TEMA 3

Herencia
(2ª parte)

Facultad de Informática

Universidad de Murcia

Índice

1.- Introducción

2.- Adaptaciones del lenguaje:

- Redefinición vs. Renombramiento
- Cambios en el nivel de visibilidad

3.- Herencia y creación

4.- Polimorfismo y ligadura dinámica

5.- Genericidad

- Estructuras de datos polimórficas
- Genericidad restringida

6.- Clases abstractas/diferidas

1.-Introducción

- La herencia es un mecanismo que posibilita la definición de una clase a partir de otra dando soporte para registrar y utilizar las relaciones de especialización o generalización existentes entre las clases.

- La herencia organiza las clases en una ***estructura jerárquica*** que puede tener o no una clase raíz:
  - Java: clase `Object`
  - Eiffel: clase `ANY`
  - C++: no existe una clase raíz
  - C#: clase `System.Object`

1.- Introducción

- **Terminología**:
  - B hereda de A
  - **Java:**
    - B es *subclase* de A
    - A es *superclase* de B
  - **Eiffel:**
    - B es *descendiente* de A
    - A es un *ascendiente* de B
  - **C++:**
    - B es una *clase derivada* de A
    - A es la *clase base* de B
  - **C#:**
    - B es la *clase hija* de A
    - A es la *clase padre* o clase base de B

2.- Adaptaciones del lenguaje

- Cuando una clase B hereda de una clase A:
  - B puede añadir nuevos atributos
  - B puede añadir nuevos métodos
  - B puede **redefinir** los métodos de A
    - Refinamiento vs. Reemplazo
  - B puede **renombrar** los métodos de A
  - B puede implementar métodos abstractos de A
  - B puede **cambiar el nivel de visibilidad** de los atributos y métodos heredados
- Las adaptaciones dependen del lenguaje

Adaptaciones en Eiffel

- **Renombramiento:**
  - Cambiamos *el nombre* de la característica
  - Es un mecanismo sintáctico
  - Cláusula **`rename`**
  - Utilidad:
    - Solucionar las colisiones de nombres
    - Proporcionar nombres más adecuados
- **Redefinición:**
  - Cambiamos *la versión* de la característica
  - Es un mecanismo semántico
  - Cláusula **`redefine`**
  - Utilidad:
    - Refinamiento
    - Reemplazo

Ejemplo renombramiento

```
class Pila
     inherit Array
        rename
          put as array_put,
          remove as array_remove,
         end
     ....
end
```

Renombrar & Redefinir

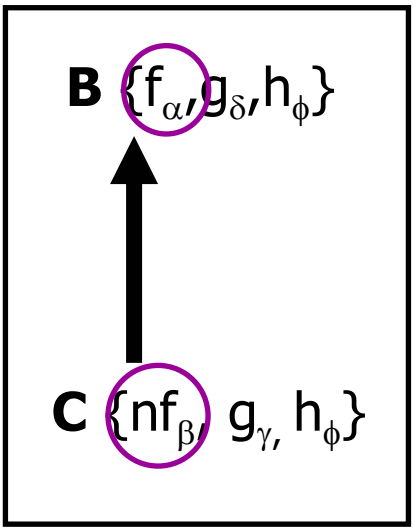
- Una característica con nombre final **f** en la clase **C** podrá tener un nombre original **g** distinto en la **clase origen A** dónde se introdujo inicialmente.
- Cada característica **f** de una clase **C** tiene un nombre final en **C**, que se define según
  - Si **f** es inmediato:

    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"

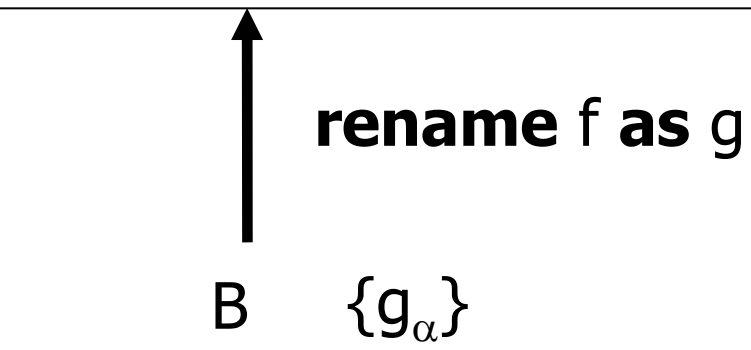
Combinar redefinir y renombrar




```
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
```

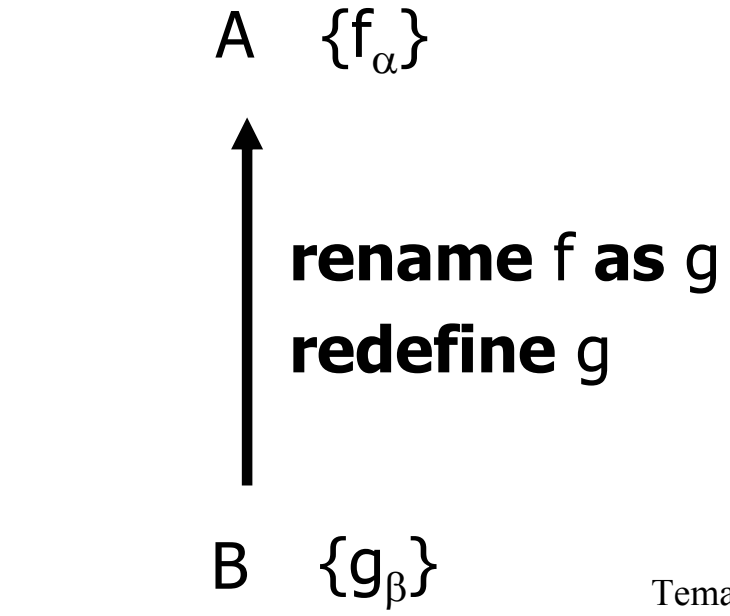
Clase donde está	Clase origen	Nombre final	Nombre original	Versión
B	B	f	f	α
B	B	g	g	δ
B	B	h	h	ϕ
C	B	nf	f	β
C	B	g	g	γ
C	B	h	h	ϕ

Ejercicio 1:




```
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:




```
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;
```

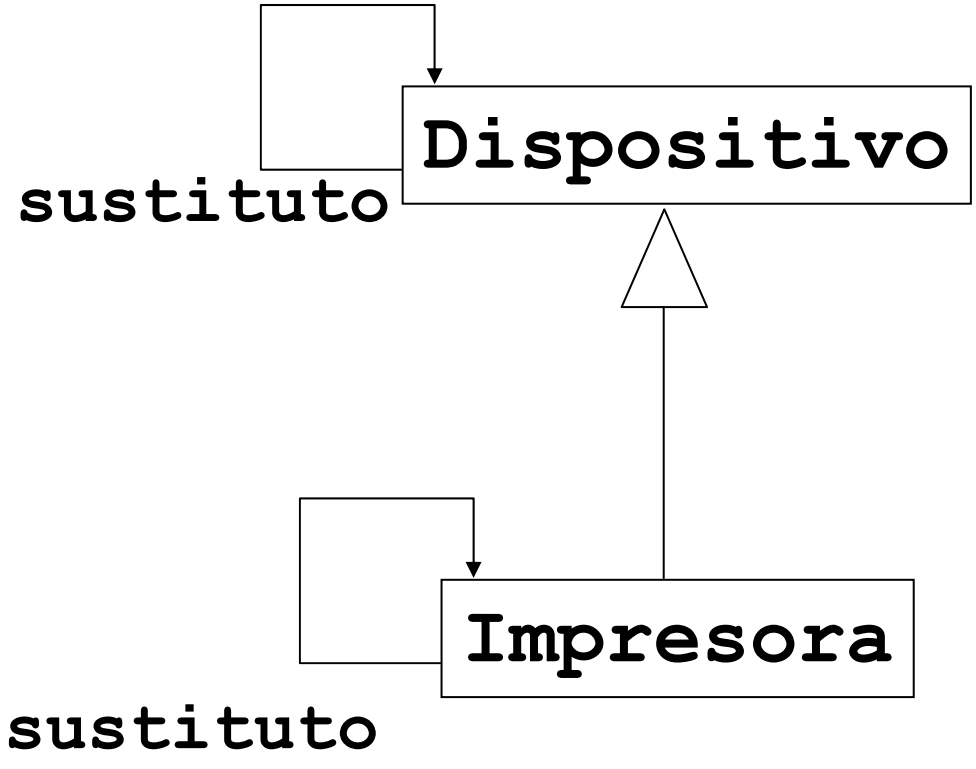
Redefinición en Eiffel

- **¿Qué podemos cambiar?**
  - Tipo de las características (Regla covariante)
  - Implementación
- Solicita explícitamente mediante la cláusula **redefine**.
  - No es necesario incluirlo cuando se redefine para hacer efectiva una característica.
- **Técnicas que le añade potencia**:

a) Redefinir una función como un atributo
  - Aplicación del Principio de Acceso Uniforme
  - Un atributo no se puede redefinir como una función

b) El uso de la versión original en una redeclaración
  - "Facilita la tarea al que redefine si el nuevo trabajo incluye el viejo".

Ejemplo: Regla covariante en Eiffel



Regla de Redeclaración de Tipo (Eiffel)

Una redeclaración de una característica puede reemplazar el tipo de la característica (si es un atributo o una función), o el tipo de un argumento formal (si es una rutina) por cualquier tipo compatible con el original.

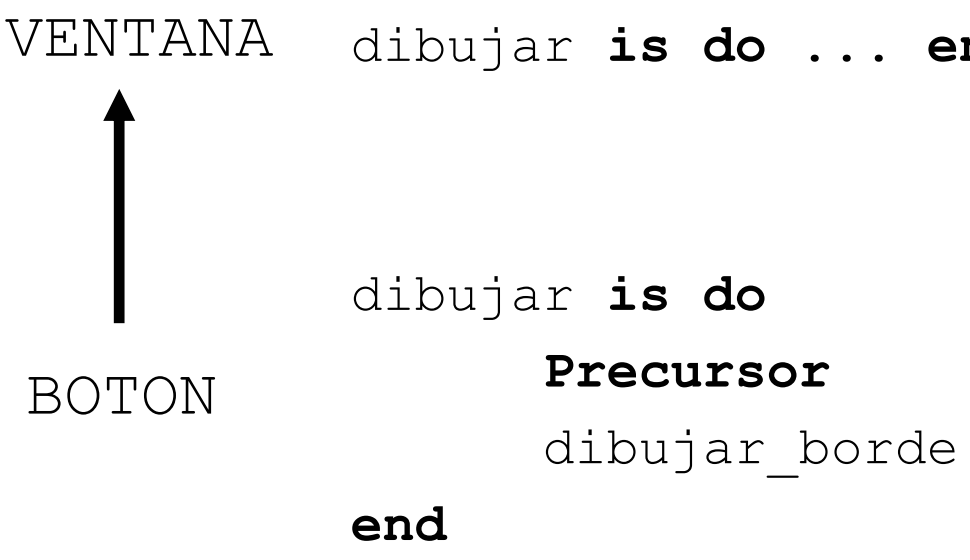
Ejemplo: Redefinir una función como un atributo

```
class Cuenta1 feature
    saldo: INTEGER is do
          Result:= ingresos.total -reintegros.total
    end
...
end


class Cuenta2 inherit
    Cuenta1 redefine saldo end
feature
    saldo:INTEGER
...
end
```

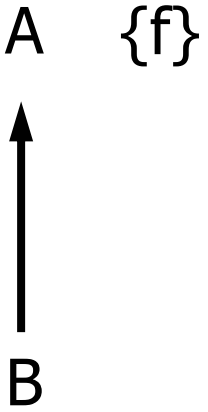
Uso de la versión original en Eiffel

- Entidad **`Precursor`** = llamar a la versión de esta característica en la clase padre.
- Su uso **está limitado** al cuerpo de las rutinas redefinidas
  - **¿Equivale a `super` en Java?**
- **Ejemplo:** Dibujar un botón es mostrarlo como una ventana y luego dibujar el borde



Herencia y Ocultación de Información en Eiffel

- Son mecanismos independientes
- Una clase B es libre de exportar o de esconder una característica f que hereda de un antecesor A.



f exportada en A y B

f privada en A y B

f exportada en A y oculta en B

f oculta en A y exportada en B

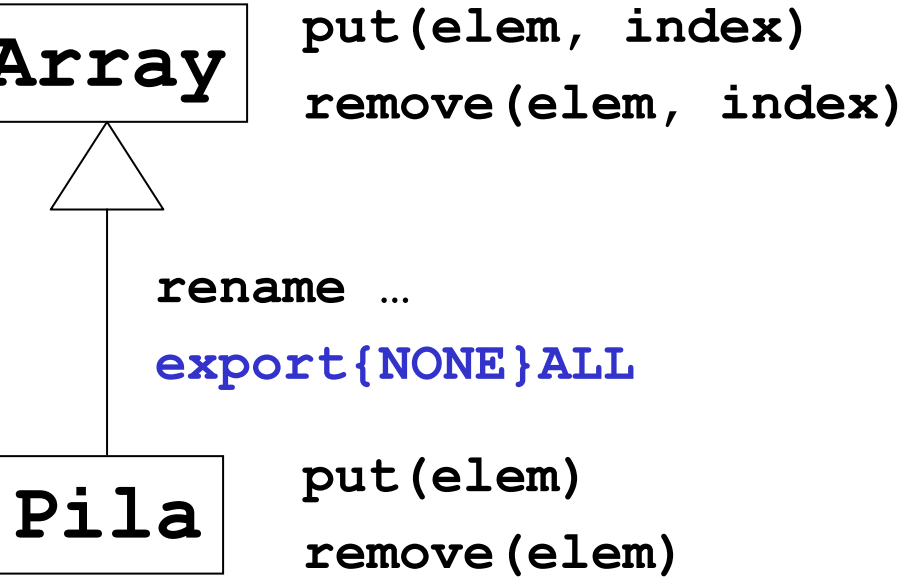
- Por defecto f mantiene el status de exportación que tenía en A, pero se puede cambiar mediante la clausula **`export`**.

¿Por qué dejar que cada descendiente escoja la política de exportación en Eiffel?

- Flexibilidad y Extensibilidad:
  - *La herencia es la clave del Principio Abierto-Cerrado*
- La propiedad básica de la herencia es permitir definir descendientes de una clase no previstos en su creación.
- ¿Cómo sabemos a priori que propiedades exportará o no una subclase no prevista?
- La herencia sólo debe estar limitada por los asertos y las reglas de tipado.

"Agujero de tipos" en Eiffel

- Violación del sistema de tipos debido a las asignaciones polimórficas



Sea la declaración:

```
p: PILA[INTEGER]

p.array_put(32,45) //Error tc
```

¿Provocaría un error el siguiente código?

```
a: ARRAY [INTEGER]
...
a:= p
a.put (32,45)
```

¿Cuál es la alternativa a la herencia de implementación?

Redefinición en C++

Clase base contiene una función virtual vf

Clase que se deriva de ella contiene una función vf del mismo tipo (si son distinto se consideran funciones diferentes y no se invoca al mecanismo virtual)

```
lass Base {
  virtual void vf1();    //Ligadura dinámica
  virtual void vf2();
  virtual void vf3();
          void f();      //Ligadura estática

lass Derivada : public Base {
  void vf1();        //redefine vf1
  void vf2 (int);    //distinta a vf2()
  char vf3();        //error, difiere tipo devuelto
  void f();
```

```
void g()
{
Derivada *d;
Base* b;
            ...
b = d;
b -> vf1();          //Derivada::vf1
b -> vf2();          //Base::vf2
b -> f();            //Base::f
d-> vf2();           //ERROR
            ...
}
```

Llamada al padre calificando las rutinas. Ej: **Base :: f()**

Herencia y Ocultación de Información en C++

- **protected**:
  - sólo accesibles por la clase y **sus descendientes**
- Tipos de herencia:
  - **Herencia privada:** `class B: private A {...}`
    - todas las propiedades de **A** se heredan como privadas
    - se puede **mantener** el estatus calificando la rutina
  - **Herencia pública:** `class B: public A {...}`
    - se mantiene el status de las propiedades heredadas (por defecto)
    - un miembro `private` o `protected` no puede ser re-exportado, es decir, `protected` no puede ser `public` y `private` no puede ser ni `protected` ni `public`
    - "agujero" debido asignaciones polimórficas (público puede pasar a privado)

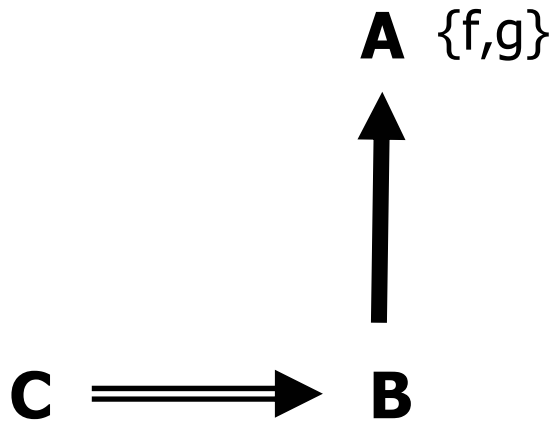
Herencia y exportación en C++

Se puede conservar el acceso si la herencia es privada, mencionando su nombre calificado en la parte `public` o `protected`. Por ejemplo:

```
class B{
     int a; //private
   public:
     int bf();
   protected:
     int c;
};
class D: private B{
   public:
     B::bf();  //hace que bf sea público en D
     B::a;     //ERROR intenta conceder acceso
   protected:
     B::bf();  //ERROR intenta reducir acceso
};
```

Todos los miembros de B son privados en D

C++ y Herencia de implementación




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

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

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

Redefinición en C#

```
class Punto2D {
  ...
 public virtual string ToString(){
     return "[" + x+ "," + y + "]";
 }
  ...
}
class Punto3D : Punto2D {
  ...
 public override string ToString(){
     return base.ToString() + "[z=" + z +"]";
 }
}
```

- Si se nos olvida poner `override` se entiende que estamos ocultando el método de la clase padre.
- Toda redefinición de un método debe mantener el mismo nivel de acceso que tuviese el método original.

Redefinición y ocultación en C# (1/2)

```
using System;

class A {
   public virtual void F(){ Console.WriteLine("A.F");
}
class B : A {
   public override void F(){ Console.WriteLine("B.F");
}
class C : B {
   new public virtual void F(){Console.WriteLine("C.F");
}
class D : C {
   public override void F(){ Console.WriteLine("D.F");
}
```

Redefinición y ocultación en C# (2/2)

```
class TestOcultacionRedefinicion{
   public static void Main(){
        A a = new D();
        B b = new D();
        C c = new D();
        D d = new D();

        a.F(); ──────────────────→ B.F
        b.F(); ──────────────────→ B.F
        c.F(); ──────────────────→ D.F
        d.F(); ──────────────────→ D.F
   }
}
```

Ocultar las propiedades del padre

Eiffel:

- `export {NONE}`
- `class Pila inherit Array export {NONE} ALL`

C++:

- herencia privada
- `class Pila: private Array{...};`

C#:

- Método `new`

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);
   }
```

3.- Herencia y creación en Eiffel

Regla de creación en la herencia

En Eiffel el status de creación de una característica heredada de la clase padre no tiene que ver con su status de creación en la clase heredada.

- Si la subclase añade atributos también tienen que inicializarse.
- La subclase puede reforzar el invariante.
- Si se puede aplicar el procedimiento de creación del padre se indica en la clausula **`creation`**.
- Instrucciones de creación básicas: **`!!x`**

  `!!x.rutina_creacion`

Ejemplo: Herencia y creación en Eiffel

- **Creación polimorfa:** Se debe poder permitir la creación directa de objetos de un tipo descendiente

!C!x.crear

```
fig: FIGURA; tipo_icono_sel: INTEGER; p1, p2, p3: PUNTO;

tipo_icono_sel:= interface. icon_selected (mouse_position);
inspect tipo_icono_sel
   when segmento_icono  then !SEGMENTO! fig. make (p1,p2)
   when circulo_icono   then !CIRCULO! fig. make (p1,radio)
   when triangulo_icono then !TRIANGULO! fig. make (p1,p2,p3)
   ...
end;
      fig. visualizar
```

Herencia y Creación en C++

- Un constructor de clase derivada siempre llamará primero a constructor de su clase base para inicializar los miembros de su clase padre.
- **Ejemplo:**

```
//Punto.cpp (x,y)
    ...
Punto::Punto (float a, float b){  //constructor
    x=a; x=b;
}

//Circulo.cpp;
class Circulo: public Punto {
    …
Circulo::Circulo(float r, float a, float b):Punto(a,b)
//llamada al constructor de la clase base
{
    radio=r;
}
```

Herencia y creación en C#

- Añadir al constructor de la clase hija
  - `: base(<parametrosBase>)`

- Ejemplo:

```
class Empleado: Persona {
   public int Sueldo;

   Empleado(string nombre, string nif, int sueldo):
                                                base(nombre,nif)
   {
        Sueldo = sueldo;
   }
```

- Si se omite, la llamada implícita es al constructor por defecto
  - `:base()`

4.- Polimorfismo y ligadura dinámica

- Formas de polimorfismo (1/2):
  - **Polimorfismo de asignación (*variables polimorfas*)**
    - Sólo se permite para entidades destino de tipo referencia (punteros en C++)
    - Distinguimos entre tipo estático y tipos dinámicos (restringido por la herencia)
  - **Polimorfismo puro (*función polimorfa*)**
    - En Eiffel y Java la ejecución depende siempre del tipo dinámico de la entidad
    - C++ y C# depende de la definición del método (virtual).
  - **Polimorfismo ad hoc (*sobrecarga*)**
    - Eiffel no soporta el mecanismo de sobrecarga
    - Entre C++ y Java/ C# existen ligeras diferencias

Formas de polimorfismo (2/2)

- **Polimorfismo de inclusión (*redefinición*)**
  - En Eiffel y C# hay que hacer explícita la redefinición de métodos (cláusula `redefine` y `override` respectivamente)
  - En C++ y C# hay que indicar explícitamente que un método puede redefinirse (virtual).

- **Polimorfismo paramétrico (*genericidad*)**
  - Java, Eiffel, C++ y C# incluyen la genericidad como elemento del lenguaje.

Sobrecarga en C++, Java y C#

- En los lenguajes OO puede existir sobrecarga
  - dentro de una clase: C++, Java y C#
  - entre clases no relacionadas (es fundamental)
- En C++ existe sobrecarga si dos funciones se encuentran definidas en el mismo ámbito (clase)

```
class B{
    public void f(int v);
}
class D: B{
    public void f(float v);
}
```

  - f no está sobrecargada en la clase `D`
  - la `f` de `D` oculta a `f` de `B`
- En Java y C# si estarían disponibles las dos funciones en la clase `D`

Ligadura dinámica

- Tanto en Eiffel como en Java, se toma la ligadura dinámica como política.
  - La ligadura estática se considera una optimización del compilador
- En C++ y C# por defecto la ligadura es estática.
- Se tiene que declarar explícitamente con la palabra reservada `virtual` sobre qué funciones se aplicará ligadura dinámica (métodos "redefinibles").
- **¿Por qué la política adoptada por los lenguajes C++ y C# viola el Principio de Abierto-Cerrado?**

Ligadura dinámica C++ y C#

- **C++**

```
class A {
    public:
        virtual void f ();
       ...
}

class B: public A {
    public:
          void f ();
       …
}
```

- **C#**

```
class A{
  public virtual void f()
   {…}
   …
}

class B : A {
  public override void f()
   {…}
   …
}
```

5.- Genericidad y Polimorfismo

- **Estructuras de datos polimorfas**:
  - Pueden contener instancias de una jerarquía de clases
- Puede perderse la información sobre el tipo de los objetos
- Necesitamos mecanismos para averiguar el tipo dinámico de los objetos:
  - **Java**: `instanceof // getClass()`
  - **C++:** `dynamic_cast<Tipo*>(p)`
  - **C#:** `<expresion> is <nombreTipo>`
    `<expresionAConvertir> as <tipoDestino>`
  - **Eiffel:** `a?=b`
- Debemos evitar estructuras condicionales que van en contra de:
  - Principio de Abierto-Cerrado
  - Principio de Elección Única

Intento de asignación Eiffel

- **Sintaxis:**

```
ox:X ; oy:Y
ox ?= oy
```

- **Semántica:**

1) Es legal si **X** es compatible con **Y**

2) Se realiza la asignación en tiempo de ejecución si el tipo dinámico de **oy** es compatible con el tipo estático de **ox**, **X**.

3) **ox** tendrá valor `void` si no se realiza la asignación

Ejemplo de intento de asignación

Encontrar las mayor diagonal de los elementos de una lista

```
max_diagonal (lista_fig: LIST[FIGURA]): REAL is
  local r: RECTANGULO
  do
   from lista_fig.start; Result:= -1.0
   until  lista_fig.after
   loop
       r ?= lista_fig.item;
       if r /= void then Result:= Result.max(r.diagonal)
       end;
       lista_fig.forth
  end
end
```

Intento de asignación en Java

```
public float maxDiagonal (LinkedList<Figura> listaFig) {
   float actual,result=-1;

   for (Figura figura : listaFig){
       if (figura instanceof Rectangulo){
             actual = (Rectangulo)figura.getDiagonal();
             if (actual>result) result = actual;
        }
    }
    return result;
}
```

Intento de asignación en C# (1/2)

```
public float maxDiagonal (LinkedList<Figura> listaFig) {
   float actual,result=-1;

   foreach (Figura figura in listaFig){
       if (figura is Rectangulo){
             actual = (Rectangulo)figura.getDiagonal();
             if (actual>result) result = actual;
       }
    }
    return result;
}
```

Intento de asignación en C# (2/2)

```
public float maxDiagonal (LinkedList<Figura> listaFig) {
   float actual,result=-1;

   foreach (Figura figura in listaFig){
       Rectangulo rec = figura as Rectangulo;
       if (rec != null){
             actual = rec.getDiagonal();
             if (actual>result) result = actual;
       }
    }
    return result;
}
```

C++ dynamic_cast<Tipo*>(p)

```
void VerSueldo(Persona *p) {
   if(Empleado *pEmp = dynamic_cast<Empleado *> (p))
      pEmp->VerSueldo();
   else
      cout << "No tiene salario." << endl;
}
```

- `p` debe ser una entidad polimórfica
  - El tipo de `p` debe ser una clase con algún método virtual
- La conversión se hace entre tipos de la misma jerarquía.
- Si la conversión falla, el puntero obtenido es del tipo T deseado, pero se le asigna cero (puntero nulo).

Genericidad restringida

- Permite ampliar el número de operaciones válidas para las entidades genéricas.
  - Ejemplos:
    - `class ListaOrdenada <T extends Comparable>`
    - `Class Vector <T extends Number>`
- Limitamos el conjunto de clases que pueden instanciar el parámetro genérico.
- El conjunto de clases que se pueden utilizar para instanciar el parámetro viene determinado por la jerarquía de herencia.

Genericidad restringida

- **C++:**
  - No existe
  - El template puede utilizar cualquier métodos sobre las entidades de tipo T
  - El error lo dará en el momento de instanciar el parámetro

- **Eiffel:**
  - `class ListaOrdenada[T -> Comparable]`
  - `class Vector[T->Number]`

- C#:
  - `class Pila<T> where T : IComparable`
  - `class Vector<T> where T : Number`

6.- Clases abstractas o diferidas

- Especifica una **funcionalidad que es común** a un conjunto de subclases.
- Son clases muy generales de las que no se desea crear objetos pero sin declarar variables de ese tipo.
- Toda clase que contenga al menos un método abstracto o diferido (heredado o no) es abstracta o diferida.
- Una clase puede ser abstracta o diferida y no tener definido ningún método que lo sea
- Solamente tienes sentido en un lenguaje con comprobación estática de tipos.

Clases diferidas en Eiffel

```
deferred class Figura feature

   dibujar is deferred end
   rotar(…) is deferred end
   …

end
```

Clases abstractas en Java y C#

```
abstract class Figura {
                ...
        public abstract void dibujar();
                ...
}
```

Clases abstractas en C++

- Una clase con funciones virtuales se puede convertir en abstracta al declarar como **pura** una de sus funciones virtuales:

virtual void f ( ) = 0;

Ejemplo:

```
class Figura {
    public:
        virtual void dibujar ( ) = 0;
        virtual void trasladar (...) = 0;
        ...
}
```