

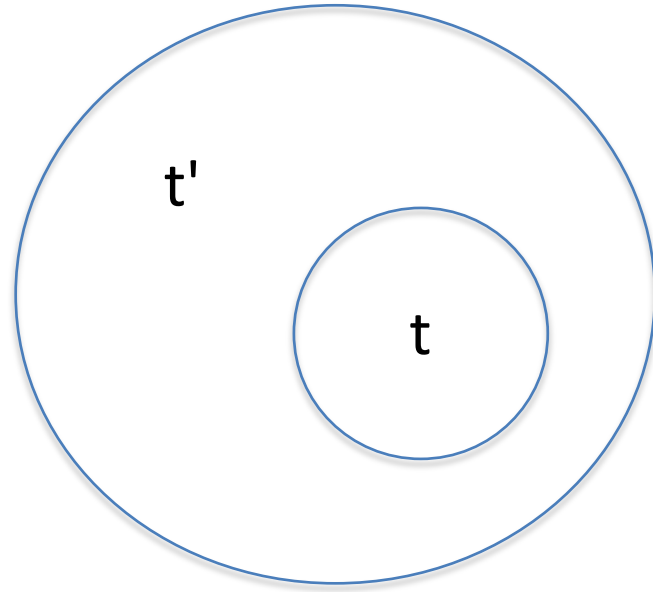


Subtipos I

Fernando Orejas

Introducción

¿Qué es un subtipo?



¿Qué es un subtipo?

Definición 1:

t es subtipo de t' si todos los elementos de t son elementos de t'. Por ejemplo, en Pearl:

```
subset Pares of Int where {$_ % 2 == 0}
```

Esta clase de subtipos no son habituales en los lenguajes de programación

¿Qué es un subtipo?

Definición 2:

t es subtipo de t' si cualquier función que se pueda aplicar a un objeto de tipo t' , se puede aplicar a un objeto de tipo t .

Esta es la definición en que se basa la programación orientada a objetos.

¿Qué es un subtipo?

Definición 2':

t es subtipo de t' si en todo contexto en que se puede usar un objeto de tipo t' , se puede usar un objeto de tipo t .

Esta es una definición en que (a veces se dice que) se basa la programación orientada a objetos

Ejemplo:

```
class Punto{  
    private:  
        double x,y;  
    public:  
        ...  
        void mover (double a, double b){  
            x = x + a; y = y + b;  
        }  
        ...  
}
```


Ejemplo:

```
class Punto_color: public Punto {  
    private:  
        int color;  
    public:  
        ...  
}  
  
...  
Punto_color p;  
  
...  
p.mover(2.1, 3.4)  
  
// Válido porque Puntocolor es subtipo  
// (subclase) de Punto
```

¿Qué es un subtipo?

Las definiciones 1 y 2 no son equivalentes.

- Si t es subtipo de t' de acuerdo con la Def. 1, entonces también lo es de acuerdo con la Def. 2.
- La inversa no es cierta, en general. Es decir, si t es subtipo de t' de acuerdo con la Def. 2, entonces no tiene por que serlo de acuerdo con la Def. 1.

Ejemplo:

```
class C1{
```

```
    T x;
```

```
    ...
```

```
}
```

```
class C2: public C1 {
```

```
    T y; // C2 tiene como atributos x e y
```

```
}
```

C2 no puede ser visto como un subconjunto de C1, porque tiene más elementos.

¿Qué es un subtipo?

Definición 3:

t es subtipo de t' si los objetos de t se pueden convertir implícitamente a objetos de t' (type cast o coerción).

Herencia y subclases

La herencia y la relación de subclase permiten:

- Estructurar mejor el código
- Tener una mayor reutilización de código
- Simplificar el diseño

Ejemplo:

```
class Empleado{
```

```
...
```

```
}
```

```
...
```

```
double sueldo (Empleado e);
```

```
...
```

```
Empleado e;
```

```
double s;
```

```
s = sueldo(e);
```

Ejemplo (programación "clásica"):

```
double sueldo (Empleado e);  
    if (e.es_vendedor()){  
        ...}  
    else if (e.es_contable()){  
        ...}  
    else if (e.es_ejecutivo()){  
        ...}  
    ...  
}  
}
```

Ejemplo (programación OO):

```
class Empleado{  
    double sueldo ();  
    ...  
}
```

```
class Vendedor: public Empleado {  
    double sueldo ();  
    ...  
}
```

```
class Contable: public Empleado {  
    double sueldo ();  
    ...  
}
```


Herencia y subclases

- En cada subclase podemos redefinir operaciones de la clase base
- Si `e` es un empleado del tipo que sea, `e.sueldo()` calcularía el sueldo que corresponde a su tipo, ya que el tipo de `e` se decide en ejecución (**No exactamente en C++**)

Herencia y subclases

Si cambia la estructura salarial: por ejemplo, hay algún nuevo tipo de empleado, desaparece algún tipo y a otros tipos se les cambia el cálculo del sueldo:

- En la programación clásica hay que rehacer entera la función de sueldo (y quizá lo mismo para otras operaciones)
- En el caso OO, se introducirían nuevas subclases, se eliminarían algunas y se redefiniría el cálculo del sueldo en otras

Lenguaje de programación OO

=

Modularidad (abstracción de datos)

+

Herencia

+

Vinculación dinámica o tardía (Late binding)



Subtipos II

Fernando Orejas

***Comprobación e inferencia de tipos con
subtipos***

La estructura de tipos

- Si $e :: t$ y $t \leq t'$ entonces $e :: t'$
- Si $e :: t$, $t \leq t'$ y $f :: t' \rightarrow t''$ entonces $f e :: t''$

La estructura de tipos

- Si $e :: t$ y $t \leq t'$ entonces $e :: t'$
- Si $e :: t$, $t \leq t'$ y $f :: t' \rightarrow t''$ entonces $f e :: t''$

Consecuencia

- Si $e :: t$, $t \leq t'$ y $f :: t' \rightarrow t'$ entonces $f e :: t'$

NO podemos asegurar que $f e :: t$

La estructura de tipos

Por ejemplo, si $x: \text{par}$, $y: \text{par} \leq \text{int}$ y

`bool es_positivo (int n);`

`int incr_1 (int n);`

entonces:

- `es_positivo(x) :: bool`
- `incr_1(x) :: int` pero no `incr_1(x) :: par`
- Por ejemplo, si $x == 6$, `incr_1(x)` no sería par.

La estructura de tipos

- En el caso de la asignación, si $x :: t$, $e :: t'$ y $t' \leq t$ entonces:

$x = e$ ¿es correcto?

La estructura de tipos

- En el caso de la asignación, si $x :: t$, $e :: t'$ y $t' \leq t$ entonces:

$x = e;$ ¿es correcto?

SI

Si $x :: \text{int}$ y $z :: \text{par}$, es correcto:

$x = z;$

La estructura de tipos

- En el caso de la asignación, si $x :: t$, $e :: t'$ y $t \leq t'$ entonces:

$x = e;$ ¿es correcto?

La estructura de tipos

- En el caso de la asignación, si $x :: t$, $e :: t'$ y $t \leq t'$ entonces:

$x = e;$ ¿es correcto?

¡¡ NO !!

Si $x :: \text{par}$ no es correcto:

$x = 7;$

Covarianza y contravarianza

- Si $s \leq t$ y $s' \leq t'$ ¿sería correcto decir $(s \rightarrow s') \leq (t \rightarrow t')$?

Covarianza y contravarianza

- Si $s \leq t$ y $s' \leq t'$ ¿sería correcto decir $(s \rightarrow s') \leq (t \rightarrow t')$?

¡¡NO!!

Supongamos que $f :: \text{par} \rightarrow \text{par}$ y $g :: \text{int} \rightarrow \text{int}$. Se supone que si $(s \rightarrow s') \leq (t \rightarrow t')$, siempre que podamos usar g , podemos usar f en su lugar. Sin embargo, si tenemos la expresión $g\ 5$, e intentamos substituir g por f , tendríamos $f\ 5$, pero no podemos aplicar f a 5 , porque el argumento de f ha de ser par y 5 no lo es.

Covarianza y contravarianza

En cambio, si $t \leq s$ y $s' \leq t'$ entonces sí podemos asegurar que $(s \rightarrow s') \leq (t \rightarrow t')$

Covarianza y contravarianza

Si $s \leq t$ ¿podemos asegurar que $\text{List } \langle s \rangle \leq \text{List } \langle t \rangle$

Covarianza y contravarianza

Si $s \leq t$ ¿podemos asegurar que $\text{List} <s> \leq \text{List} <t>$

¡¡NO!!

```
void push (List& <Empleado> L, Empleado e){  
    L.insert(L.end(), e)  
}  
...  
List <contable> L; Vendedor e;  
...  
push(L,e) // ¿es correcto esto?
```

Covarianza y contravarianza

- Un constructor de tipos C , C es covariante si $s \leq t$ implica que $C<s> \leq C<t>$.
- C es contravariante si $s \leq t$ implica que $C<t> \leq C<s>$.
- C es invariante si $s \leq t$ no implica $C<s> \leq C<t>$ ni lo contrario.
- El constructor \rightarrow es contravariante en el primer argumento y covariante en el segundo.
- ¿El constructor `List` es covariante, contravariante o invariante?

Otros problemas

Sabemos que si `p::Punto_color`, `p.mover(1.2, 3.4)` es correcto ya que `mover:: Punto -> double -> double -> Punto`, pero entonces:

`p.mover(1.2, 3.4):: Punto.`

¿Quiere esto decir que, al mover el punto, perdemos el campo del color?

Otros problemas

Si tenemos:

```
class C{  
    T f (C x);    ...  
}  
  
class C1: public C {  
    T f (C1 x);    ...  
}  
  
C x; C1 y;
```

¿Es correcto?:

y.f(x);

```
class Punto{
```

```
...
```

```
bool f (Punto p){
```

```
    return x == p.x and y == p.y;
```

```
}
```

```
}
```

```
class Punto_color: public Punto {
```

```
...
```

```
bool f (Punto p){
```

```
    return (x==p.x) and (y==p.y) and (color== p.color);
```

```
}
```

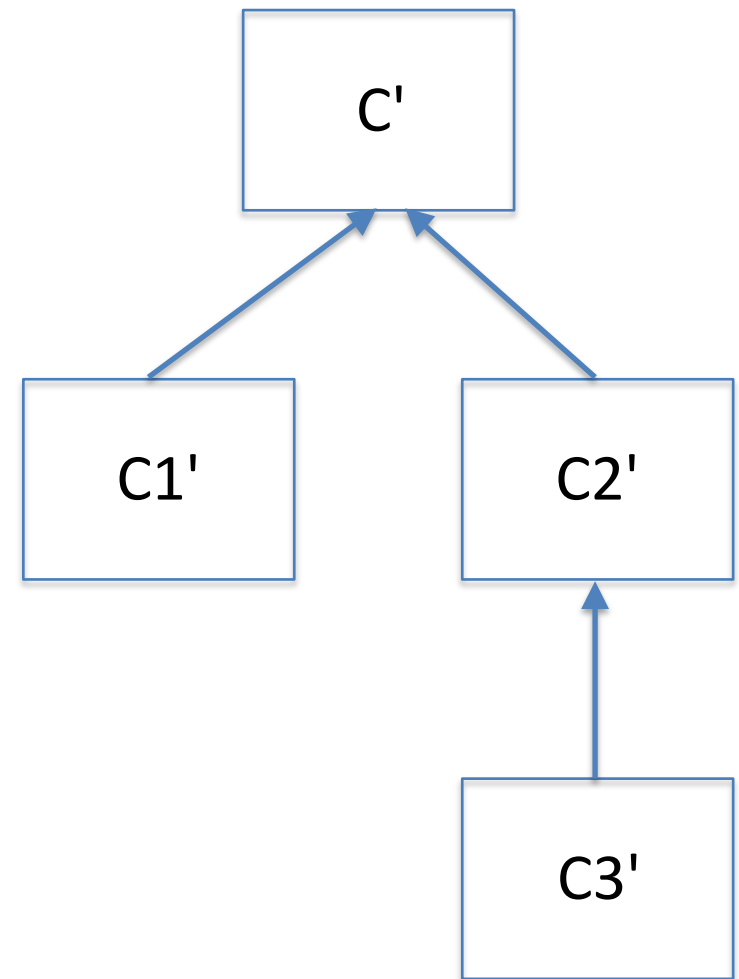
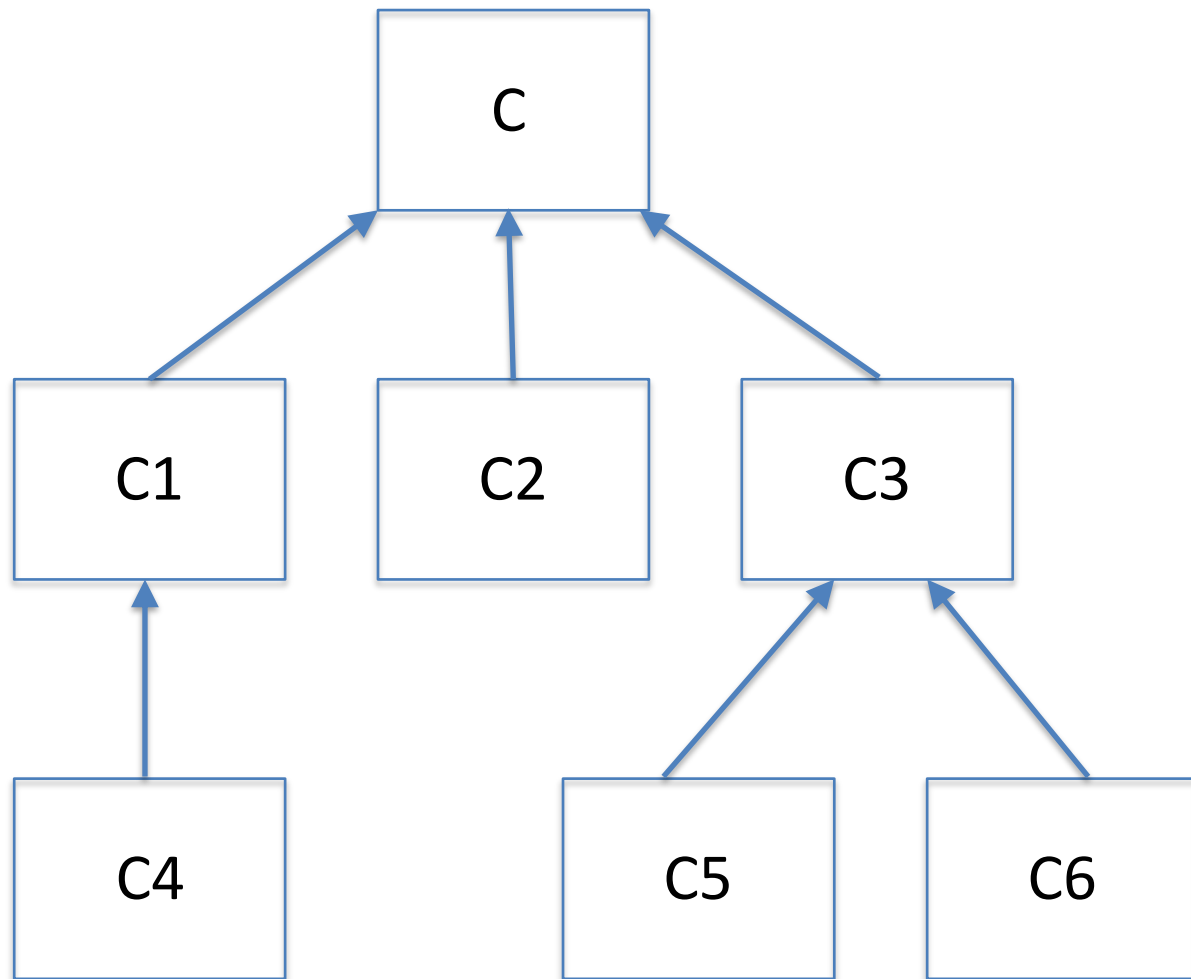
```
}
```

Si $p :: \text{Punto}$ y $p1 :: \text{Punto_color}$ ¿Es correcto?: **$p1.f(p);$**

Subclases y Herencia en Python, C++ y Java

Herencia simple y herencia múltiple

- Un lenguaje de programación tiene solo herencia simple cuando una clase solo puede ser subclase de otra clase
- Un lenguaje de programación tiene herencia múltiple cuando una clase solo puede ser subclase de varias clases



Cuadrilatero

Poligono regular

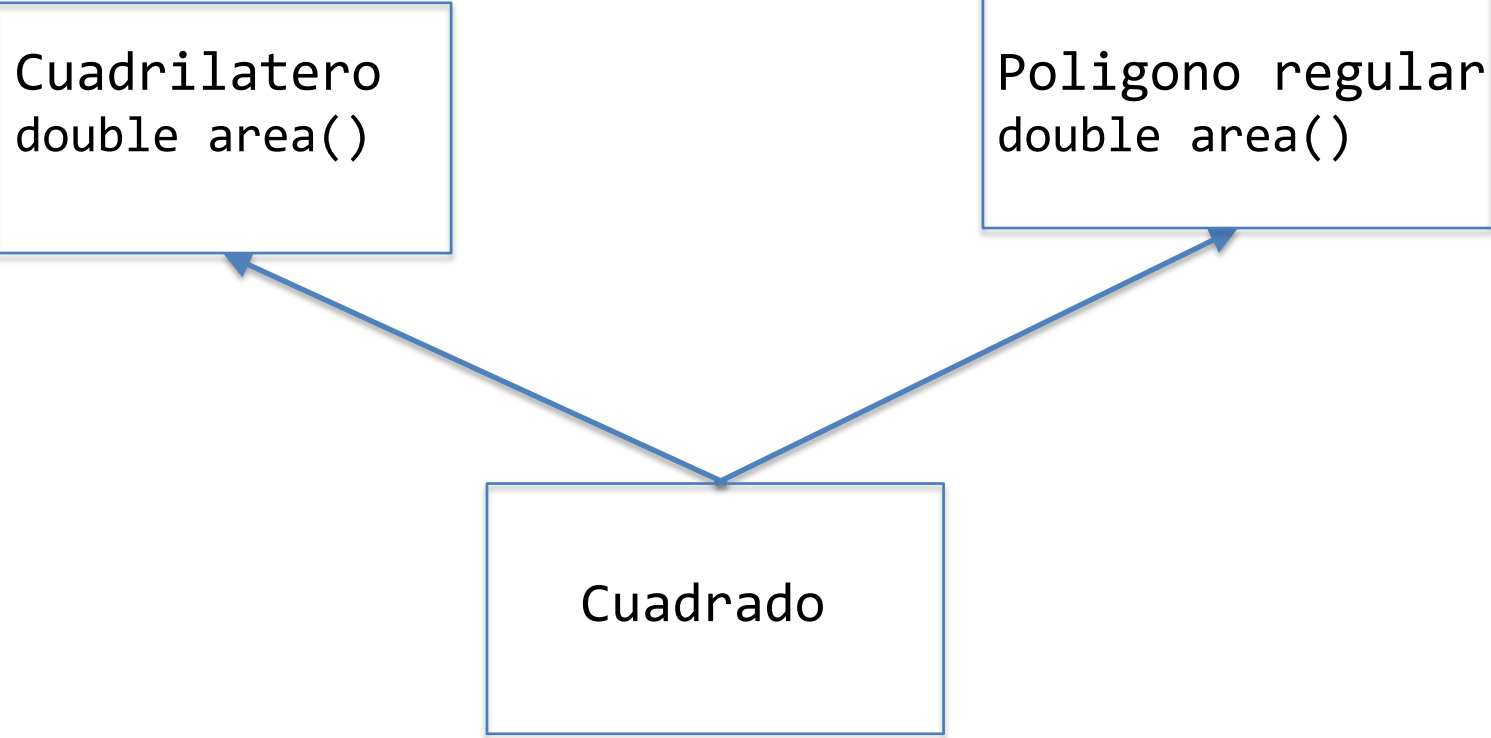
Cuadrado



Cuadrilatero
double area()

Poligono regular
double area()

Cuadrado



Declaración de subclases en C++

```
class Empleado{...}
```

```
class Vendedor: Empleado {...}
```

Pero también

```
class Vendedor: public Empleado {...}
```

```
class Vendedor: private Empleado {...}
```

```
class Vendedor: protected Empleado {...}
```

La diferencia afecta a la visibilidad

Declaración de subclases en C++

Además en C++ tenemos herencia múltiple:

```
class Cuadrado: public Cuadrilatero,  
                public Poligono_regular
```

Declaración de subclases en C++

Además en C++ tenemos herencia múltiple:

```
class Cuadrado: Cuadrilatero,  
                Poligono_regular
```

Resolución de conflictos cuando en las dos clases hay métodos con el mismo nombre y tipo:

```
C.Cuadrilatero::area()
```

```
C.Poligono_regular::area()
```

Declaración de subclases en Java

```
class Empleado{...}
```

```
class Vendedor extends Empleado {...}
```

Declaración de subclases en Java

En Java no hay herencia múltiple con clases, pero sí con interfaces

```
Interface Empleado {...}
```

```
Interface Comisionista {...}
```

```
class Vendedor: implements Empleado,  
Comisionista {...}
```

Declaración de subclases en Python

```
class Empleado:
```

```
...
```

```
class Vendedor (Empleado)
```

```
...
```


Declaración de subclases en Python

En Python hay herencia múltiple

```
class Cuadrilatero: ...
```

```
class Poligono_regular: ...
```

```
class Cuadrado (Cuadrilatero,  
                Poligono_regular)
```

```
...
```

Cuando en las dos clases hay métodos con el mismo nombre y tipo se hereda el de la primera clase

Tipos en Python

- En Python todo objeto tiene un tipo (más concreto) dinámico

```
e = Empleado()
```

```
v = Vendedor()
```

e tiene tipo (dinámico) **Empleado** y **v** tiene tipo **Vendedor**

Tipos en Java

- En Java los objetos tienen un tipo estático y otro dinámico.

```
Empleado e;
```

```
e = new Vendedor(...)
```

e tiene tipo estático **Empleado** y tipo dinámico **Vendedor**

Tipos en C++

- En C++ los objetos estáticos tienen solo tipo estático.
- En cambio, los objetos dinámicos tienen tipo estático y dinámico.

```
Empleado e; Vendedor v;
```

```
Empleado *e1;
```

```
e = v;
```

```
e1 = new Vendedor();
```

e tiene tipo estático **Empleado** mientras que **e1** tiene tipo estático **Empleado** y tipo dinámico **Vendedor**

Vinculación en Python

- En Python la vinculación es dinámica, dados:

```
class Empleado {  
    def sueldo(self)  
        ...  
  
class Vendedor (Empleado)  
    def sueldo(self)  
        ...
```

La llamada `e.sueldo()` ejecutaría la operación sueldo dependiendo del tipo dinámico de `e`.

Vinculación en Java

- En Java la vinculación es en parte estática y en parte dinámica, dados:

```
class Empleado {  
    public double sueldo() ... }  
class Vendedor extends Empleado {  
    public double sueldo()...}  
class Contable extends Empleado {  
    public double sueldo()...}
```

Si declaramos **Empleado e**; la llamada **e.sueldo()** ejecutaría la operación sueldo dependiendo del tipo dinámico de **e**.

Vinculación en Java

Pero si tenemos:

```
Empleado e;
```

```
Vendedor v;
```

```
e = v;
```

```
e.M( . . . );
```

Siendo **M** un método de la clase **Vendedor**, la llamada **e.M(. . .);** provocaría un error de compilación.

Vinculación en Java

Es decir, en Java:

- La decisión si se puede llamar a un método **M** sobre un objeto **e**, **e.M(. . .)**, depende del tipo estático de **e**.
- La decisión sobre qué método se seleccionaría depende del tipo dinámico de **e**.

Vinculación en C++

- En C++ la vinculación "normal" es estática, dados:

```
class Empleado {  
    double sueldo() ... }  
class Vendedor: public Empleado {  
    double sueldo()...}  
class Contable: public Empleado {  
    double sueldo()...}
```

Si declaramos **Empleado e;** la llamada **e.sueldo()** ejecutaría la operación sueldo de la clase **Empleado**,

Vinculación en C++

Si queremos tener vinculación similar a la de Java debemos:

1. Declarar como **virtual** el método **suelo**. de la clase **empleado**. Al hacer eso, la clase **empleado** se convierte en abstracta.
2. Como las clases abstractas no pueden tener instancias, hay que pasar como argumentos objetos dinámicos

```
class Empleado {  
    virtual double sueldo(); ... }  
class Vendedor: public Empleado {  
    double sueldo()...}  
class Contable: public Empleado {  
    double sueldo()...}
```

```
Empleado* e;
```

En este caso, la llamada `e->sueldo()` ejecutaría la operación sueldo dependiendo del tipo del objeto apuntado por e en ese momento de la ejecución.

Vinculación en Python, Java y en C++

En Python, dado un objeto **e** y la operación **e.op(...)**:

1. En tiempo de ejecución, se calcula cual es la clase **C** de **e**.
2. Si **C** tiene declarada la operación **op(...)**, entonces se ejecuta esa operación.
3. En caso contrario, se ejecuta la operación **op(...)** de la superclase de **C** que la tenga declarada y que sea más cercana en la jerarquía de clases.

Vinculación en Python, Java y en C++

En Java dado un objeto **e** y la operación **e.op(...)**:

1. En tiempo de compilación, se ve cual es la clase **C** de **e**.
2. Si **C** tiene declarada o heredada la operación **op(...)**, entonces se ejecuta la operación del tipo dinámico de **e**, de forma similar a Python.
3. En caso contrario, se señala error.

Vinculación en Python, Java y en C++

En C++, en objetos dinámicos, las cosas funcionan como en Java. En objetos estáticos, todo es estático:

1. En tiempo de compilación, se ve cual es la clase **C** de **e**.
2. Si **C** tiene declarada la operación **op(. . .)**, entonces se ejecuta la operación de su clase.
3. En caso contrario, se ejecuta la operación **op(. . .)** de la superclase de **C** que la tenga declarada y que sea más cercana en la jerarquía de clases.

```
class Empleado {  
    def m(self)  
        ...  
  
class Vendedor (Empleado)  
    def m(self)  
        ...  
  
def f(x)  
    x.m()  
  
...  
  
e = Empleado(...)  
v = Vendedor(...)  
  
e.m()  
v.m()  
  
f(e)  
f(v)
```

```
class Empleado {
    public void m() {...}
class Vendedor extends Empleado
    public void m() {...}
        void p() {...}

...
void f1(Empleado x) {x.m()}
void f2(Vendedor x) {x.m()}

...
Empleado e = new Empleado(...);
Vendedor v = new Vendedor(...);
Empleado e1 = v;
e.m(); v.m(); e1.m(); e.p(); v.p(); e1.p();
f1(e); f1(v); f1(e1);
f2(e); f2(v); f2(e1);
```



```
class Empleado {
    public void m() {...}
class Vendedor extends Empleado
    public void m() {...}
...
void f1(Empleado x) {x.m()}
void f2(Vendedor x) {x.m()}
...
Empleado* e = new Empleado(...);
Vendedor* v = new Vendedor(...);
Empleado e1 = *v;
e.m(); v.m(); e1.m();
f1(e); f1(v); f1(e1);
f2(e); f2(v); f2(e1);
f2((Empleado)e1); f2((Empleado)e);
```

```
class Empleado {
    void m() {...}
class Vendedor: Empleado
    void m() {...}
...
void f1(Empleado x) {x.m()}
void f2(Empleado& x) {x.m()}
void f3(Vendedor& x) {x.m()}
...
Empleado e;
Vendedor v;
e.m(); v.m();
f1(e); f1(v); f2(e); f2(v); f3(e); f3(v);
```

```
class Empleado {
    virtual void m() {...}
class Vendedor: Empleado
    void m() {...}
...
void f1(Empleado x) {x.m()}
void f2(Empleado& x) {x.m()}
void f3(Vendedor& x) {x.m()}
...
Empleado e;
Vendedor v;
e.m(); v.m();
f1(e); f1(v); f2(e); f2(v); f3(e); f3(v);
```

Ámbitos y visibilidad en C++ y Java

En Java y C++, dada una declaración de subclase:

```
class Vendedor: Empleado {...}
```

En **Vendedor** son visibles todas los atributos y métodos de declarados como **public** o **protected** en **Empleado**

Ámbitos y visibilidad en C++ y Java

Dada la declaración:

```
class Vendedor: public Empleado {
```

- Los atributos y métodos privados en **Empleado** no son visibles en **Vendedor**.
- Los atributos y métodos públicos o protegidos (**protected**) en **Empleado** son, respectivamente, públicos o protegidos en **Vendedor**.

En Java la definición de subclase es similar a la **public** de C++

Ámbitos y visibilidad en C++

Dada la declaración:

```
class Vendedor: private Empleado {
```

- Los atributos y métodos privados en **Empleado** no son visibles en **Vendedor**.
- Los atributos y métodos públicos o protegidos en **Empleado** son privados en **Vendedor**.

Ámbitos y visibilidad en C++

Dada la declaración:

```
class Vendedor: protected Empleado {
```

- Los atributos y métodos privados en **Empleado** no son visibles en **Vendedor**.
- Los atributos y métodos públicos o protegidos en **Empleado** son protegidos en **Vendedor**.