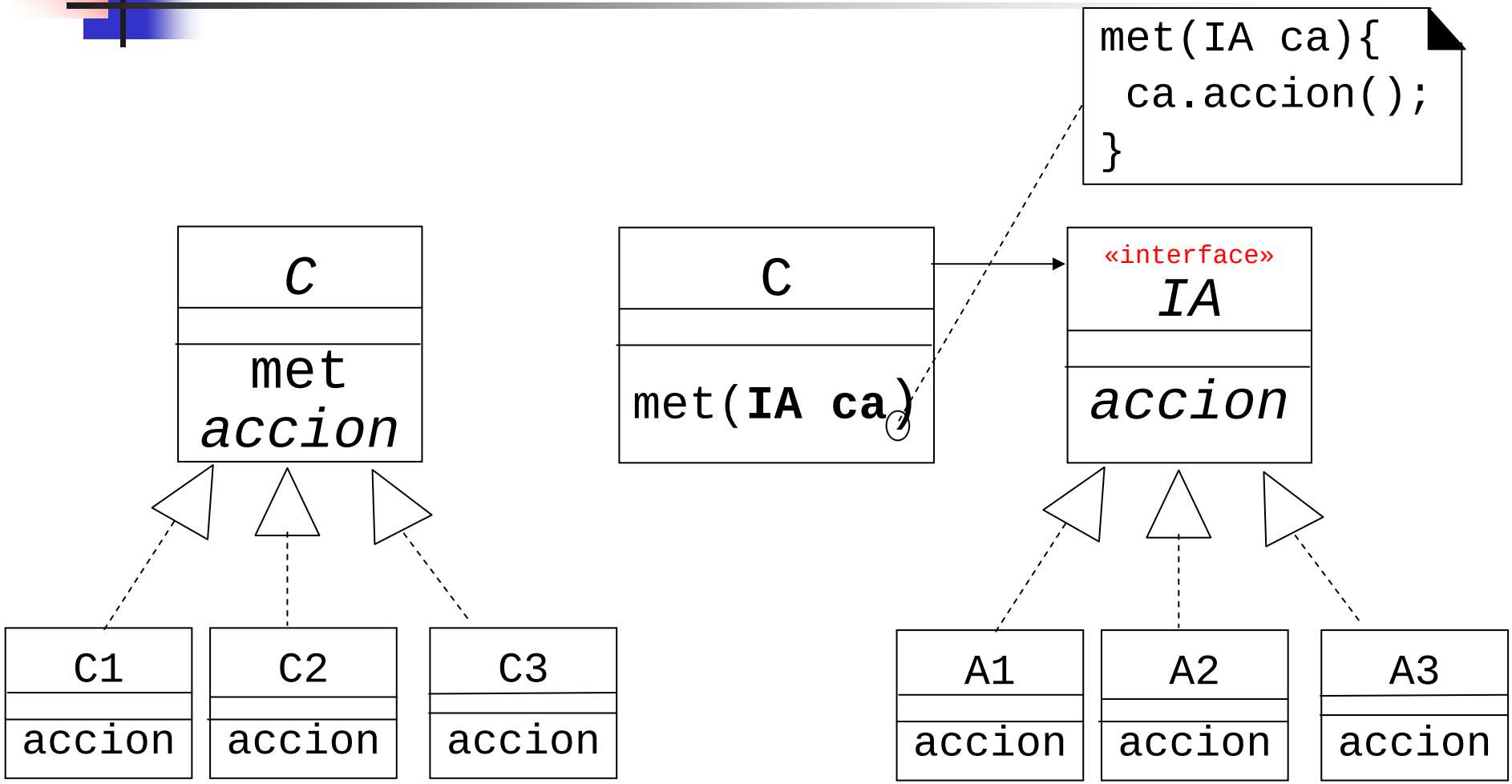


a) Método plantilla



b) Composición

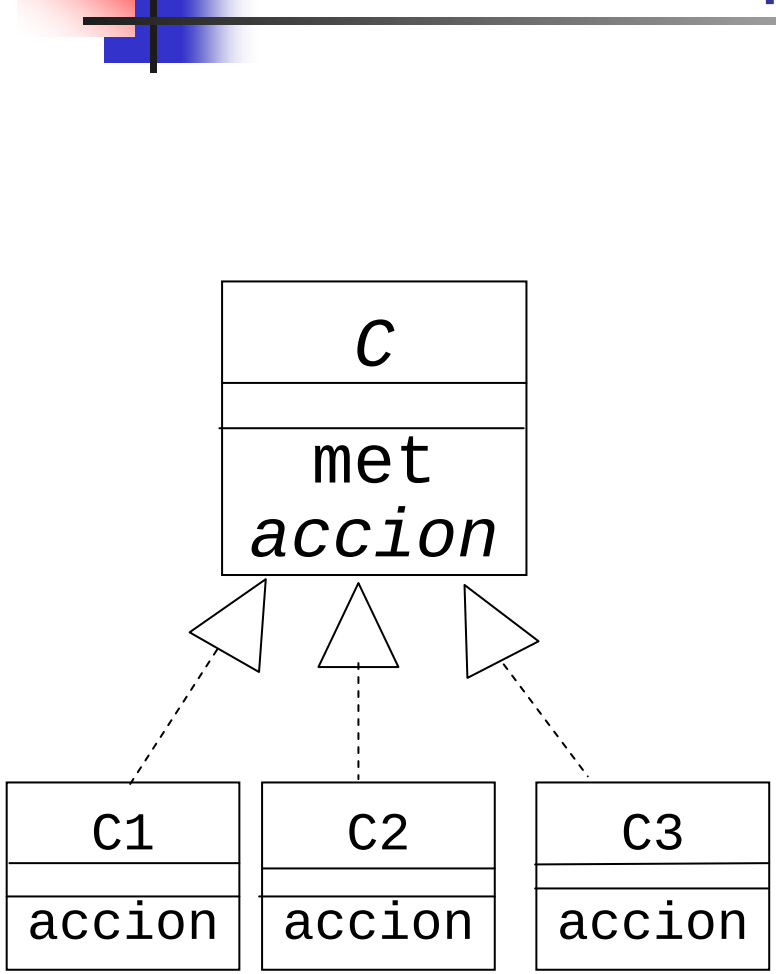
Resumen: Implementación de acciones



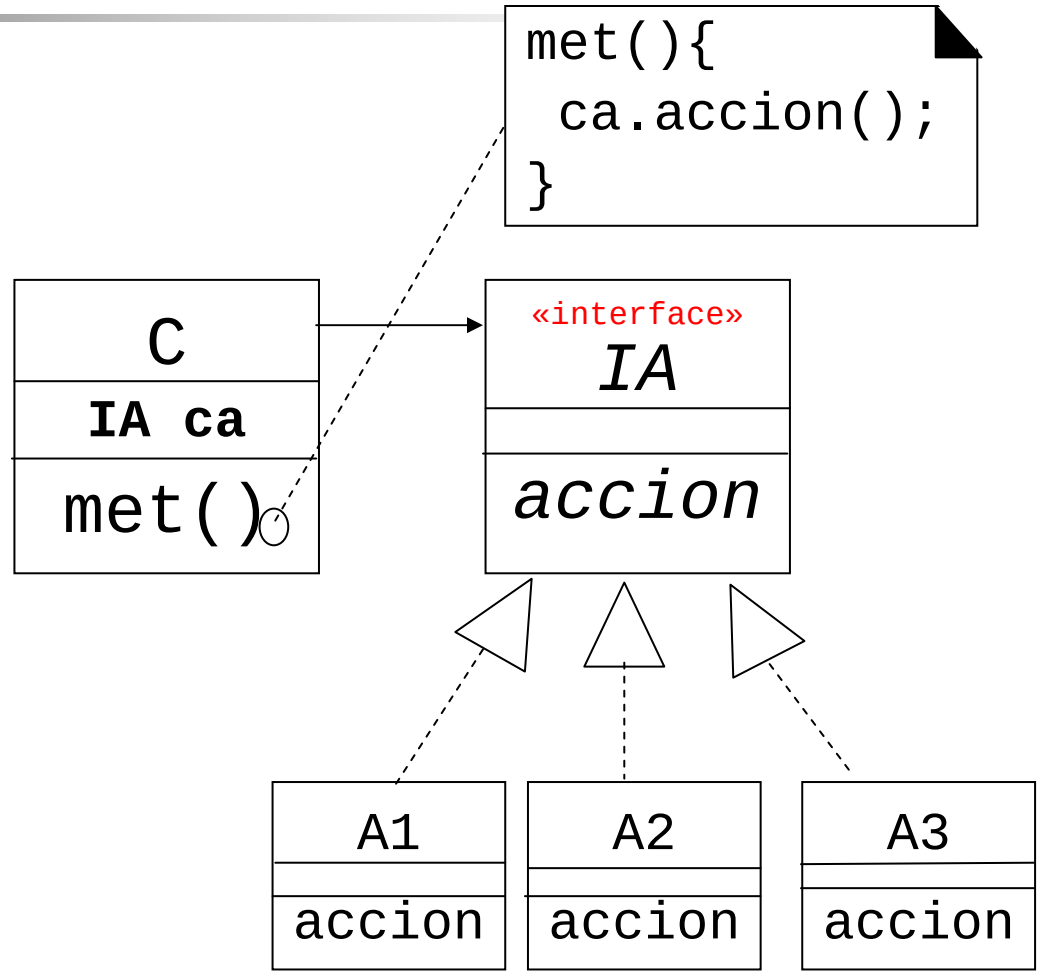
a) Método plantilla

b) Composición

Resumen: Implementación de acciones



a) Método plantilla



b) Composición

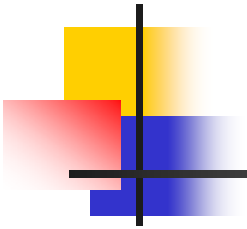
Iteradores

- “Iterar” significa ejecutar un cierto procedimiento (**accion**) sobre todos los elementos de una estructura de datos (**coleccion**) o sobre aquellos que cumplan una condición (**test**).

```
List<Cuenta> listaCuentas;

for (Cuenta cta : listaCuentas)
   if (cta.estaNumerosRojos())
      cta.informarTitular();
```

- El cliente que realiza la interacción no debe conocer las particularidades de la estructura de datos (**abstracción por iteración**).



Iteradores internos

- Interesa capturar "**patrones o esquemas de recorrido** de estructuras de datos": reutilizar en vez de escribir de nuevo.
- Un sistema que haga uso de un mecanismo general para iterar debe ser capaz de aplicarlo para cualquier `accion` y `test` de su elección.
- El método de iteración debe estar parametrizado por la acción y la condición.
- Como ya sabemos, en Java no es posible pasar una rutina como argumento.

Implementación de los Iteradores internos

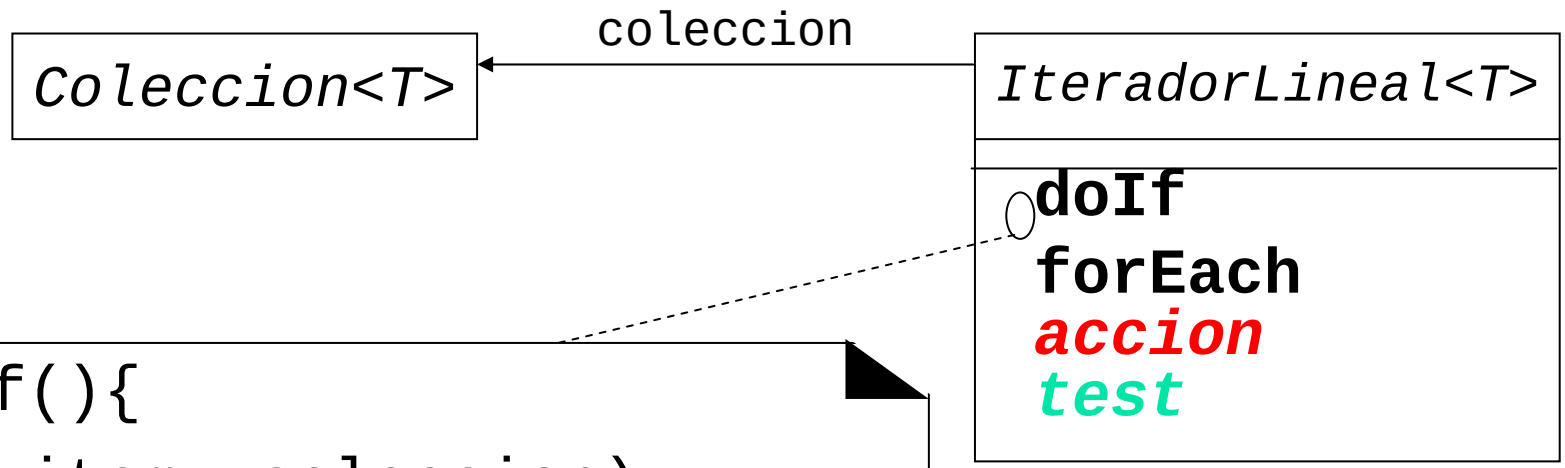
A. Definir los métodos de iteración en la clase `Coleccion` → **NO**

- Una iteración es una propiedad del cliente, no de la colección
- Deberíamos crear descendientes de las clases que representan colecciones para crear diferentes esquemas de iteración.

B. Implementar la clase `Iterador` → **SI**

- Representa objetos con capacidad para iterar sobre colecciones.

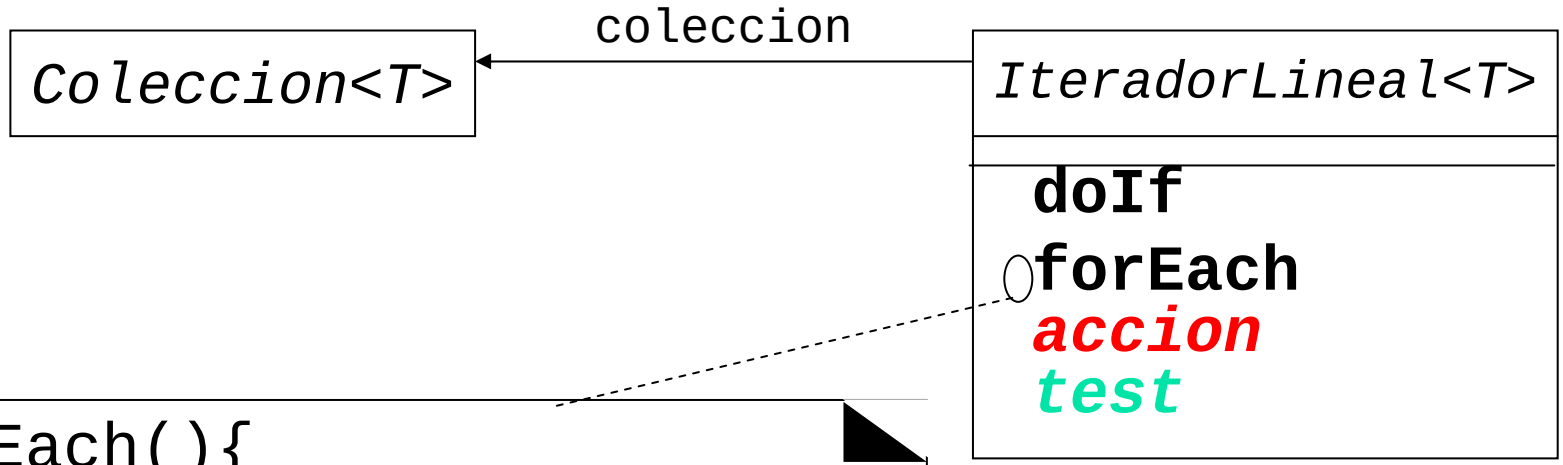
Iteradores internos




```
void doIf(){
  for (T item: coleccion)
     if (test(item))
        accion(item);
}
```

Método plantilla!!

Iteradores internos




```
void forEach(){
  for (T item: coleccion)
        accion(item);
}
```

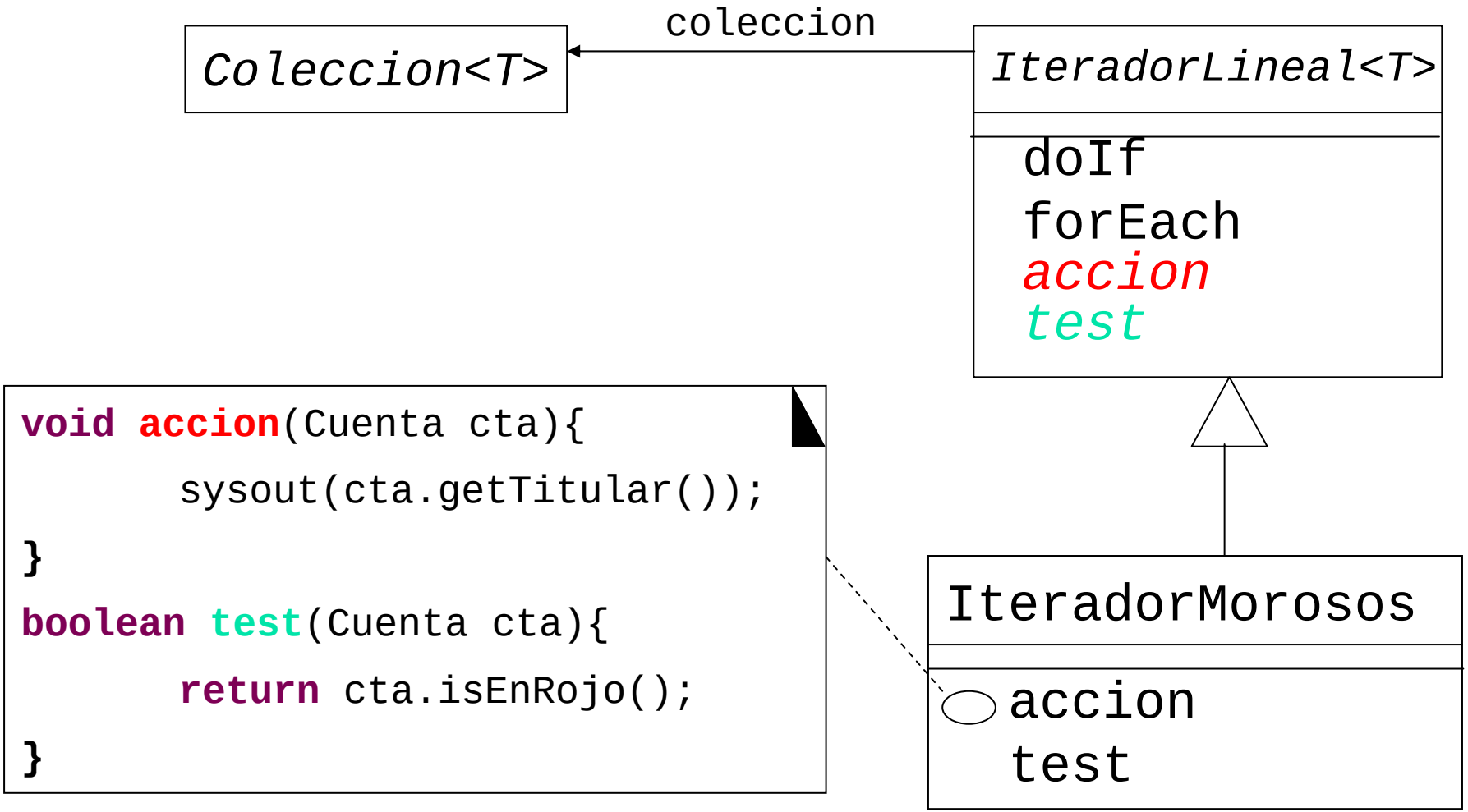
Método plantilla!!

Iteradores internos

```
abstract class IteradorLineal <T> {
   private List<T> coleccion;
   public IteradorLineal(List<T> c){
        coleccion = c;
   }
   public abstract void accion (T item);
   public abstract boolean test (T item);

   /* Ejecuta una acción sobre todos los elementos de
      la colección */
   public void forEach(){
        for (T item : coleccion)
              action (item);
   }
   // Ejecuta la acción sobre los items que cumplen test
   public void doIf(){
        for (T item : coleccion)
              if test(item) accion(item);
   }
}
```

Iteradores internos

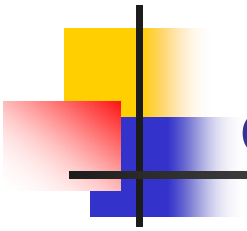


Iteradores internos

- `class IteradorMorosos extends IteradorLineal<Cuenta> { }`

- Suponiendo que `LinkedList<Cuenta> cuentas;` contiene todas las cuentas del banco, si queremos imprimir por pantalla los titulares de las cuentas que están en números rojos haríamos:

```
public void imprimeMorosos(){
 IteradorMorosos iterador = new IteradorMorosos (cuentas);
 iterador.doIf();
}
```



¿Quién controla la iteración?

- Iterador interno:
  - El iterador es quien controla la iteración.
  - Es el iterador el que aplica una operación a cada elemento de la colección.
  - Ejemplo: clase `IteratorLineal`

- Iterador externo:
  - El cliente es el que controla la iteración.
  - El cliente es el que avanza en el recorrido y pide al iterador el siguiente elemento.
  - Ejemplo: `Iterator` de Java.

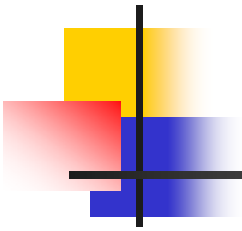
Iteradores externos en Java

- 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();
}
```

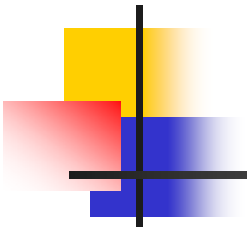
- A los objetos iterables se les exige que creen objetos iterador (*iterator*) para realizar la iteración.



Iteradores externos 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();
}
```



Iteradores

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

  → Evita tener que manejar explícitamente el objeto iterador.

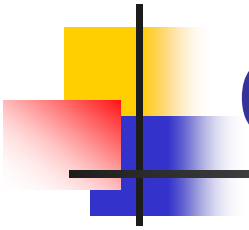
- La **implementación de un iterador** requiere crear una clase responsable de llevar el estado de la iteración → **Clases internas**

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 anidada tiene visibilidad sobre las declaraciones de la clase contenedora
- Los objetos anidados 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`

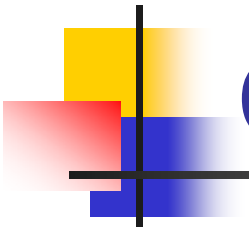
Uso del iterador

- Utilizamos ahora el ***iterador externo*** para imprimir los morosos:

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

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

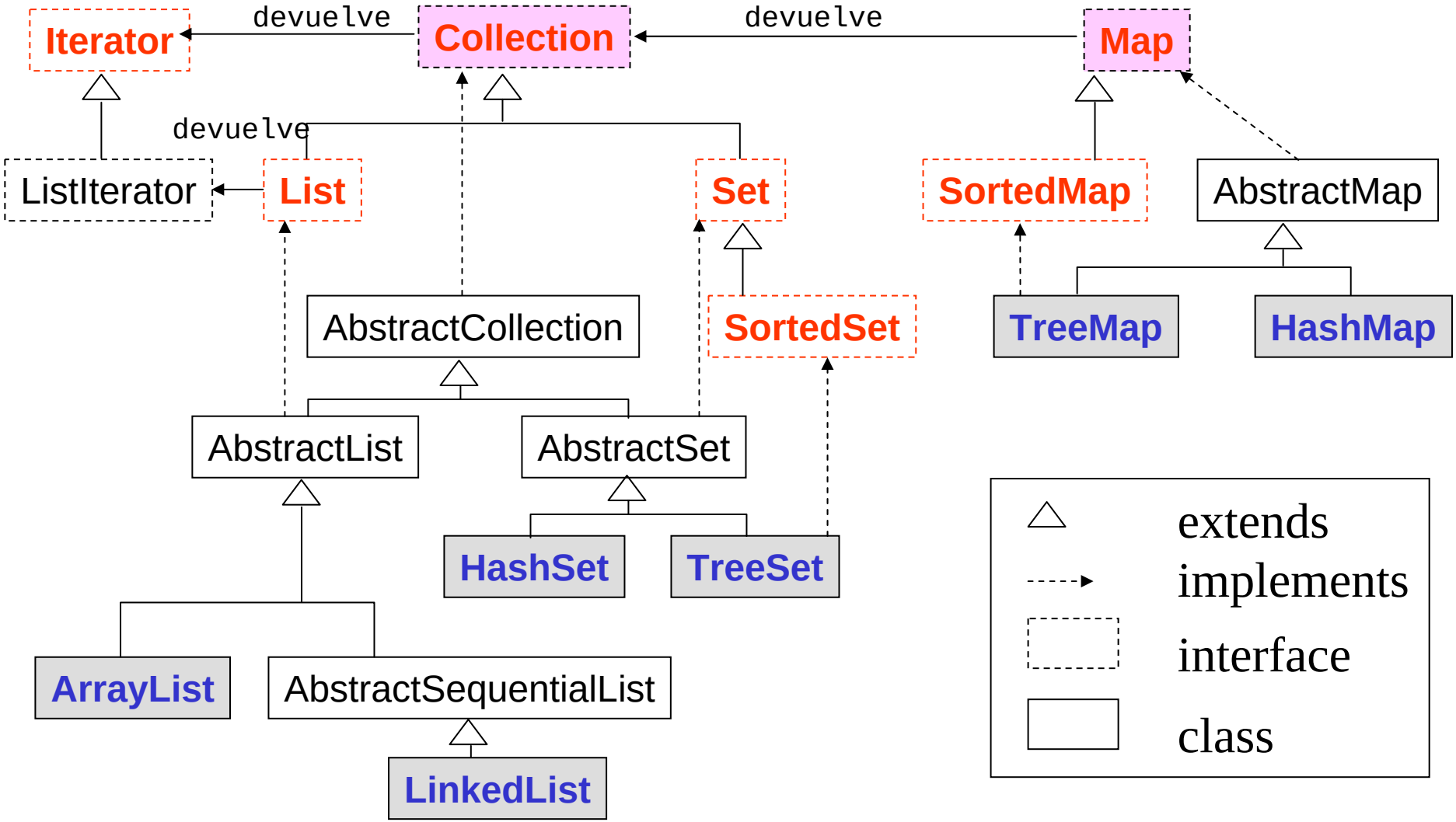
      }
}
```

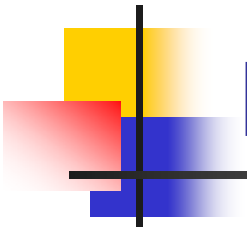


Colecciones en Java

- 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`)





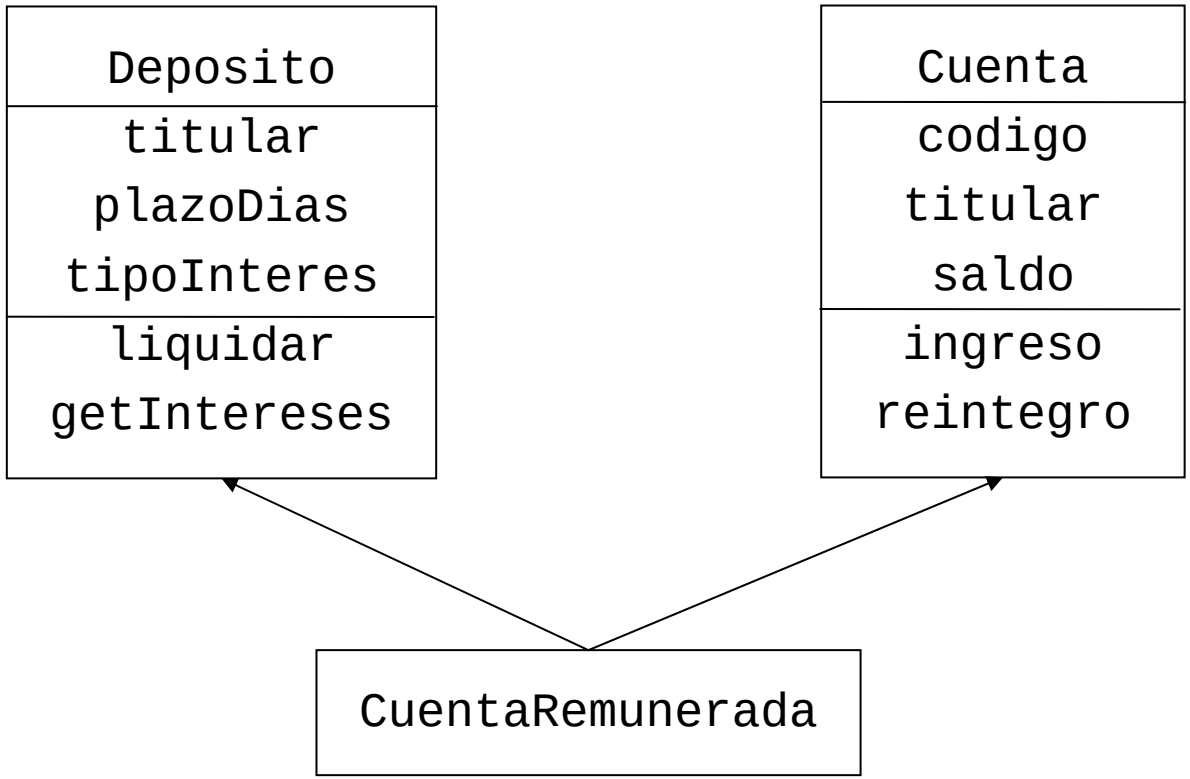
Herencia múltiple

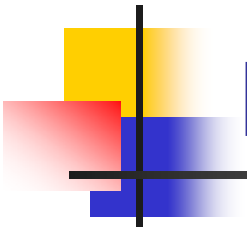
- ¿Qué proporciona la herencia?
  - Posibilidad de reutilizar el código de otra clase (perspectiva módulo).
  - Hacer que el tipo de la clase sea compatible con el de otra clase (perspectiva tipo).

- 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

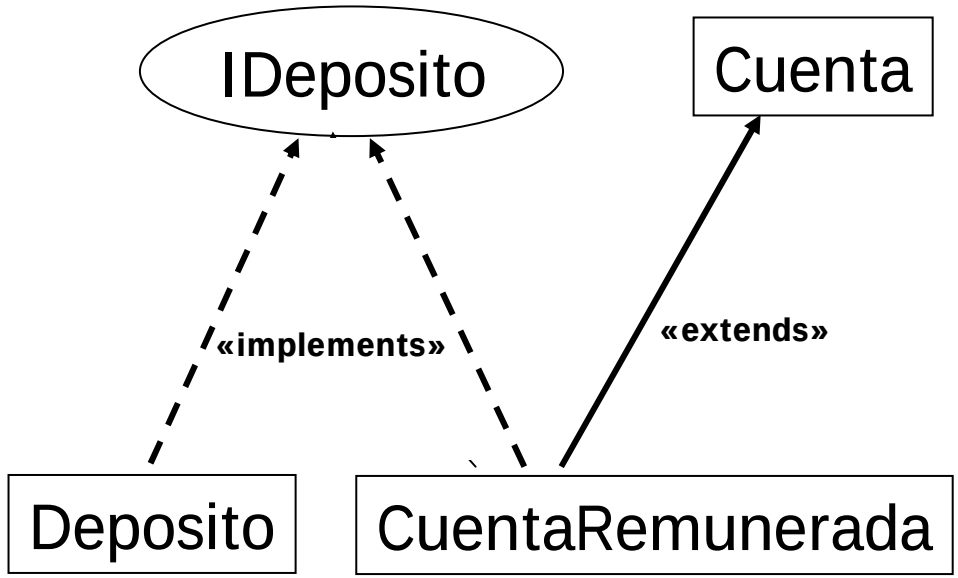
- En Java sólo podemos heredar de una clase:
  - Elegimos la clase `Cuenta` como clase padre.

- ¿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`.

Herencia múltiple en Java

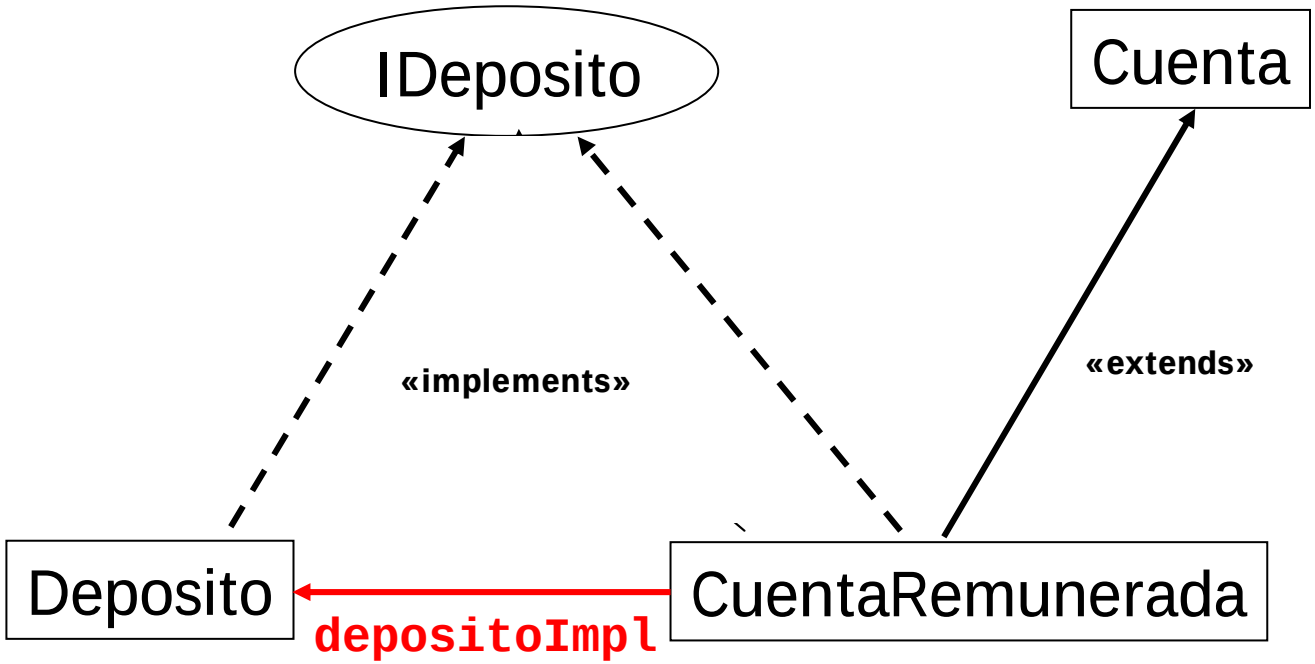
```
public interface IDeposito
{
    double liquidar();
    double getIntereses();
    double getCapital();
    int getPlazoDias();
    double getTipoInteres();
    Persona getTitular();
}
```



- `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) {

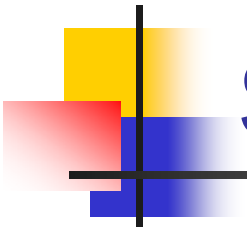
     super(titular, saldoInicial);
       // Liquidacion mensual de intereses
     depositoImpl =
       new Deposito(titular,saldoInicial,30,tipoInteres);

   }
```

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) {

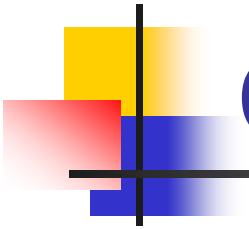
     super(titular, saldoInicial);
       // Liquidacion mensual de intereses
     depositoImpl =
       new DepositoImpl(titular,saldoInicial,30,tipoInteres);
   }
```

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 campos 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