Tecnología de Programación

Martín L. Larrea

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

Relaciones entre clases

Existe una relación especial entre clases que toma varias formas:

Una clase puede ser una extensión de otra clase

Una clase PilaRápida con las mismas operaciones de Pila pero ademas con una operación desapilarDos() que desapila dos elementos.

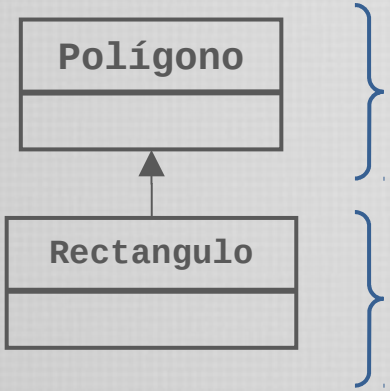
Una clase puede ser una especialización de otra clase

Un cuadrado es un caso especial de polígonos. La clase Cuadrado es una especialización de la clase Poligono.

Una clase puede ser una combinación de otras clases

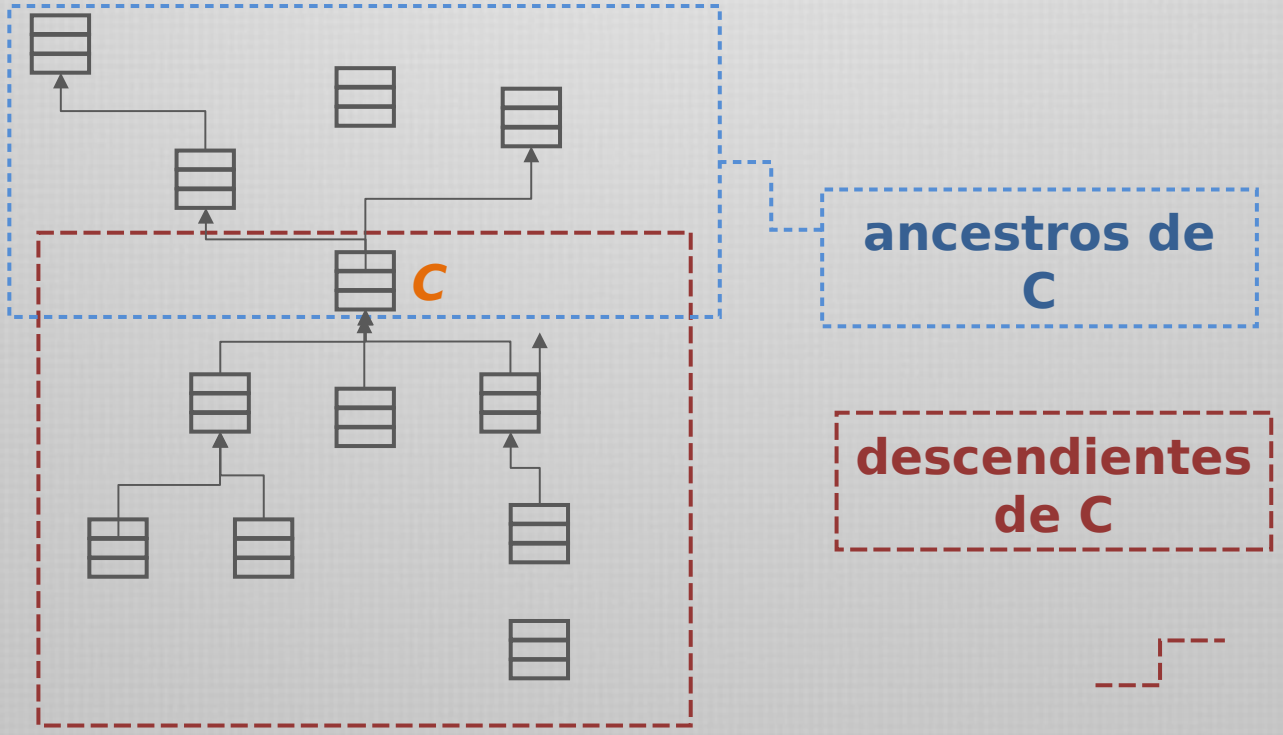
Un ayudante B es un alumno de la Universidad y un docente de la Universidad. Es una combinación de la clases Alumno y Docente

Terminología



clase padre o superclase o clase base

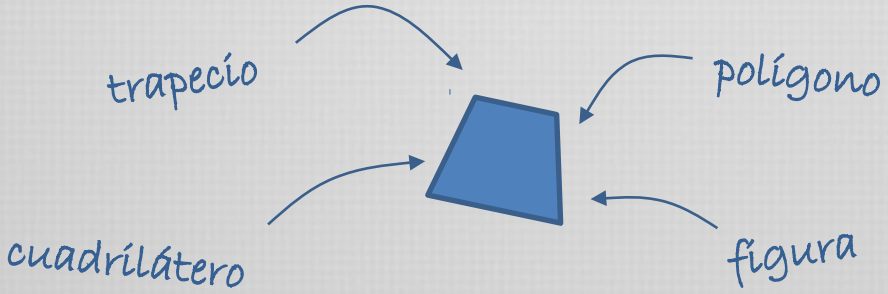
clase hija o subclase o clase derivada



Instancias de una clase

Las instancias de una clase son los objetos
que son instancia de algún descendiente de la clase.
Las instancias propias son las instancias de la misma clase

Consequencia importante:
Un objeto puede ser de varios tipos de datos

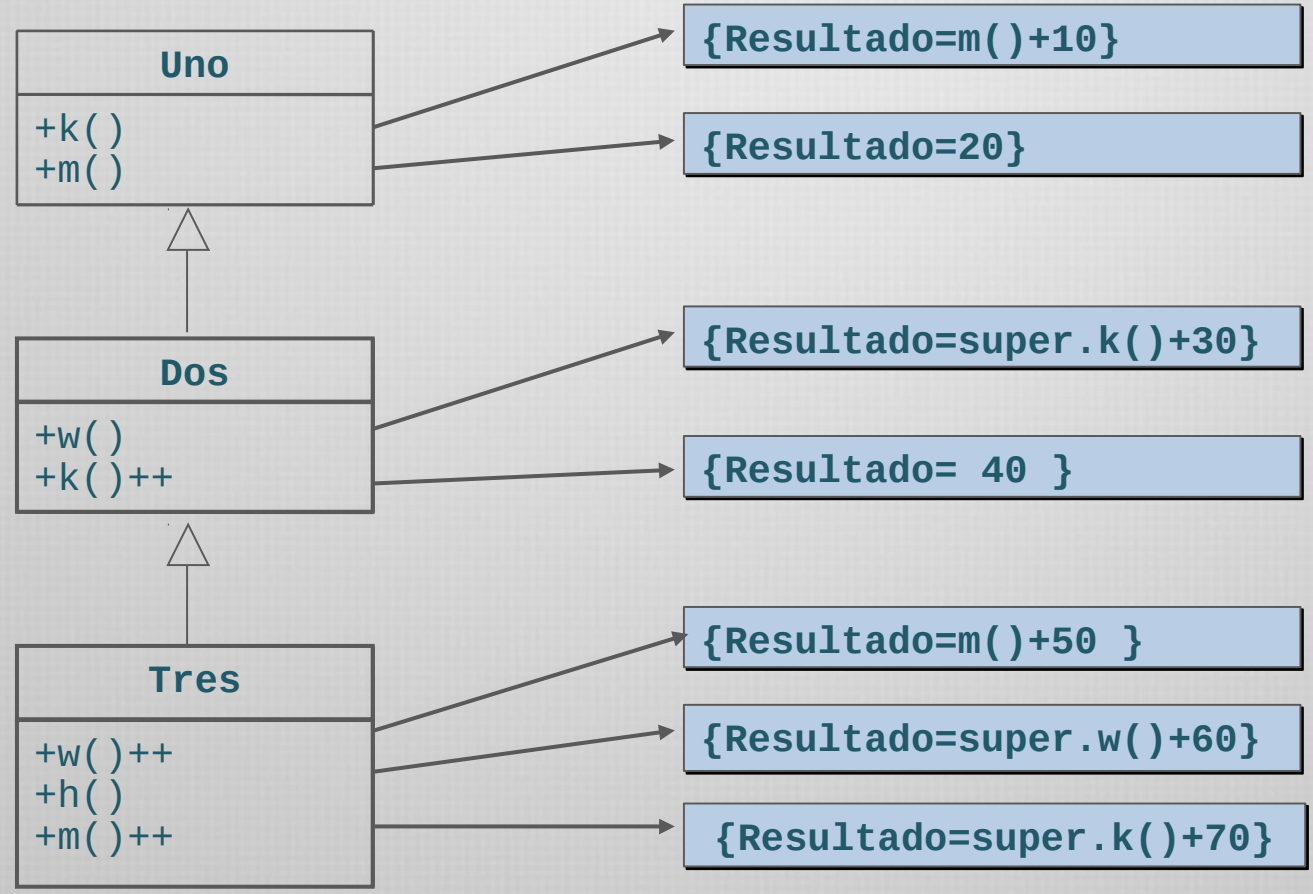


en consecuencia....

Una referencia puede estar asociada a objetos de diferente tipo
(polimorfismo)

Como el tipo de los objetos asociados a una referencia puede variar, también puede hacerlo alguna operación de esos objetos
(vinculación dinámica de código)

Polimorfismo y vinculación dinámica de código




```
p,s:Uno;
q:Dos;
r:Tres;
a,b:entero;
```

```
p ← r
a ← s.k()
s ← p
b ← s.m() + r.h()
b ← b - p.w()
```

Tipos y subtipos

¿Qué es un tipo de datos?

Un conjunto de valores y las operaciones que pueden aplicarse sobre esos valores.

valores + operaciones

números enteros	*suma, resta, multiplicación, división*
true, false	*and, or ,not, xor, xand*
figuras geometricas	*rotar, colorear, expandir, modificar vertices,...*

qué se puede hacer

no cómo se hace

Tipos y subtipos

¿Qué es un tipo de datos?

Un conjunto de valores y las operaciones que pueden aplicarse sobre esos valores.

qué ----> especificación del tipo de datos

(las operaciones que podemos invocar a un objeto)

cómo ----> implementación del tipo de datos.

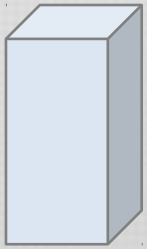
Tipos y subtipos

¿Qué es un tipo de datos?

Un conjunto de valores y las operaciones que pueden aplicarse sobre esos valores.

tipo

enemigo



qué

cómo

caminar
detenerse
emitirSonido
atacar
morir

subtipo

qué

cómo

qué

cómo

Tipos y subtipos

¿Qué es un tipo de datos?

Un conjunto de valores y las operaciones que pueden aplicarse sobre esos valores.

A

B

El tipo B es un subtipo del tipo A cuando la especificación de B incluye la especificación de A

(qué)

(qué)

Es decir, un objeto que satisface la especifiación de B también satisface la de A

Tipos y subtipos

Una noción importante de tipos y subtipos es la substitución de objetos...

Si B es un subtipo de A,
entonces
si el código está preparado para tratar objetos de tipo A,
todo debería seguir funcionando bien cuando se proveen objetos de tipo B.

Esto implica más que simplemente "responder a los mismos pedidos o mensajes"

Los objetos de tipo B deberían respetar el rol de los objetos de tipo A
(volveremos sobre esto más adelante)

Herencia de interfaz y herencia de implementación

La relación entre tipos y subtipos y la separación con la implementación
lleva a dos concepciones diferentes de herencia:
de interfaz y de implementación.

Es una diferencia sutil, pero importante

Para eso es necesario comprender la diferencia entre la *clase* de un objeto y su *tipo*.

- La clase define cómo el objeto es implementado: el estado interno y las implementaciones concretas de sus operaciones.
- El tipo sólo se refiere a la interfaz, es decir, al conjunto de pedidos que ese objeto es capaz de responder.

cómo

qué

Por esta razón,

Un objeto puede tener varios tipos de datos.

Objetos de diferentes clases pueden tener el mismo tipo

Herencia de interfaz y herencia de implementación

Por supuesto, existe una gran cercanía entre las clases y los tipos.

Como una clase define
las operaciones que el objeto puede responder,
también define su tipo.

Una clase es un módulo y un tipo de datos

Un objeto instancia de una clase C, puede responder a los pedidos
declarados en la interfaz definida por la clase C.

C++ y Eiffel utilizan clases para especificar al mismo tiempo la interfaz y la implementación de un objeto.

Java y C# también, pero agregan la idea de interfaz de forma separada, denominada *interface*.

Herencia de interfaz y herencia de implementación

Por esa razón existe una diferencia formal entre herencia de interfaz y herencia de clase o implementación.

La herencia de interfaz describe cuándo un objeto puede ser usado en lugar de otro, porque al fin y al cabo, *responden a los mismos pedidos*.

La herencia de clase define la implementación de un objeto en términos de la implementación de otro objeto. *Desde cierto punto de vista, es un mecanismo para compartir código y representación.*

A veces se confunden porque muchos lenguajes no hacen una distinción explícita.

En C++ y Eiffel y en muchos otros basados en clase, la herencia significa las dos cosas: herencia de interfaz y de clase.

Java, por ejemplo, propone dos formalismos separados para aproximarse a este concepto.

Herencia en Java

En Java existe la herencia de clases...

```
class Ventana extends JFrame {

    public Ventana(){
        ...
    }

...
}
```

JFrame es una clase como cualquier otra.

Define atributos (y por lo tanto el tipo de dato elegido para cada uno) y
provee código para operaciones,
por lo que **Jframe** es realmente
la implementación de un tipo de datos.

La herencia de clase es, en parte, un mecanismo para reutilizar código.

Interfaces en Java

En Java existe también la herencia de interfaces...

```
class Oyente implements MouseListener {
...
}
```

MouseListener no es una clase común.
Es una descripción de operaciones que un objeto debe responder.
¡La interfaz corresponde a la descripción de un tipo de dato!

(qué)

```
public interface MouseListener extends EventListener {
void mouseClicked(MouseEvent e)
void mouseEntered(MouseEvent e)
void mouseExited(MouseEvent e)
void mousePressed(MouseEvent e)
void mouseReleased(MouseEvent e)
}
```

La cláusula **implements** puede considerarse una forma de herencia, en donde el compromiso de la clase “hija” es implementar todas las operaciones declaradas en la clase “padre” (la interfaz).

Interfaces en Java

```
public interface inter1{
    public void f();
    public void g();

}
```

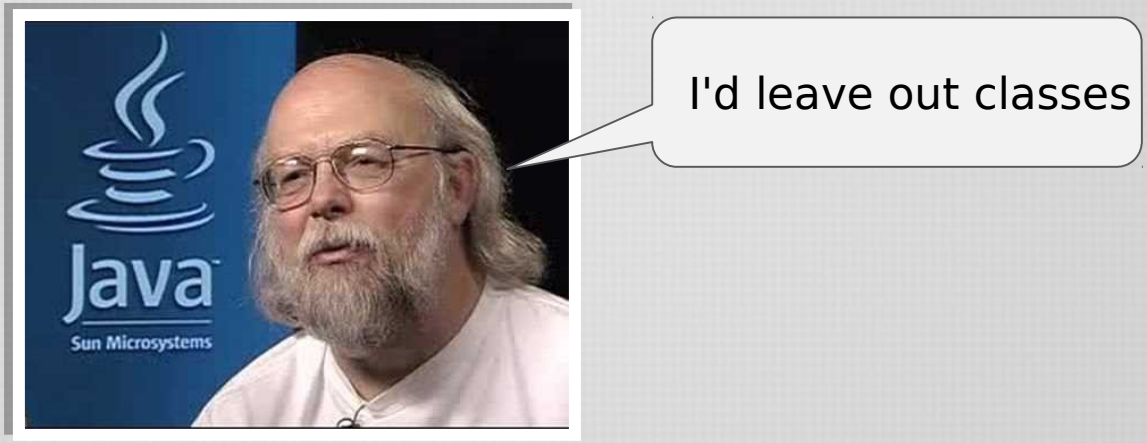
```
class Uno implements inter1 {
  public void f() {
   System.out.println("f, Uno-inter1"); }
  public void g() {
   System.out.println("g, Uno-inter1"); }
}
```

```
class Dos implements inter1 {
  public void f() {
    System.out.println("f, Dos-inter1"); }
  public void g() {
    System.out.println("g, Dos-inter1"); }
}
```

```
…
Uno u = new Uno();
Dos d = new Dos();
inter1 i1;
i1 = u;
i1.f();
i1 = d;
i1.f();
…
```

Interfaces vs. Implementación

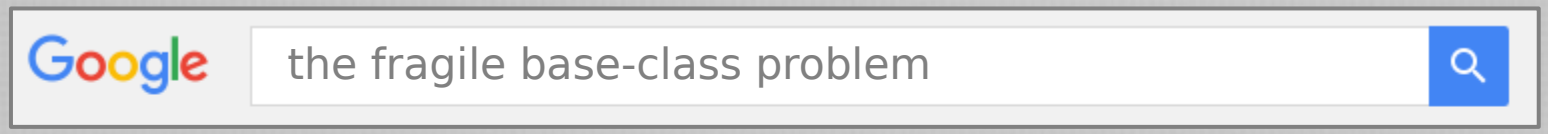
Mr. Gosling, if you could do Java over again, what would you change?



Luego aclaró que no son las clases en sí, sino la herencia de implementación.

Afirma que en muchos casos, la herencia de interfaz es preferible.

Puede proveer mejor flexibilidad.
Provee menor acoplamiento.
Evita "the fragile base-class problem"



Interfaces vs. Implementación

```
f() {
  LinkedList list = new LinkedList();
  //...
  g( list );
}

g( LinkedList list ) {
  list.add( ... );
  g2( list )
}
```

¿Qué pasa si luego descubrimos que conviene usar **HashSet** en lugar de **LinkedList**?

HashSet()

```
f() {
  LinkedList list = new LinkedList();
  //...
  g( list );
}
```

HashSet()

HashSet()

```
g( LinkedList list ) {
  list.add( ... );
  g2( list )
}
```

Interfaces vs. Implementación

```
f() {
  LinkedList list = new LinkedList();
  //...
  g( list );
}

g( LinkedList list ) {
  list.add( ... );
  g2( list )
}
```

¿Qué pasa si luego descubrimos que conviene usar **HashSet** en lugar de **LinkedList**?

Pensar en abstracto ✓

Tratar con abstracciones ✓

```
f() {
  Collection list = new LinkedList();
  //...
  g( list );
}

g( Collection list ) {
  list.add( ... );
  g2( list )
}
```

Clases abstractas

Si bien la herencia de clases se puede ver como una herencia de implementación,
es posible posponer la implementación de una operación en pos de

definir ya su interfaz.

Esto es, la clase declara una operación pero no la implementa.

Esa operación se denomina *abstracta*, y
la clase que la contiene se denomina clase abstracta.

En contrapartida, las clases completamente definidas se denominan clases concretas.

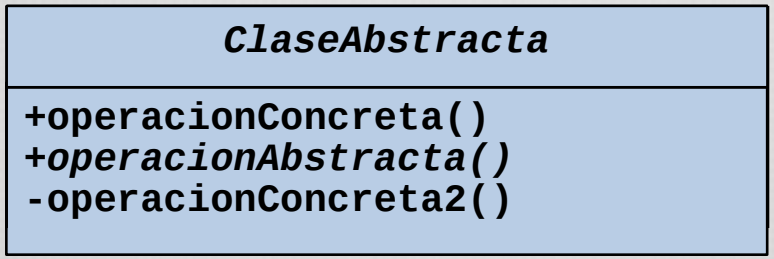
Las clases y operaciones abstractas están presentes en todos los lenguajes,
aunque a veces con terminología diferente:

En C++ las operaciones abstractas se denominan *funciones virtuales puras*.
Una clase abstracta pura es básicamente una interfaz.

En Eiffel, las clases y operaciones abstractas se denominan diferidas.

Clases abstractas

En UML las clases y operaciones abstractas se indican con letras cursivas.



Formalmente en orientación a objetos,
una clase abstracta es una clase que al menos tiene una operación abstracta.

Técnicamente en algunos lenguajes debe decirse explícitamente que la clase es abstracta.

En Java, por ejemplo, si declaramos una operación ***abstract****,*
la clase debe ser ***abstract*** *también.*
Sin embargo, no necesariamente al revés.

Clases abstractas

Las clases abstractas no poseen implementación completa y por lo tanto no pueden producirse objetos, es decir, no pueden instanciarse.

Por lo tanto, naturalmente deben ser heredadas.

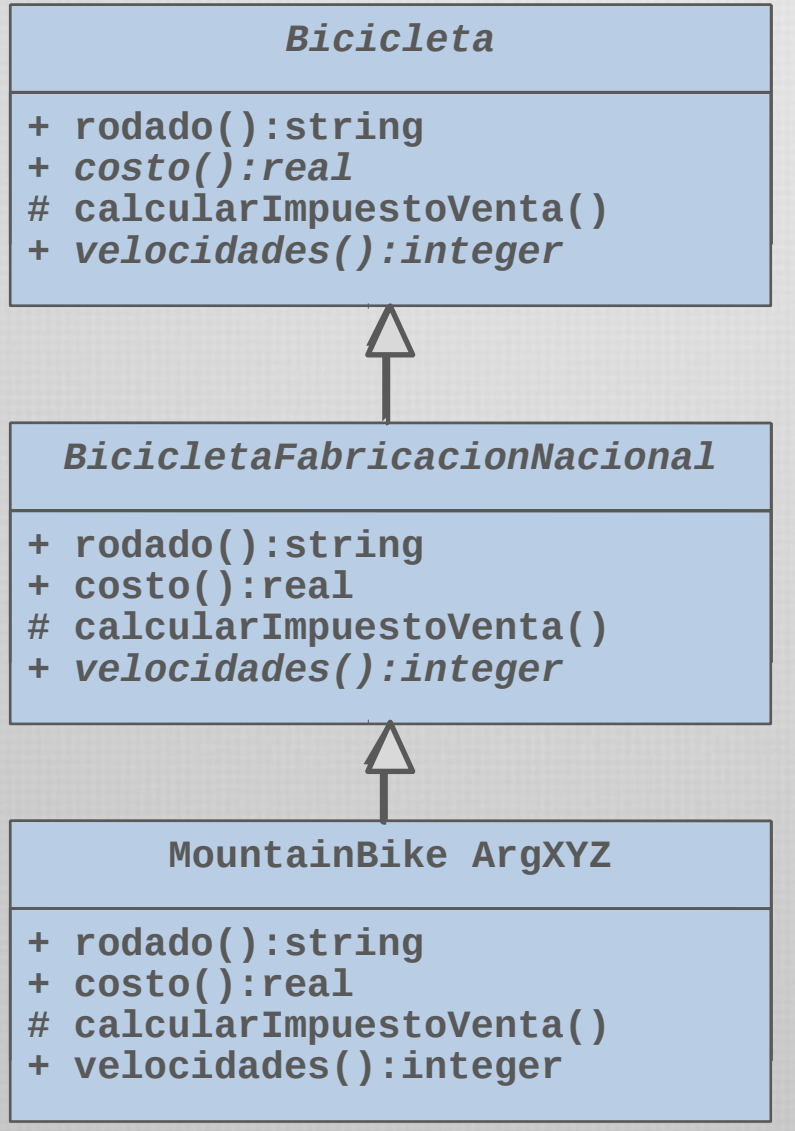
Una clase que hereda de una clase abstracta puede incluir sus propias implementaciones de las operaciones abstractas.

En ese caso se dice que la operación es efectivizada en la clase hija.

Si se efectivizan todas las operaciones, entonces la clase hija es una clase concreta.

Si se deja alguna operación sin efectivizar, permanece abstracta y la clase hija es también una clase abstracta.

Clases abstractas y clases concretas



Una clase abstracta que hereda de otra clase abstracta refina un concepto generalizado previamente.

Sigue siendo una generalización

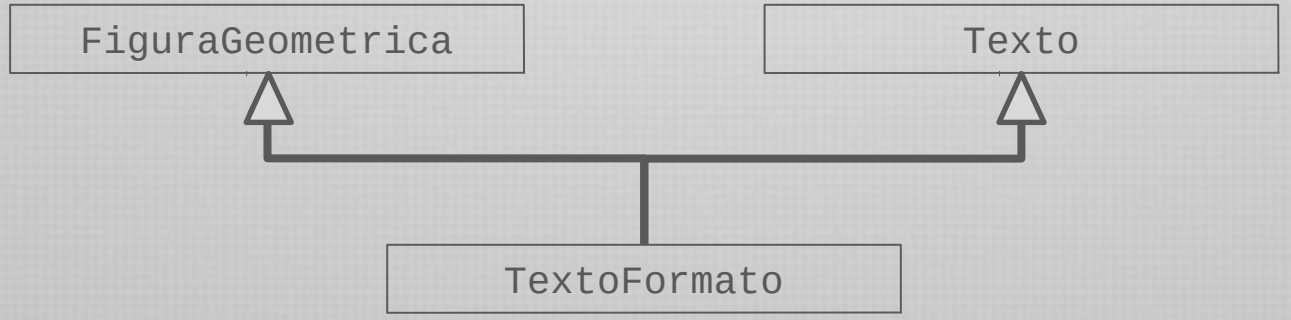
Herencia múltiple

Anteriormente mencionamos una forma de generalización:

Una clase puede ser una combinación de otras clases

Un ayudante B es un alumno de la Universidad y un docente de la Universidad. Es una combinación de la clases Alumno y Docente

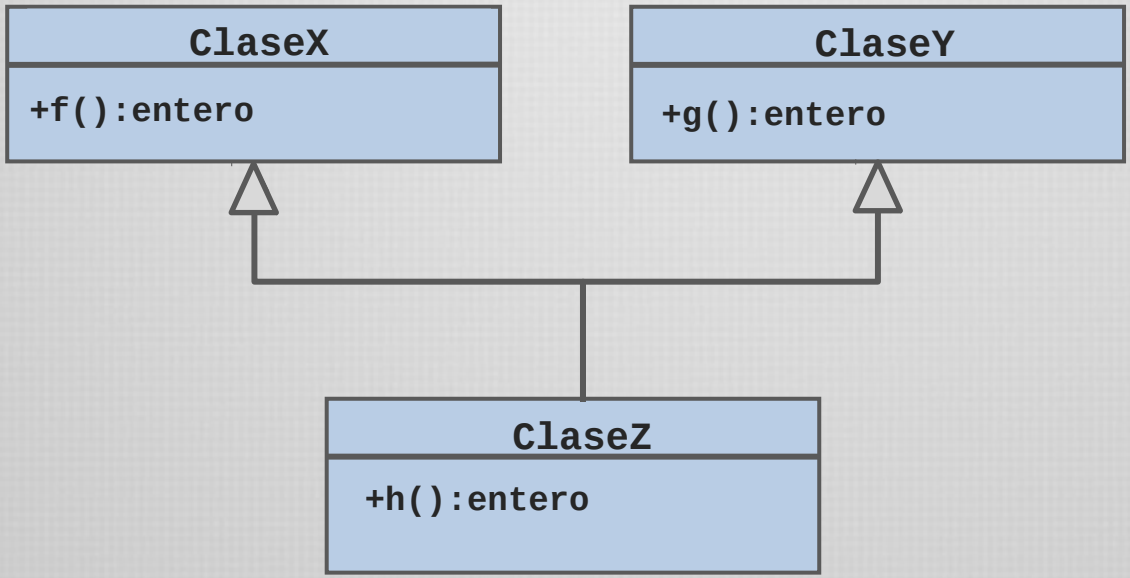
En este caso dos o más clases son generalizaciones parciales de conceptos que admite más de una vista.



La clase hija heredará todos los miembros de las clases padres (siempre de acuerdo a las reglas de visibilidad)

Herencia múltiple

En primera instancia no parece haber nada nuevo en este concepto...

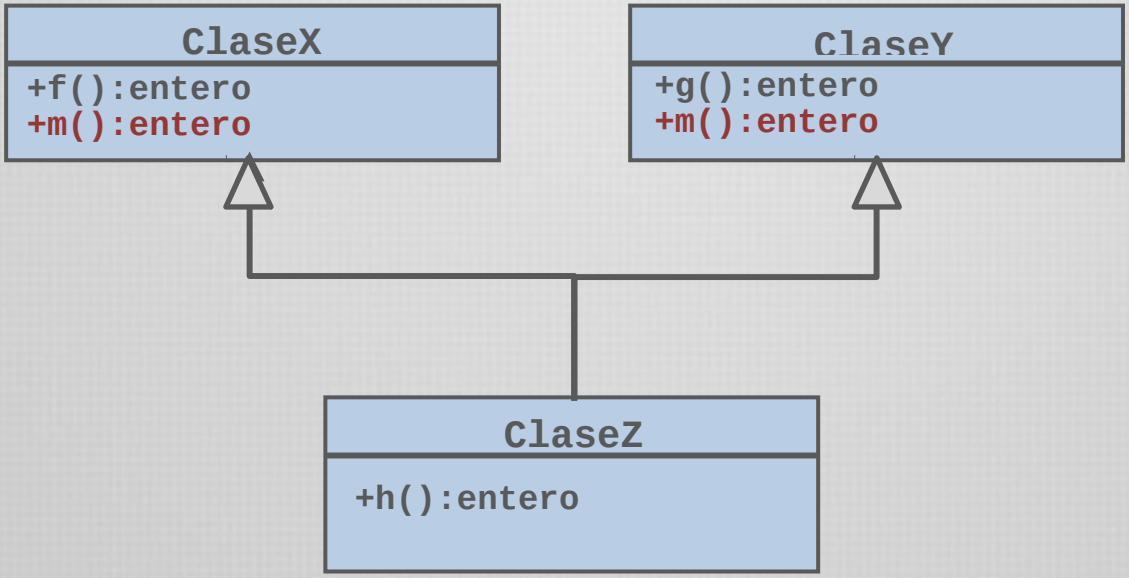


La clase **`ClaseZ`** hereda la función `f()` y la función `g()`y puede usarlas con total libertad, incluso redefinirlas.

La idea de heredar es exactamente la misma que en el caso de herencia simple

Herencia múltiple

Sin embargo, algunas situaciones pueden ser problemáticas...



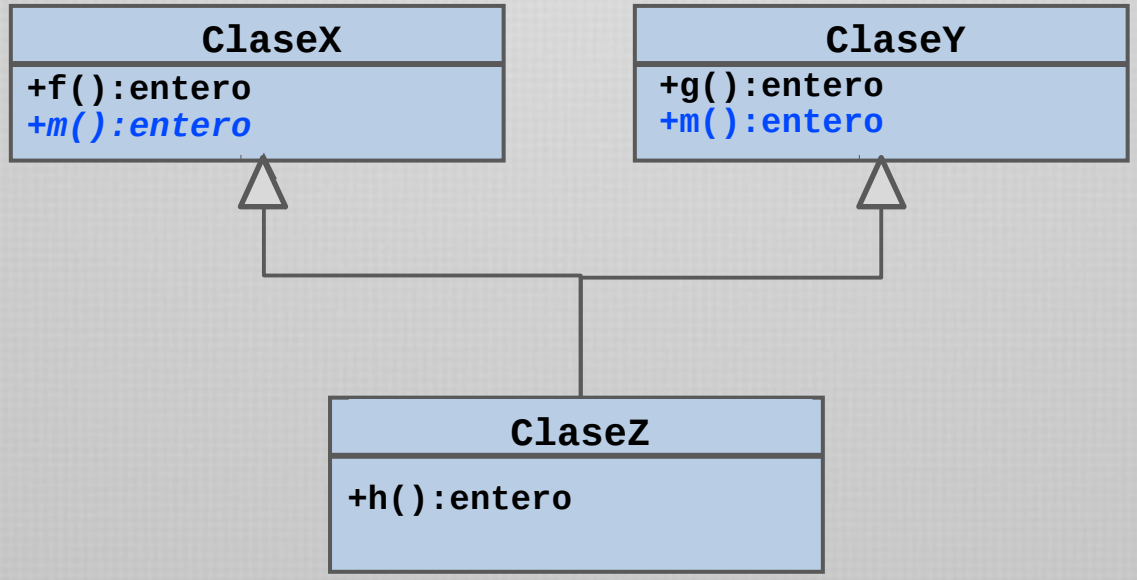
La clase **ClaseZ** hereda la operación **m()** , pero... ¿cuál?

Colisión de nombres

En estas situaciones se dice que existe una ambigüedad de nombres o colisión de nombres.

La colisión se produce cuando los dos servicios son efectivos.

Cuando uno de ellos es diferido (abstracto) no se produce una colisión pues se considera al otro como una efectivización.



En este caso NO hay colisión de nombres, porque una de las operaciones es abstracta

Soluciones a la colisión de nombres

Los lenguajes pueden optar por dos caminos:
proveen algún mecanismo para tratar la ambigüedad, o
proveen algún comportamiento por defecto.

La solución más simple es obviamente prohibir estas situaciones.

Es decir, la herencia múltiple con colisión de nombres es inválida

y

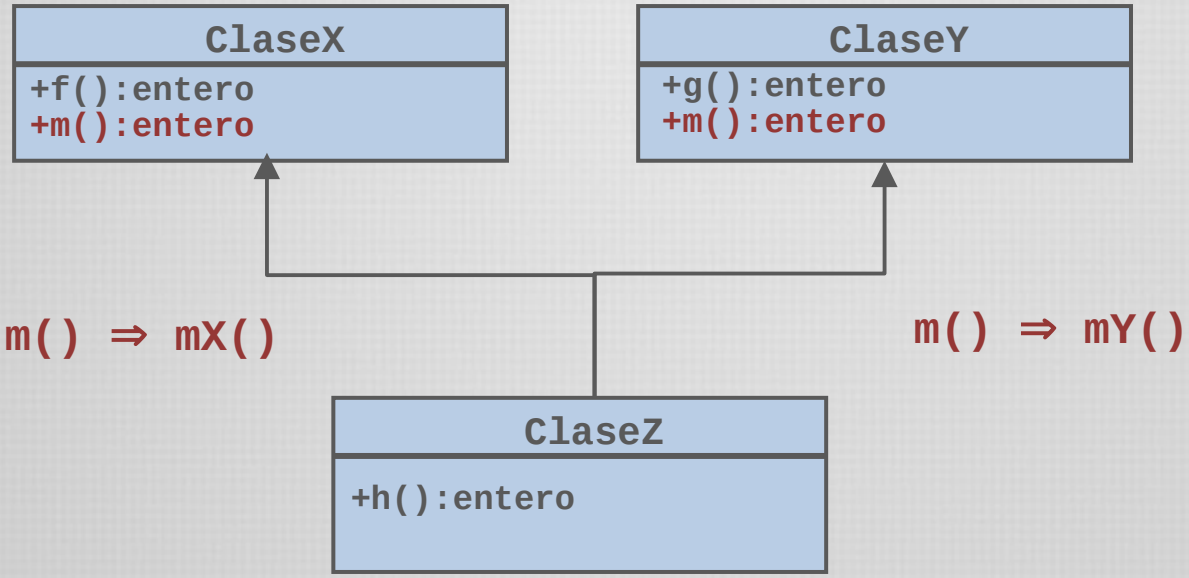
debe solucionarla el desarrollador, diseñando de otra forma

Otra solución es permitir renombrar un atributo u operación en el momento en que es heredado

La idea central es otorgarle un nuevo nombre a alguno de los servicios heredados, de forma tal de eliminar el conflicto existente.

No se modifican otros aspectos de la operación renombrada, tales como tipo del resultado, parámetros, precondiciones o postcondiciones.

Soluciones a la colisión de nombres - renombre



La operación **m()** de la clase **ClaseX** es renombrada a **mx()**

La operación **m()** de la clase **ClaseY** es renombrada a **my()**

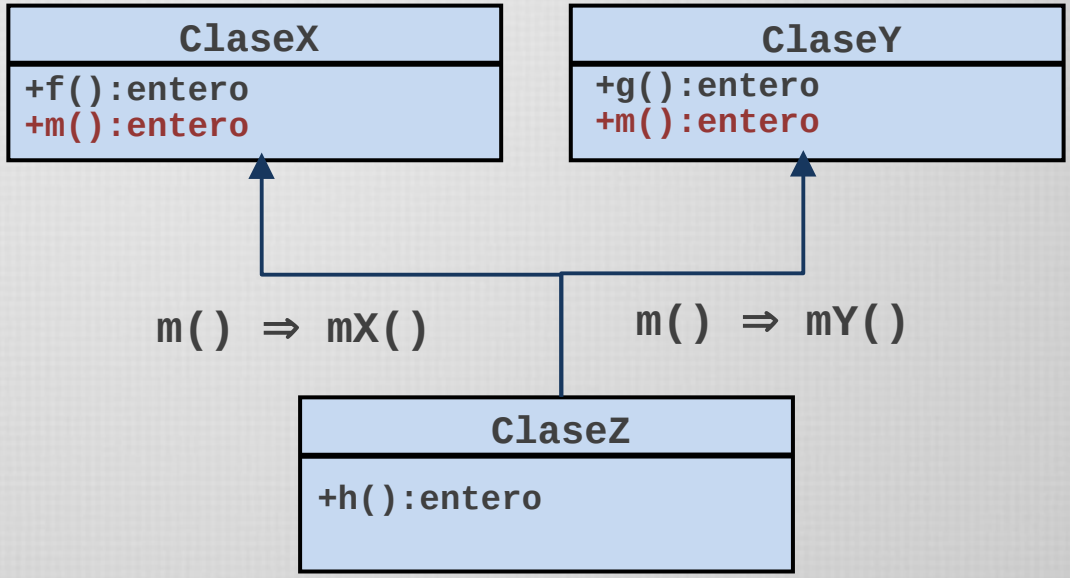
El renombre es un aspecto sintáctico.

Pretende eliminar el conflicto producido al nombrar con el mismo identificador diferentes operaciones.

No todos los lenguajes utilizan esta técnica.

Efectos del renombre de operaciones

```
x:ClaseX
y:ClaseY
z:ClaseZ
```



- Operaciones Válidas

  `x.m() ; y.m() ; z.mX() ; z.mY()`

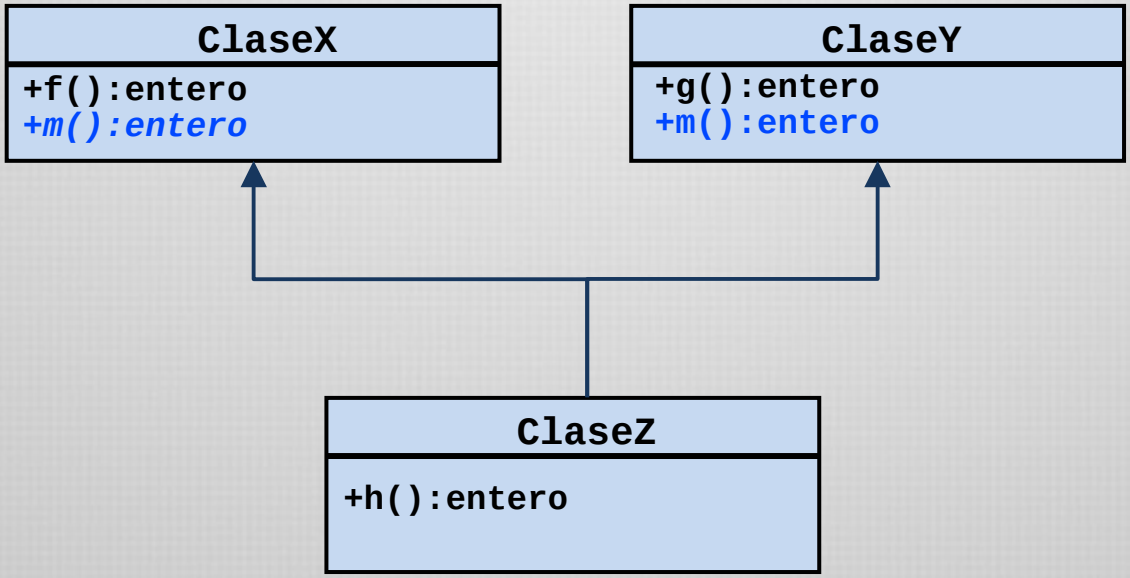
  Despues de **x←z** o de **y←z** tienen el mismo efecto

- Operaciones Inválidas

  `x.mX() ; y.mY() ; x.mY() ; y.mX() ; z.m()`

Soluciones a la colisión de nombres – desambiguar llamadas

Otra solución es mantener dos versiones y explicitar en la llamada cuál de las operaciones se desea invocar.



En C++

```
ClaseX* x;
ClaseZ* z;

...
x->m();
x=z
x->ClaseX::m()
x->ClaseY::m()
```

Soluciones a la colisión de nombres – prioridades

La solución de C++ es en realidad un tratamiento por defecto al problema de la ambigüedad.

Otro tratamiento por defecto es el de establecer prioridades.

Un lenguaje puede establecer una prioridad dada por **el orden** en que son enumeradas las clases.

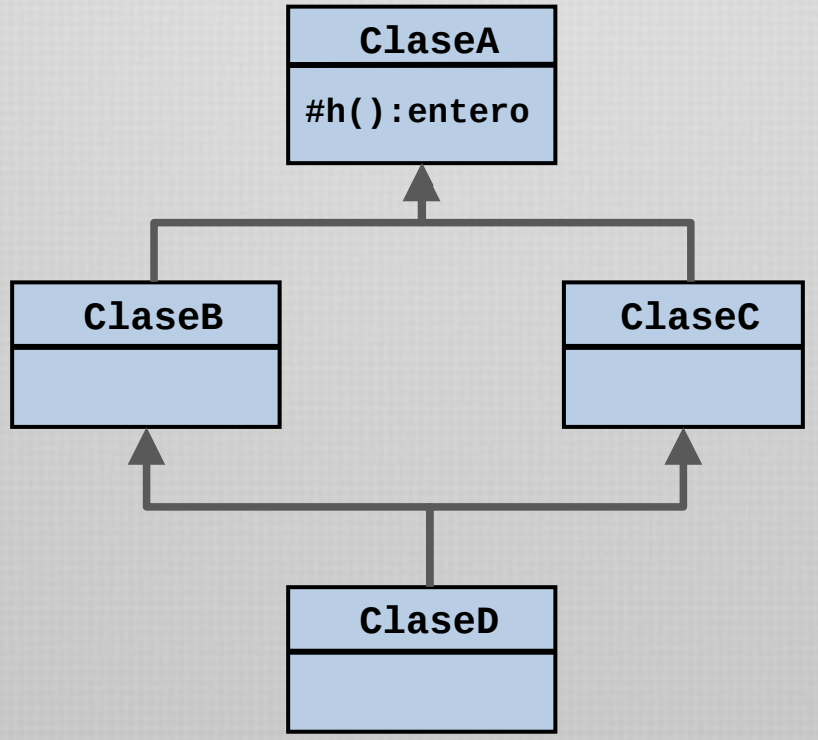
Si una clase C hereda de las clase A y B “en ese orden”,
entonces
si existe ambigüedad o colisión de nombres,
se tomará la primera versión encontrada (en este caso A).

Esto es implementado así en el lenguaje Python...

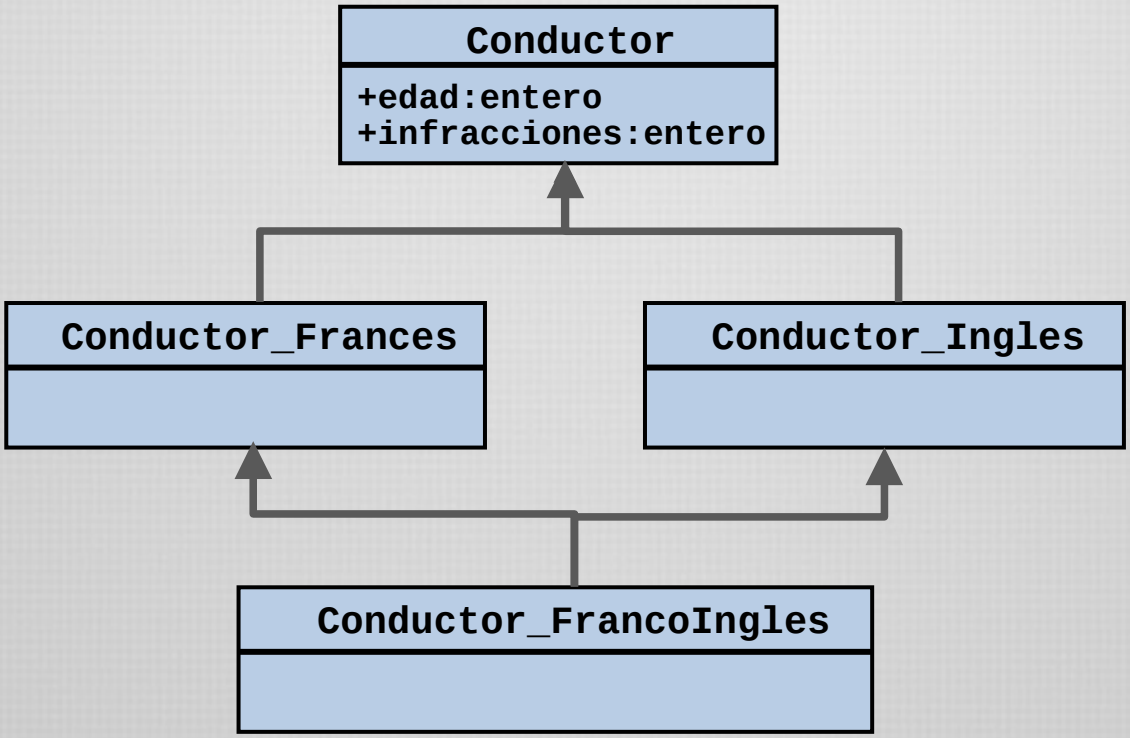
Herencia repetida

Otro de los problemas que acarrea la herencia múltiple es la denominada herencia repetida, también conocido como el *problema del diamante*.

La herencia repetida surge cuando una clase es descendiente de otra,
en más de una forma.



Herencia repetida



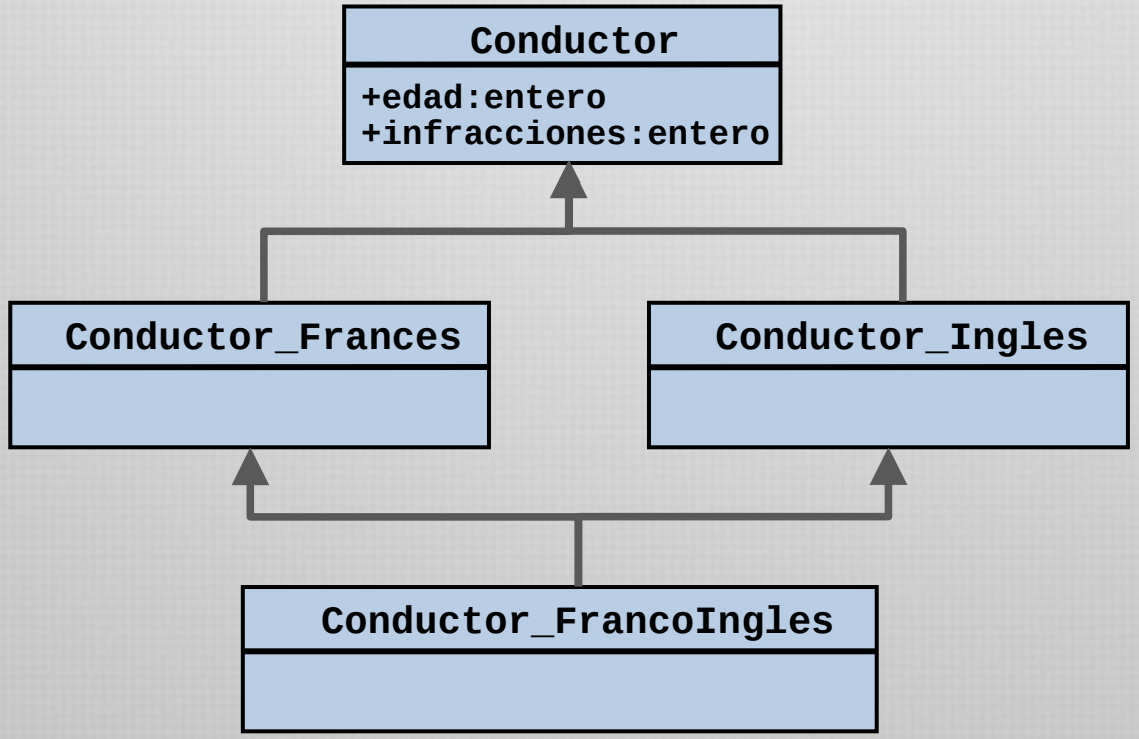
En el caso puntual del atributo **edad** no existe controversia, dado que representa la *edad* del conductor, y por lo tanto, tiene el mismo significado durante toda la cadena de *herederos*.

Si bien **Conductor_FrancoIngles** hereda dos atributos ***edad***, los resume en uno solo, porque el significado es el mismo.

Herencia repetida

En este caso, el resultado de la herencia es una sola *copia* del atributo u operación en la clase descendiente. Esto se denomina *compartir (sharing)*.

Sin embargo, a veces la entidad heredada tiene un significado diferente por cada camino posible...



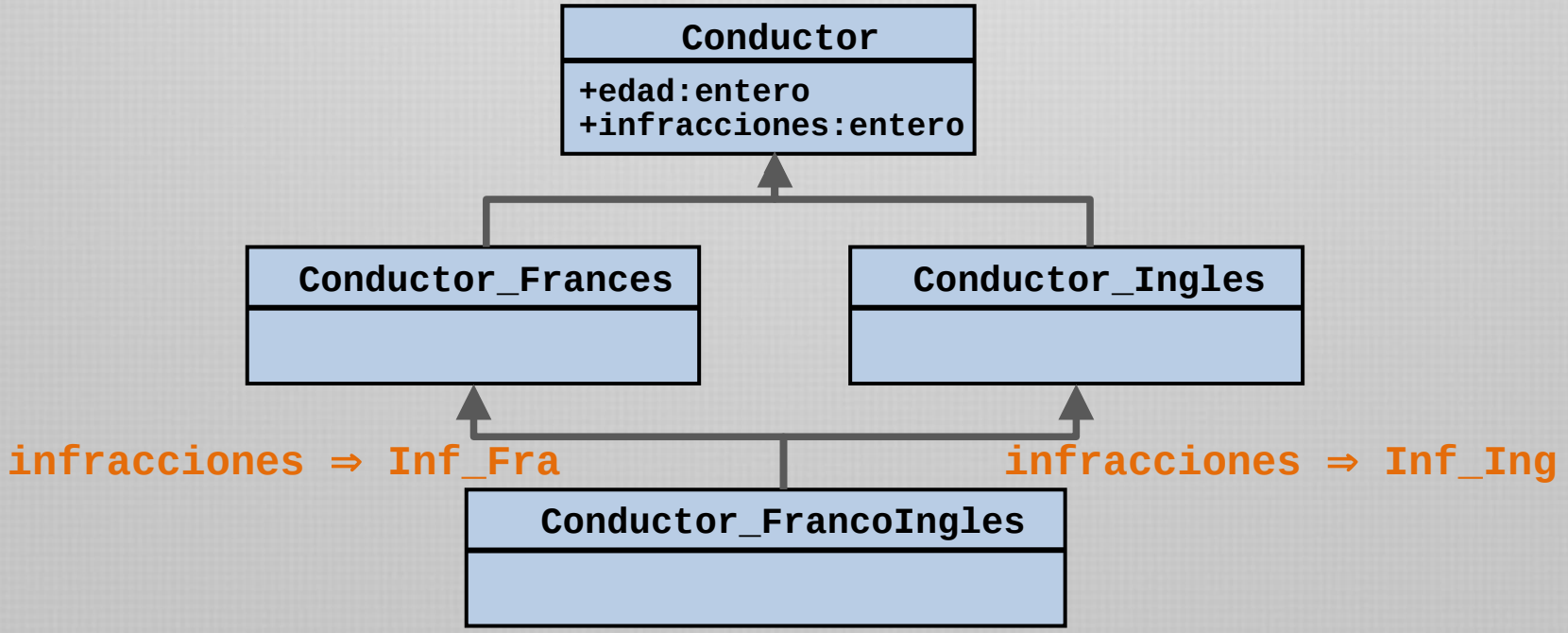
Las infracciones pueden tener un significado diferente, dado que corresponde a lo que hace el conductor en Francia o en Inglaterra.

Herencia repetida

Necesitamos la habilidad de mantener dos copias separadas del mismo atributo heredado de formas diferentes...

Cuando un atributo u operación se hereda en forma repetida bajo diferentes nombres, el resultado de la herencia es la *replicación* del atributo u operación en la clase descendiente.

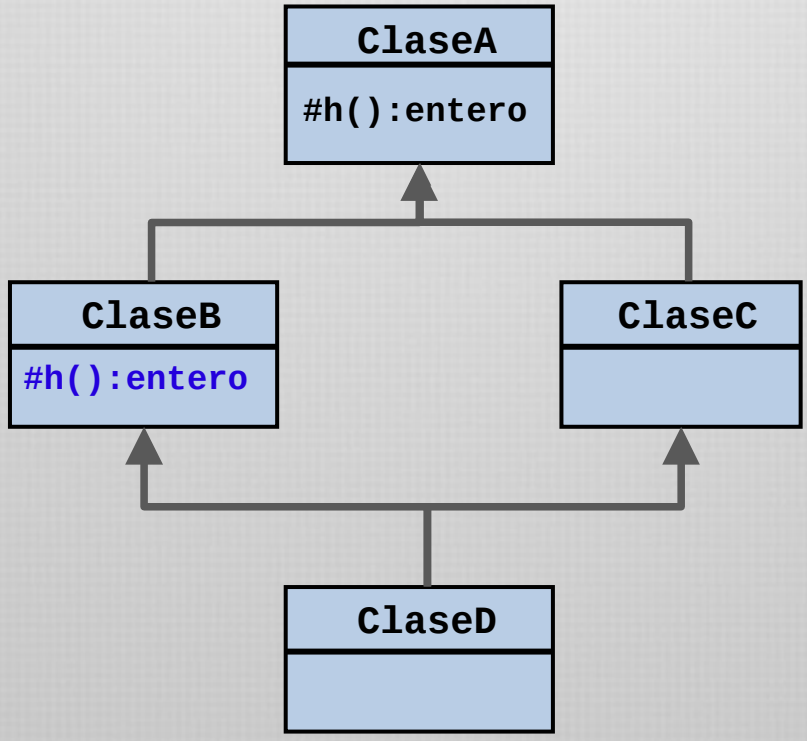
En este caso se puede hacer un renombre de operaciones, para que sean heredadas bajo nombres diferentes.



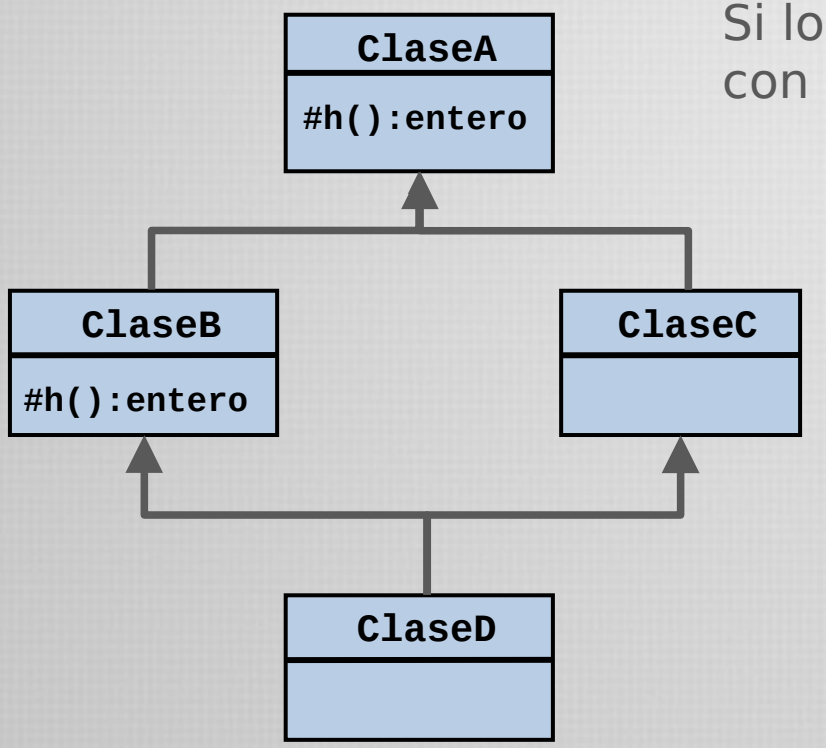
Herencia repetida

Pueden darse aún ciertos casos problemáticos...

¿Qué sucede cuando se redefine una operación en alguno de los caminos de herencia?



Herencia repetida



Si los atributos o servicios son heredados con el mismo nombre:

[1] Si una de las versiones es concreta, y las demás abstractas, entonces la versión concreta efectiviza a las demás.

[2] Si ambas versiones son concretas, pero el servicio es redefinido en la última clase, entonces todas las versiones son unificadas en la nueva versión.

[3] Si hay mas de una versión concreta, pero no hay redefinición, entonces ocurre una colisión de nombres.

Si los atributos o servicios son heredados con distintos nombres
entonces hay *replicación*.

¿que problema pueden traer
relacionado con lo que vimos hoy?

¿cómo hacer que una clase produzca una sola instancia en el sistema?