Programación Orientada a Objetos

TEMA 5

Herencia Múltiple

Facultad de Informática

Universidad de Murcia

Índice

1.- Introducción

2.- **Utilidades** de la herencia múltiple

3.- **Problemas** con la herencia múltiple.

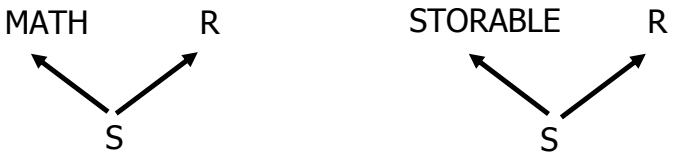
- Colisión de nombres
- Herencia repetida
  - Conflicto cuando hay compartición
  - Conflicto cuando hay replicación

4.- Herencia múltiple en Java: **Interfaces**

1.- Introducción

- Las clases pueden necesitar mas de una clase padre
- Más difícil de usar e implementar que la herencia simple.
- Algunos consideran que produce más inconvenientes que ventajas. Según B. Meyer:
  - *"No puedo pensar en una aplicación seria escrita en Eiffel que no use la herencia múltiple en un modo significativo"*
  - *"Para discutir cuestiones conceptuales sobre H.M. es necesario considerar el doble aspecto de la herencia: subtipos y extensión modular"*

Herencia múltiple y Extensión modular



- S es una especialización de R
- MATH incluye un paquete de rutinas matemáticas
- STORABLE ofrece servicios de persistencia de objetos

Herencia Múltiple y Subtipos

Ejemplo1: ***Implementación de los menús de una aplicación***

i) Los menús son **simples**

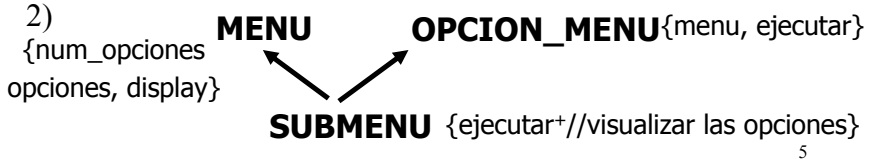
MENU ⟹ **ARRAY [OPCION_MENU]**

ii) Los menús contienen **submenús**

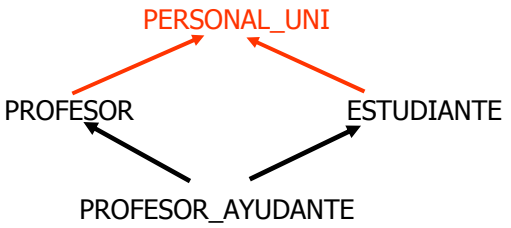
1) **MENU** ⟹ **TREE [OPCION_MENU]**

"provocaría muchos cambios"

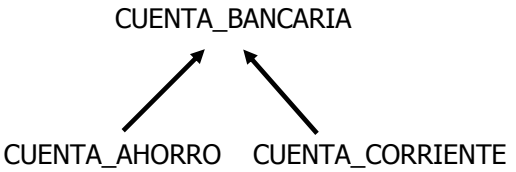
2)



Ejemplo2: *"Profesores ayudantes están matriculados de los cursos de doctorado"*

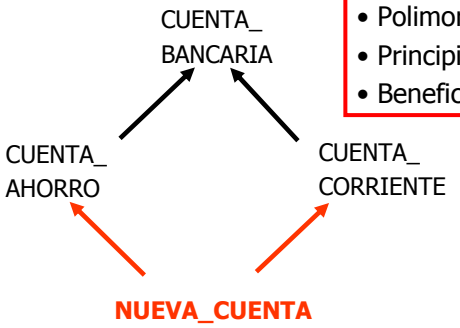


Ejemplo3: *"Añadir un nuevo tipo de cuenta que comparte a la vez propiedades de cuenta de ahorro y cuenta corriente"*



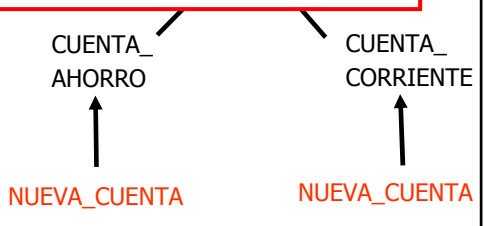
SOLUCIÓN:

Con herencia múltiple



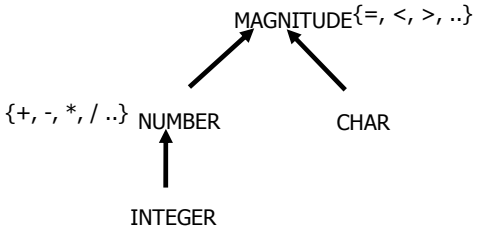
Sin herencia múltiple

Perdemos:
- Polimorfismo
- Principio de Abierto-Cerrado
- Beneficios de la reutilización de la herencia



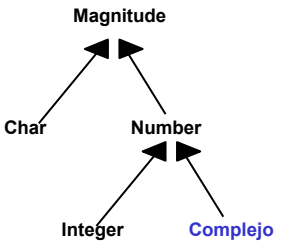
Debemos añadir a **NUEVA_CUENTA**
las propiedades de
CUENTA_CORRIENTE

Ejemplo4: *"Añadir a la jerarquía Smalltalk una clase para representar números complejos"*



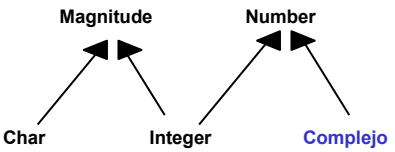
Soluciones:

OPCION A



{=, >, <, ERROR}

OPCION B



2.- Utilidades de la herencia múltiple

A) Combinar abstracciones de tipos (padres simétricos)

B)Matrimonio de conveniencia (Herencia de implementación)

C) Herencia de estructura

D) Herencia de facilidades

A) Combinación de interfaces no relacionadas

Ejemplo1:

```
class WINDOW inherit
    TREE [WINDOW]
    RECTANGLE
feature
    ....
end
```

"Una ventana es un objeto gráfico y un árbol de ventanas"

```
class TEXT_WINDOW inherit
     WINDOW
     STRING
feature
    ....
end
```

"Una ventana de texto es una ventana que manipula texto"

A) Combinación de interfaces no relacionadas

Ejemplo2:

```
class TREE [G] inherit
    LIST [G]
    LINKABLE [G]
feature
    ....
end
```

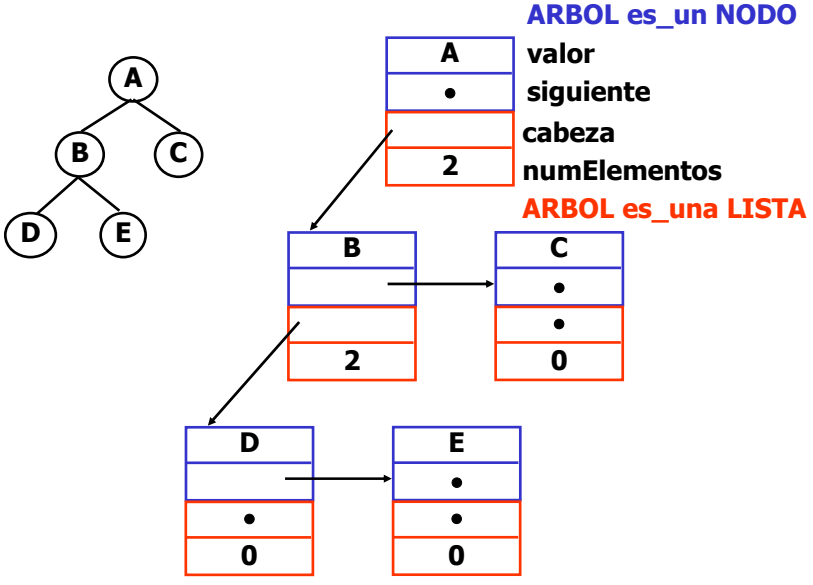
LIST permite obtener los hijos de un nodo, añadir/eliminar un hijo,...

LINKABLE permite obtener el valor de un nodo, su hermanos, su padre, añadir un hermano...

"Un árbol es una lista, la lista de sus hijos, pero también es un elemento potencial de la lista (un subárbol de otro árbol)

"Un árbol es una lista que es también un elemento de la lista"

TREE[G] gráficamente

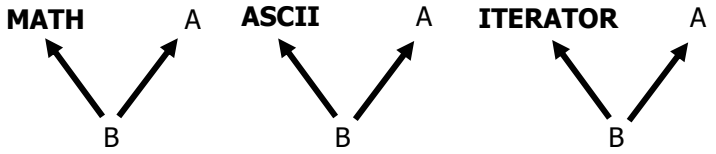


C) Herencia de Estructura:

Se desea que una clase posea ciertas propiedades además de la abstracción que representa.

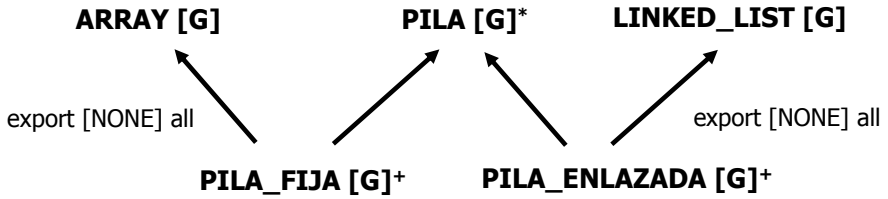
D) Herencia de Facilidades:

Existen clases que existen con el único propósito de ofrecer unos servicios a sus descendientes



B) Matrimonio por conveniencia

Proporcionar una implementación para una abstracción definida por una clase diferida usando las facilidades que proporciona una clase efectiva

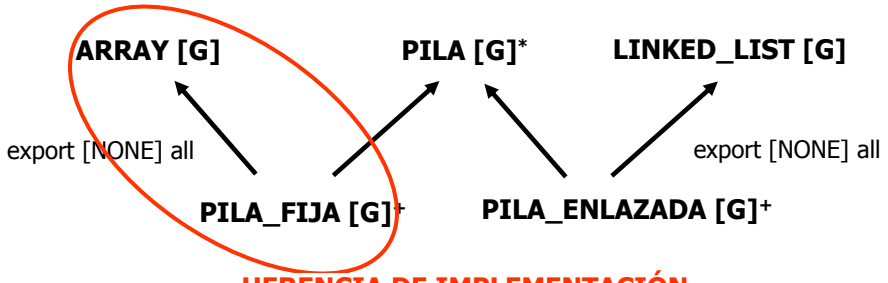


• La clase **PILA_FIJA [G]** sólo exporta las características exportadas por **PILA [G]** y oculta las propiedades de **ARRAY [G].**

• La clase **PILA_ENLAZADA [G]** sólo exporta las características exportadas por **PILA [G]** y oculta las propiedades de **LINKED_LIST [G].**

B) Matrimonio por conveniencia

Proporcionar una implementación para una abstracción definida por una clase diferida usando las facilidades que proporciona una clase efectiva



Implementación de Pilas usando arrays 1/2

```
class PILA_FIJA [G] inherit
    PILA [G]
    ARRAY [G]    export {NONE} all
                 rename
                        put as array_put, make as make_array,
                        count as capacity
                 end
    creation make
    feature
        count: INTEGER;   --hace efectiva como atributo una
                          --característica diferida
        make (n: INTEGER) is
            require   tamaño_no_negativo: n>=0;
            do
                array_make (1,n)
            ensure
                capacidad: capacity = n;
                vacia:  count = 0
            end
```

Implementación de Pilas usando arrays 2/2

```
    full: BOOLEAN is do
          --¿Está llena la representación de la pila?
                Result:= (count = capacity)
    end;

    put (x: G) is
          -- Pone x en la cima de la pila
              require  not full
              do
                  count:= count + 1;
                  array_put (x, count)
    end;

    invariant
          count>=0 ;
          count <= capacity
              ...
end -- PILA_FIJA [G]
```


Problema en ejecución

Sea la declaración

```
p: PILA_FIJA [INTEGER]
```

entonces

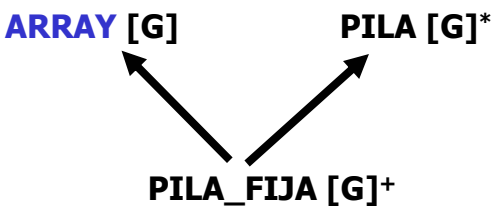
```
p.array_put (32,45)
```

provocaría error en tiempo de compilación.

¿Provocaría error el siguiente código?

```
p: PILA_FIJA [INTEGER]; a: ARRAY [INTEGER]
...
a:= p
a.put (32,45)    {se ejecuta la rutina put de array}
```

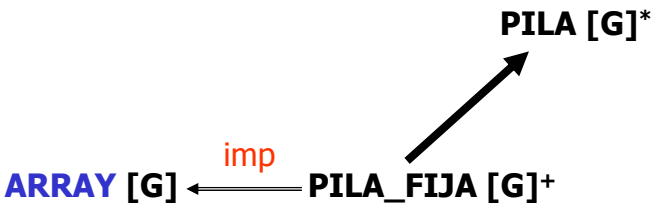
Herencia de implementación




```
full: BOOLEAN is do
        Result:= (count = capacity)
end;

put (x: G) is do
        count:= count + 1;
        array_put (x, count)
end;
```


¿Cómo se haría sin herencia de implementación?




```
full: BOOLEAN is do
        Result:= (count = imp.capacity)
end;

put (x: G) is do
        count:= count + 1;
        imp.put (x, count)
end;
```

☹ Penalización rendimiento

☹ Tedioso uso prefijos

Toda manipulación de la representación requiere una llamada a una característica de **ARRAY** con **imp** como receptor.

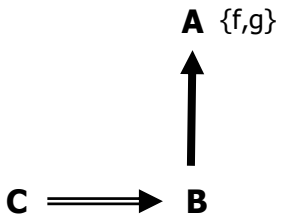
Clientela vs. Herencia de implementación

- "*B hereda de A*" es una decisión de diseño más comprometedora que "*B es cliente de A*".
- Si "*B es cliente de A*" **podemos cambiar la implementación de A sin afectar a B.**

¿Cuándo es apropiada una herencia de implementación?

- "Cuando la elección de la implementación es una parte esencial del diseño de la clase, y no va a cambiar, como en PILA_FIJA [G]"
- En ese caso ofrece una solución más simple, eficiente y legible.

C++ y Herencia de implementación




```
C* oc; A* oa; B* ob
oa = ob     {ERROR}
ob.f( )     {ERROR}
```

```
class A {
    public:
        void f ( );
        void g ();
    protected:...
    private:...}

class B: private A {
    public: ...
    protected:...
    private:...}
```

Ocultar las propiedades del padre

- Eiffel:
  - export {NONE}
  - `class Pila inherit Array export {NONE} ALL`
- C++:
  - herencia privada
  - `class Pila: private Array{...};`
- Java:
  - throw new UnsupportedOperationException();
  - invocar con super a las propiedades del padre
  - ```
    class Pila extends Array{
     public void add(int pos, Object valor) {
           throw new UnsupportedOperationException();
     }
     public void push(Object valor){
           super.add(tope,valor);
     }
    }
    ```


Ejemplo: Figuras Compuestas

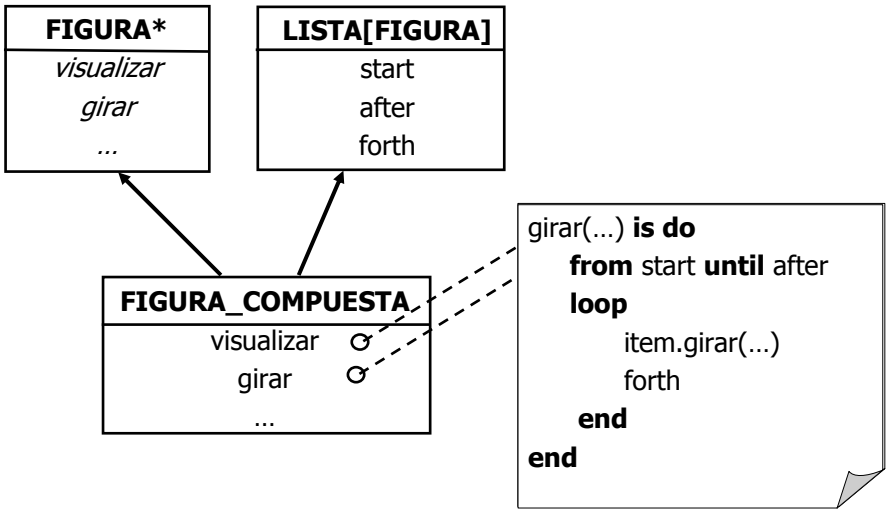
Vamos a ver un **patrón de diseño** general (de utilidad en muchas áreas) que describe estructuras compuestas a través de la herencia múltiple, usando una clase contenedora (lista) como una de sus clases padre, **PATRÓN COMPOSITE**.

```
class FIGURA_COMPUESTA inherit
                              FIGURA
                              LINKED_LIST [FIGURA]
feature
        ...
end
```

"Una figura compuesta es una figura"

"Una figura compuesta es una lista de figuras"
```

Figura Compuesta con herencia múltiple



"Muchas rutinas con esta estructura" (rotar, trasladar, ...)

Patrón Composite: Herencia simple

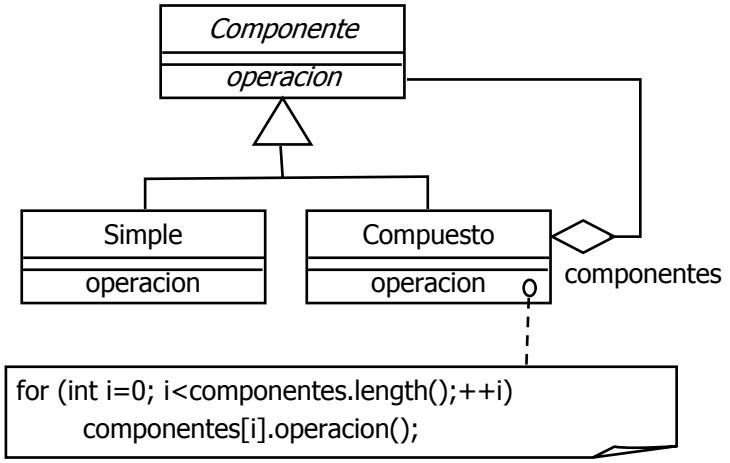
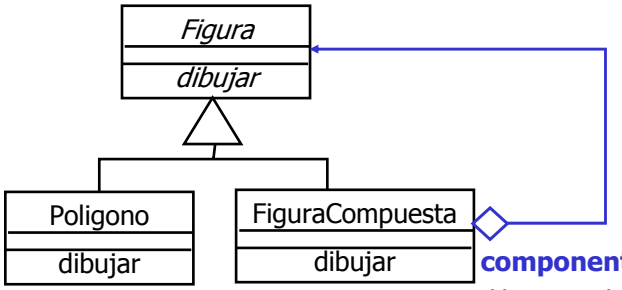


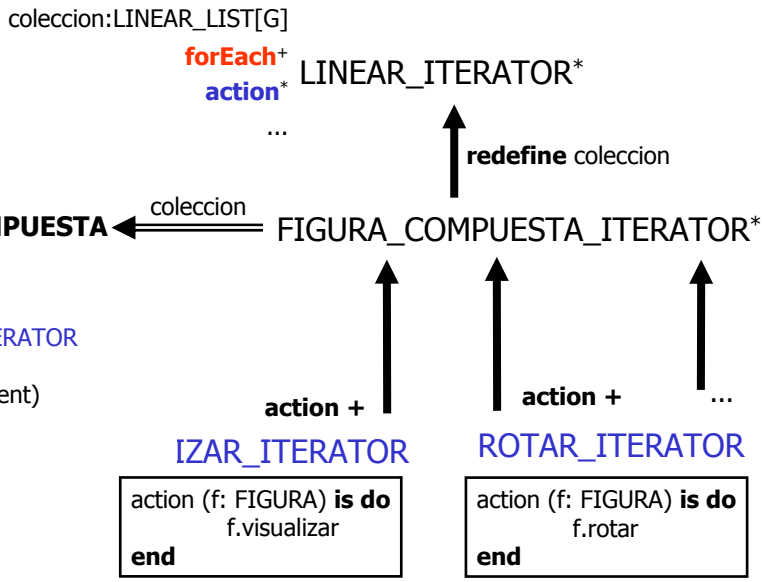
Figura Compuesta con Herencia simple




```
componentes: LINKED_LIST[FIGURA]
dibujar is do
   from componentes.start
   until componentes.after
   loop
      componentes.item.dibujar
      componentes.forth
   end
end
```

vs Herencia múltiple

Solución: Figuras Compuestas e Iteradores




```
rotar is
local
   ir: ROTAR_ITERATOR
do
   !!ir.make(Current)
   ir.forEach
end
```

FIGURAS COMPUESTAS + ITERADORES

```
deferred class FIGURA_COMPUESTA_ITERATOR inherit
        LINEAR_ITERATOR [FIGURA_COMPUESTA]
                redefine coleccion end
creation make
feature
    coleccion: FIGURA_COMPUESTA      -- Colección sobre la que itera
    …                                -- (lista de figuras)
end                                  -- antes era Linear_List[G]
```

```
class VISUALIZAR_ITERATOR inherit FIGURA_COMPUESTA_ITERATOR
                redefine action end
creation make
feature
    action(f: FIGURA) is do
        f.visualizar
    end
end
```

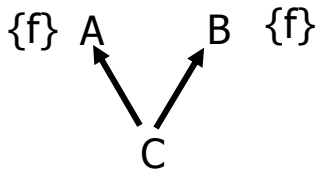
FIGURAS COMPUESTAS + ITERADORES

```
class FIGURA_COMPUESTA inherit
                        FIGURA
                        LINKED_LIST[FIGURA]
feature
    ...
    visualizar is
    local
       iv: VISUALIZAR_ITERATOR
    do
       !!iv.make (Current)      -- colección:= Figura_compuesta actual
       iv.forEach
    end
    ...
end
```

"NO HAY NADA MALO EN TENER CLASES TAN PEQUEÑAS"
"NO ES ACEPTABLE PASAR RUTINAS COMO ARGUMENTOS"
(B. Meyer)

3.-Problemas con la Herencia Múltiple

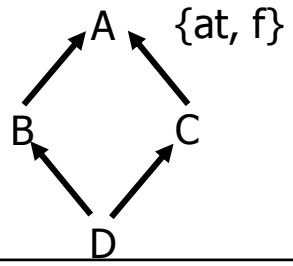
- **(a) Colisión de nombres**



Solución:

Renombrar [y redefinir]

- **(b) Herencia repetida**

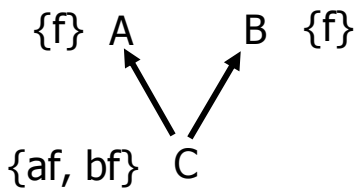


Solución:

¿compartir o duplicar?

Colisión de nombres en Eiffel

- Se considera un problema sintáctico
- Debe ser resuelto por la clase que hereda
- **Solución:** **RENOMBRAR** en la clase C, al menos una de las dos características heredadas que colisionan




```
class C inherit
   A rename f as af
   B rename f as bf
feature
...
end
```

•No hay colisión de nombres si:

i) colisiona una rutina diferida con otra efectiva

ii) hay herencia repetida con compartición

Renombrar & Redefinir

- Importante distinguir entre la versión de una característica y su nombre
  - **Redefinir:** cambiamos la **versión** → Mecanismo semántico
  - **Renombrar**: cambiamos el **nombre** → Mecanismo sintáctico
    - Resolver colisiones de nombres
    - Ofrecer nombres más adecuados.

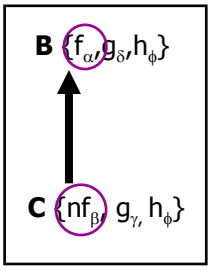
Ejemplo:

```
class WINDOW inherit
      RECTANGLE
      TREE [WINDOW]
         rename
          put as add_window,
          remove as delete_window,
          parent as super_window,...
           end
....
end
```


Renombrar & Redefinir

- Una característica con nombre final **f** en la clase **C** podrá tener un nombre original **g** distinto en la **clase origen A** dónde se introdujo inicialmente.
- Cada característica **f** de una clase **C** tiene un nombre final en **C**, que se define según
  - Si **f** es inmediato:
    nombre final = nombre original
  - Si **f** es heredado de **A** y no se renombra:
    nombre final = nombre final en A
  - Si **f** es heredado de **A** y se renombra
    nombre final = "nuevo nombre"

Combinar redefinir y renombrar




```
class C inherit
    B
      rename f as nf      -- debe ir antes de redefine
      redefine g, nf
    end;
feature
    nf is do .. end;
    g is do .. end;
...
end
```

Clase donde está	Clase origen	Nombre final	Nombre original	Versión
B	B	f	f	α
B	B	g	g	δ
B	B	h	h	ϕ
C	B	nf	f	β
C	B	g	g	γ
C	B	h	h	ϕ

* **Ejercicio 1:**

A {f$_\alpha$}

rename f **as** g

B {g$_\alpha$}

```
oa: A; ob: B
      !!ob
oa:= ob
oa.f          -- {α}
oa.g          -- {error t.c.}
ob.f          -- {error t.c.}
ob.g          -- {α}
```

* **Ejercicio 2:**

A {f$_\alpha$}

rename f **as** g
redefine g

B {g$_\beta$}

```
oa: A; ob: B
      !!ob
oa:= ob
oa.f          -- {β}
oa.g          -- {error t.c.}
ob.f          -- {error t.c.}
ob.g          -- {β}
```

* Ejercicio 3:

B $\{f_\alpha, g_\beta\}$

rename f **as** h
redefine h

C $\{h_\delta, i_\gamma\}$

ob: B; oc: C
!!ob;
ob.f () -- se ejecuta α
ob.g() -- se ejecuta β
ob:= oc
ob.f () -- se ejecuta δ
ob.i () -- error en compilación
oc.h () -- se ejecuta δ
oc.f () -- error en compilación

Clase	Nombre final	Nombre original	Versión	Clase Origen
B	**f**	**f**	α	B
B	**g**	**g**	β	B
C	**h**	**f**	δ	B
C	**g**	**g**	β	B
C	**i**	**i**	γ	C

Ejercicio 4

```
class C inherit
   A
      rename f as g;
   B
      rename f as h;
      redefine h;
feature
    h is do … end
    f  is do … end
end
```

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

 oc.f;
 oc.g;
 oc.h;

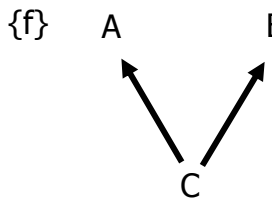
oa:= oc;

 oa.f;
 oa.g;

ob:= oc;
 ob.f;
 ob.h;
```

Colisión de nombres en C++

- No es posible renombrar
- ¿No sería suficiente con la **sobrecarga**?
- **Solución:** Calificación de las rutinas
  - poco adecuada para los clientes
  - elimina la **Ligadura Dinámica** (A::f puede aplicarse desde cualquier clase)



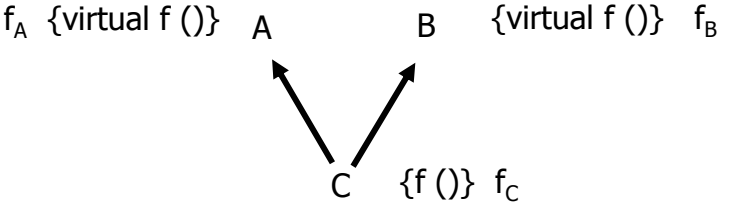
C* oc
c -> A::f ( )
c- > B::f ( )

{f}

```
class C: private A, private B {
    public:
        void af ( ) {return A::f ()};
        void bf ( ) {return B::f ()};
    // reexportar las otras funciones de A y B
    // no renombradas
...
}
```


Colisión de nombres en C++

¿Y si las funciones son **virtual** y quiero conservas ambas porque tienen significados distintos?



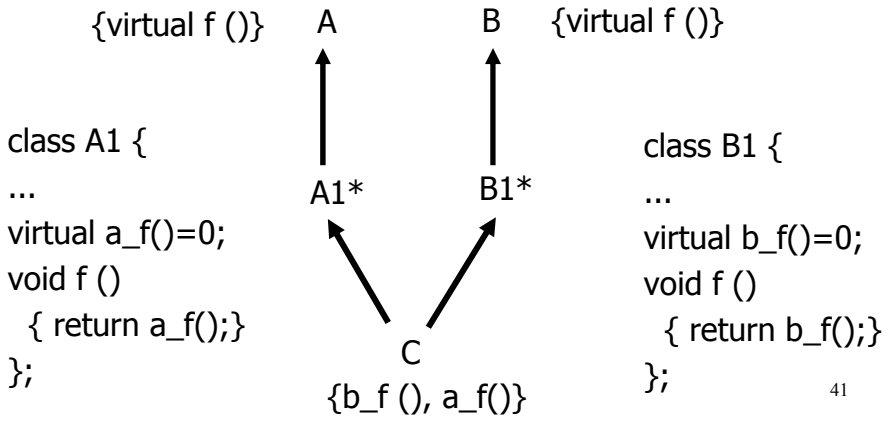
C* pc = new C; A* pa = pc; B* pb = pc

pa -> f (); pb -> f (); pc -> f ()

¡¡¡ Todos los mensajes invocan a C:: f () !!!

Cambio de nombre en C++

Solución: Introducir una **clase extra** por cada clase que tenga una función virtual a la que se quiera cambiar el nombre en la que se define el **nuevo nombre** de la función.



Ejemplo: *quiero redefinir los métodos de las clases bases y mantener los dos.*

```
class Loteria{
   //...
   virtual int dibuja();
};

class ObjetoGrafico{
   //...
   virtual void dibuja();
};
```

Loteria::dibuja() y ObjetoGrafico::dibuja() tienen significados diferentes

```
class Lloteria:public Loteria{
   virtual int l_dibuja()=0;
   int dibuja()
   //redefine Loteria::dibuja
        {return l_dibuja();}
};

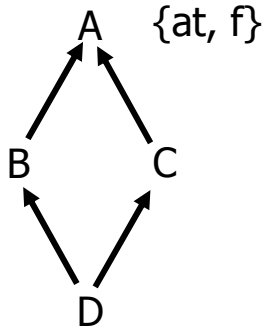
class OObjetoGrafico: public
   ObjetoGrafico{
   virtual int ir_a_dibuja()=0;
   void dibuja()
   //redefine ObjetoGrafico::dibuja
        {ir_a_dibuja();}
};

class SimulaLoteria
   :public LLoteria,
    public OObjetoGrafico{
    //...
    int l_dibuja();
    void ir_a_dibuja();
};
```

Problemas con la Herencia Múltiple:

(a) Colisión de nombres ⇒ rename

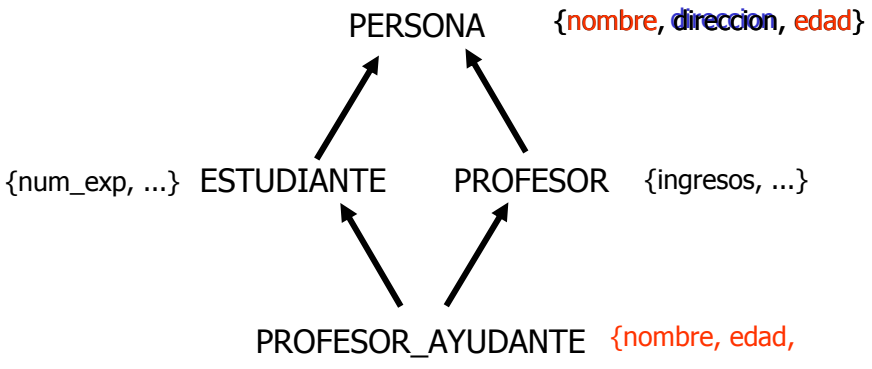
(b) Herencia repetida



¿Qué sucede con las propiedades heredadas más de una vez?

REPLICACIÓN o **COMPARTICIÓN**

Ejemplo: Herencia repetida



- **edad** ⇒ Compartición
- **direccion** ⇒ ¿particular o del Dpto? ⇒ Replicación

Herencia repetida: ¿Replicar o Compartir?

Sea la clase **D** y **B_1, .., B_n** (n≥1) son ascendientes de **D** que tienen la clase **A** como ascendiente común; sean **f_1, .., f_n**, características de **B_1, .., B_n**, respectivamente, que tienen como "semilla" la propiedad **f** de **A**, entonces:

1) Cualquier subconjunto de **f_1, .., f_n** heredado **bajo el mismo nombre** final genera una única característica en **D**

(COMPARTICIÓN)

2) Cualesquiera dos de las **f_1, .., f_n** heredadas **bajo diferente nombre**, generan características diferentes en **D**

(REPLICACIÓN)

El primer caso es lo que normalmente se necesita

Herencia repetida: ¿Replicar o Compartir?

Sea la clase **D** y **B_1, .., B_n** (n≥1) son ascendientes [illegible] que tienen la clase **A** como ascendiente común; sean **f_1, .., f_n**, características de **B_1, .., B_n**, respectivamente, que tienen como "semilla" la propiedad **f** de **A**, entonces:

atributos y rutinas

1) Cualquier subconjunto de **f_1, .., f_n** heredado **bajo el mismo nombre** final genera una única característica en **D**

(COMPARTICIÓN)

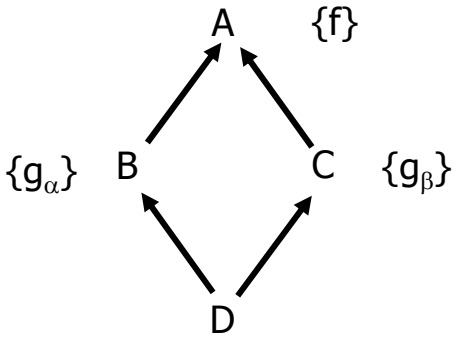
2) Cualesquiera dos de las **f_1, .., f_n** heredadas **bajo diferente nombre**, generan características diferentes en **D**

(REPLICACIÓN)

El primer caso es lo que normalmente se necesita

Regla del nombre único

Dos características **efectivas** diferentes de una misma clase NO pueden tener el mismo nombre final.



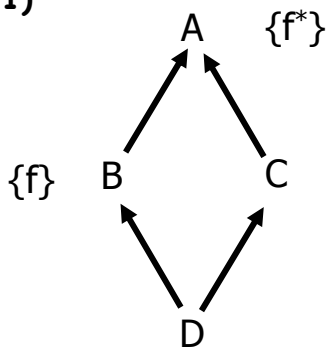
- f no provoca conflicto (compartición)
- g causa conflicto

Ocurre un **conflicto de nombres** si una clase hereda dos características **diferentes**, **ambas efectivas**, con el mismo nombre.

b.1) Conflicto cuando hay compartición

Si se heredan dos características bajo el mismo nombre puede ocurrir:

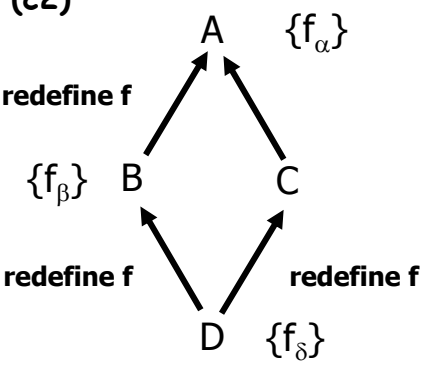
(c1)



LEGAL

Solo hay conflicto entre dos efectivas ▷◁

(c2)

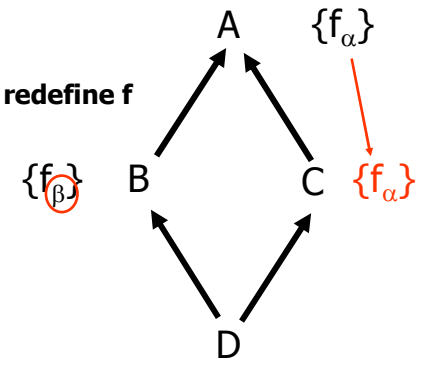


LEGAL

Ambas versiones se funden en una nueva

Conflictos cuando hay compartición:

(c3) Ambas versiones efectivas y ambas no se redefinen:



ILEGAL

Viola la Regla del nombre único

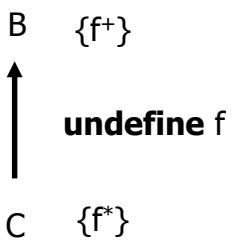
Soluciones:

- **rename** ⇒ replicación
- **undefined** ⇒ (c1)

Conversión en diferida

- Dejar que una de las variantes se imponga sobre las otras
- Es posible al heredar transformar una rutina efectiva en diferida.

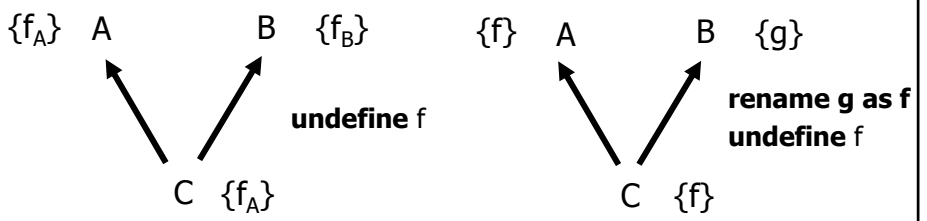
```
class C inherit
      B
       undefine f
feature
      ...
end
```



- Viene después de **rename** (se aplica al nombre final) pero antes de **redefine**.

Indefinición y unión

- El mecanismo **undefine** proporciona la capacidad de unir características bajo herencia múltiple (no necesariamente repetida).
- **Ejemplo:** Se desea que C trate a f y g como una única característica (requiere semántica y signatura compatibles)

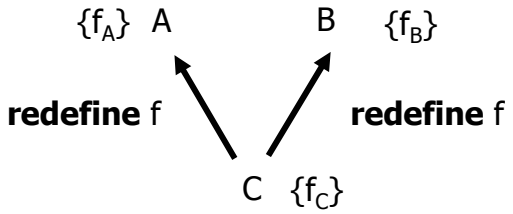


A impone la característica

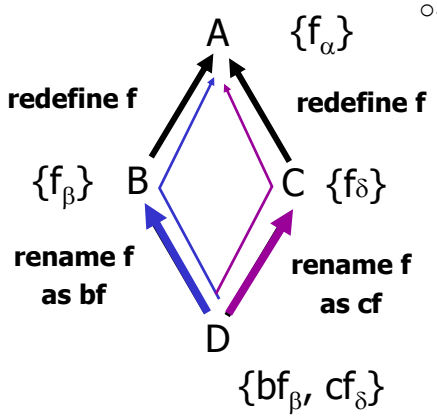
Combinación de propiedades

Todas las restantes combinaciones son posibles:

- tomar la característica de A y el nombre de B
- renombrar f y g y darle un nombre nuevo en C
- Reemplazar ambas versiones por una nueva (caso (c2))
  - ambas tienen el mismo nombre final (utilizar rename si no)



b.2) Conflicto cuando hay replicación




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

ob:= od;
ob. f          {β}

oc:= od;
oc. f          {δ}

oa:= od
oa.f       {β o δ}
```

No hay conflicto de nombres pero surge un nuevo problema debido a la **ligadura dinámica**.

Solución al conflicto con la replicación

```
class D inherit
      B
         rename f as bf;
         select bf          -- elimina ambigüedad
      C
         rename f as cf;
         select cf          -- elimina ambigüedad
feature
      ...
end
```

Regla del Select

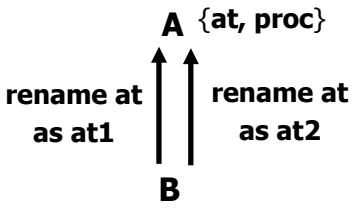
Una clase, que de un antecesor repetido, hereda dos o más versiones efectivas diferentes de una característica y no las redefine a ambas, debe entonces incluir a una de las dos en una cláusula select

La clausula **select** debe aparecer después de rename, undefine y redefine.

Ejemplo: Conflicto con la replicación

```
class A feature
   at: INTEGER
   proc is do
      print(at)
   end;
end

class B inherit
   A
      rename at as at1;
      select at1
   A
      rename at as at2
feature
   ...
 end
```

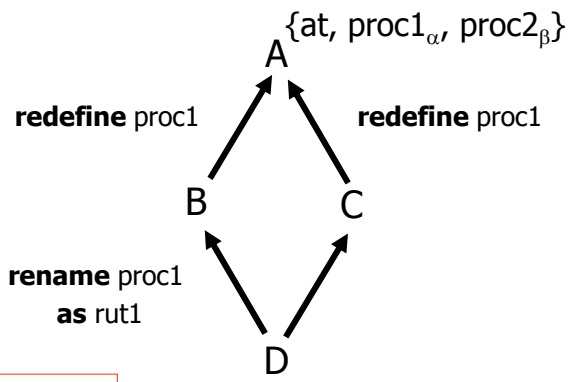



```
oa: A; ob: B
!!ob
oa:= ob
oa. proc
```

¿se imprime at1 o at2?

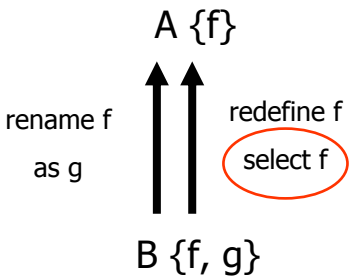
Ejemplo: Conflicto con la replicación

```
class A feature
   at: INTEGER
   proc1 is do .. end;
   proc2 is do
        proc1
        print(at)
   end;
end
```




```
oa:A; od: D;
!!od;
oa:=od;
oa. proc2 ??
```

Utilidad de la herencia repetida: mantener la versión original junto con la redefinida.

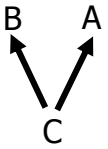



```
class B inherit
   A
      redefine f
      select f
   A
      rename f as g
feature
   ...
end
```

- Antes de introducir **`Precursor`** era la técnica usada para que una rutina redefinida pudiese invocar a la rutina original.

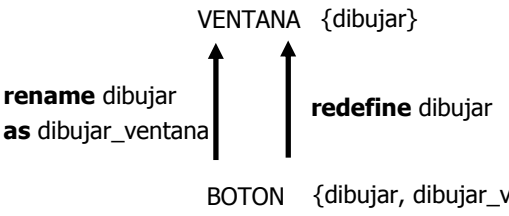
Entidad "Precursor" (Eiffel)

- ¿Que sucede si hay herencia múltiple y una rutina tiene varios precursores ?



{A} Precursor
{B} Precursor

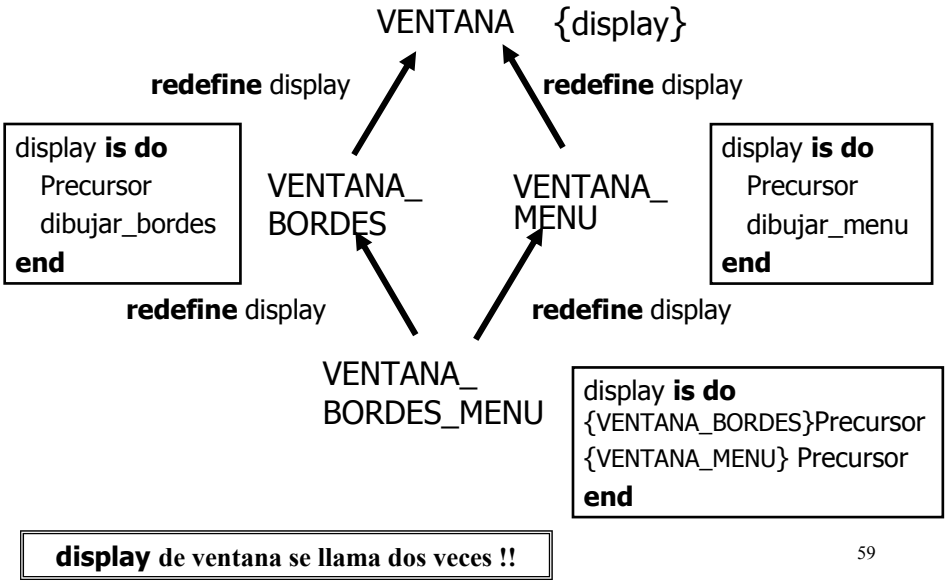
- ¿Que hacemos si queremos disponer en la nueva clase de la rutina original y la redefinida?



¡¡OJO!! AMBIGÜEDAD
(ver Herencia Repetida)

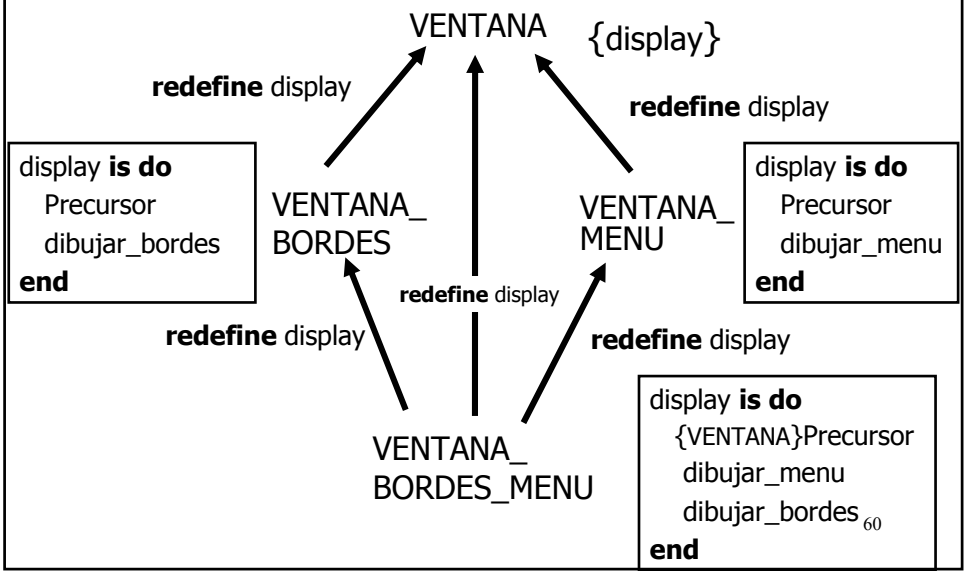
¿En JAVA?

Ejemplo:



display de ventana se llama dos veces !!

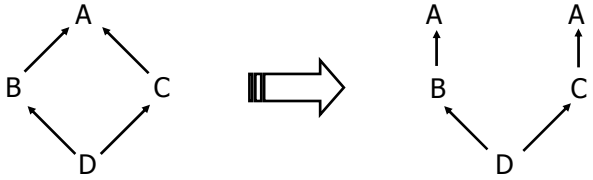
Solución: no será necesario el uso de la cláusula select en VENTANA_BORDES_MENU ¿por qué?



Herencia repetida en C++: Replicación

• Por defecto, **NO HAY COMPARTICIÓN, SE REPLICA TODO**

```
class A {int at1; ...}
class B: public A {...};
class C: public A {...}
class D: public B, public C {...}
```



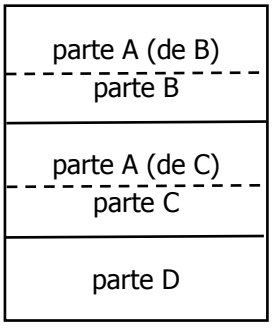
•El nivel de granularidad para decidir si compartir o duplicar es la **clase**

• **Ambigüedad** se resuelve con **calificación de nombres**:

C::at1; B::at1

Herencia repetida en C++

- Hay dos objetos A en un objeto D
- La asignación entre un puntero a A y otro a D no está permita, es ambigua



Estructura del objeto D

```
D* od = new D();
A* oa = od;          //ERROR: ambiguo

oa=(A*)od;           //ERROR: ambiguo

oa=(A*)(B*)od;       //OK!!
                     //asigna el
                     //subobjeto A de B
```

Herencia repetida en C++: Replicación

class B {public: int b; ...}

class D: **public** B, **public** B {...} {**ILEGAL**}

No se pueden resolver las ambigüedades.


```
void f (D *p)
{
        p->b=7;          //ambiguo
}
```


Herencia repetida en C++: Compartición

Si queremos que la clase derivada herede sólo una copia de la clase "abuelo" entonces las clases intermedias tienen que declarar que su **herencia** de la clase paterna es **virtual**.

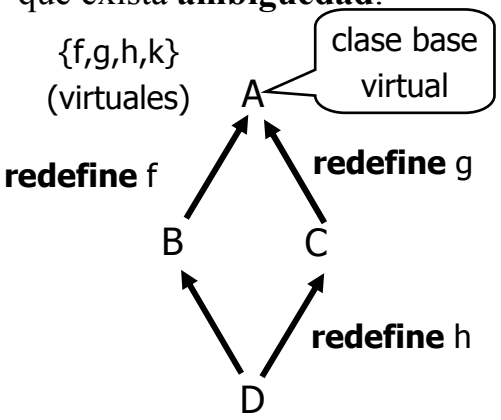
Clase base virtual

```
class A {char* at1; int f (); ...}
class B: virtual public A {...};
class C: virtual public A {...}
class D: public B, public C {...}
```

- Significado distinto en este contexto que en el de las funciones
- **No** significa que todas las funciones son virtuales

Mecanismo que se opone a la extensibilidad

Cuando se utilizan **clases base virtuales** se puede llegar a la misma característica a través de mas de un camino **sin** que exista **ambigüedad**.




```
D* od= new D

od->f();      //B::f()
od->g();      //C::g()
od->h();      //D::h()

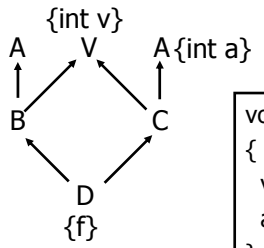
C* oc = od

oc->f();      //B::f()
```

- La versión redefinida **domina** a la virtual
- La misma situación con clases base no virtuales produce **ambigüedad** (implica más de un objeto).

Herencia repetida en C++ y clases virtuales

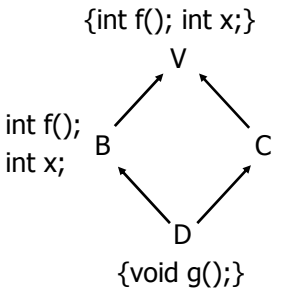
```
class V {public: int v};
class A {public: int a};
class B:public A, public virtual V {};
class C:public A, public virtual V {};
classD : public B, public C {
   public: void f();};
```




```
void D::f()
{
  v++; //correcto solo hay un 'v'
  a++; //ambiguo hay dos 'a'
}
```

```
class V {public: int x; int f();};
class B: public virtual V {
      public: int x; int f();};
class C: public virtual V {};
classD : public B, public C {void g();};

void D::g()
{
    x++;   //OK!  B::x domina a V::x
    f();   //OK!  B::f() domina a V::f()
}
```



Resumen Adaptaciones en Eiffel

- Redefinir **(redefine)**
- Renombrar **(rename)**
- Resolver ambigüedades en los nombres de las características
- Hacer efectiva una rutina
- Cambiar el tipo de un atributo o un parámetro de una rutina
- Combinar características ("join")
- Cambiar el status de visibilidad de una característica **(export)**
- Convertir en diferida una rutina efectiva **(undefine)**
- Eliminar conflictos de ligadura dinámica cuando existe herencia repetida **(select)**

4.-Herencia múltiple en Java: Interfaces

- Java soporta **herencia simple**.
- Herencia múltiple:
  - útil cuando una clase quiere añadir nuevo comportamiento y conservar parte del antiguo
  - origina problemas: Herencia repetida (herencia de implementación).
- Solución Java: **herencia múltiple de interfaces**.
- Java permite que una clase pueda heredar de más de una INTERFACE
- **Interface** es el modo de declarar un tipo formado sólo por ***métodos abstractos (públicos***) y ***constantes***, permitiendo que se escriba cualquier implementación para estos métodos.

Ejemplo: "Interfaces" en Java

```
interface Pila {
  void push(Object elem);
  void pop;
  Object top;
  boolean empty;
  boolean  full;
}
```

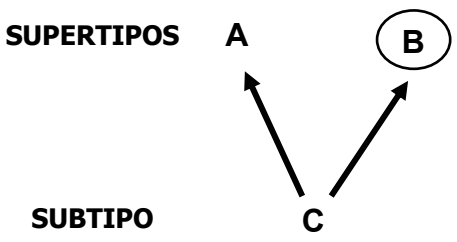
implícitamente **abstract**
siempre **public**

Varias implementaciones posibles

class PilaFija **implements** Pila {..}

class PilaVariable **implements** Pila {..}

"Interfaces" Java




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

Puede utilizarse como nombre de tipo

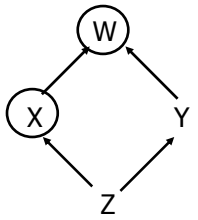
Puede utilizarse polimórficamente

Se le puede asignar cualquier obj que implemente la interfaz

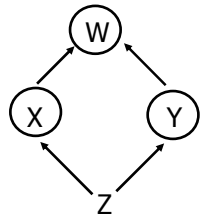
- Surgen colisiones de nombre
  1) Sobrecarga
  2) Compartición
  3) Ilegal (si difieren en el tipo del objeto que retorna)

Herencia de interfaces

- A diferencia de las clases una interfaz puede heredar de más de una interfaz
- Se utiliza herencia múltiple si se quiere que una clase implemente un interfaz y herede de otra clase




```
interface W { }
interface X extends W { }
class Y implements W { }
class Z extends Y implements X { }
```




```
interface W { }
interface X extends W { }
interface Y extends W { }
class Z implements X, Y { }
```


Implementación de una Interface

- Una interfaz sólo es interesante si una clase la implementa:
  - Un conjunto de implementaciones generales agrupadas en un paquete
  - Implementación específica de cada clase

Utilización de una Interface

- Simplemente heredando de la clase que implementa el interfaz
- ¿Que hacemos si una clase hereda de otra y desea utilizar cierta implementación de una interface?

!NO PODEMOS HEREDAR DOS VECES!

Uso de interface en Java

Solución: crear un objeto de la clase de implementación y remitirle los métodos de la interface

```
class B extends A implements X {
   Ximplem at1;
   …
   public void metodo1(..) {at1.metodo1(..)}
   public int metodo2(..) {return at1.metodo2}
   …
}
class Ximplem implements X {…}
```


Ejemplo: uso de interfaces 1/2

- Abstracción de una máquina que se mueve con motor

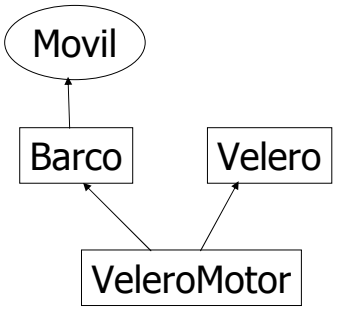
```
interface Movil{
   void enMarcha();
   void alto();
   void girar(double grados);
   void cambiaVelocidad(double kmHora);
   boolean quedaCombustible();
}
```

- Un Barco que se mueve a motor

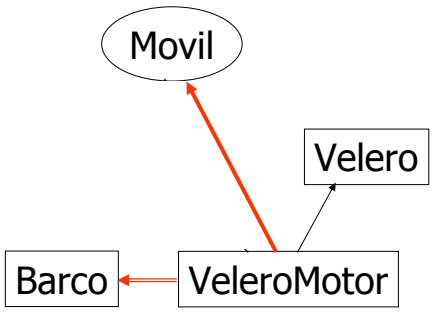
```
public class Barco implements Movil { … }
```

- Suponiendo que existe la clase `Velero` ¿Cómo modelaríamos un VeleroMotor que debe heredar de `Barco` y de `Velero`?

Ejemplo: uso de interfaces 2/2



Ejemplo: uso de interfaces 2/2




```
class VeleroMotor extends Velero
                        implements Movil{
    Barco b;

    void enMarcha(){
          b.enMarcha();
    }

    …

    boolean quedaCombustible(){
          return b.quedaCombustible();
    }
}
```

"Interfaces" vs clases abstractas

- **Una interface no es una clase abstracta**.
  - Una clase abstracta puede estar parcialmente implementada.
  - Una clase puede heredar de una única clase, incluso si sólo tiene métodos abstractos.
- **Importante**:
  - Es conveniente que el programador Java declare interfaces para las clases que cree, ya que NO EXISTE HERENCIA MULTIPLE.

Conclusión: OO y Objetivos Iniciales

La combinación de **clases**, **genericidad**, **herencia**, **redefinición**, **polimorfismo**, **ligadura dinámica** y **clases diferidas** permite satisfacer:

- los principios/ reglas/ criterios de *modularidad*
- los requisitos para la *reutilización*

planteados en el primer tema.