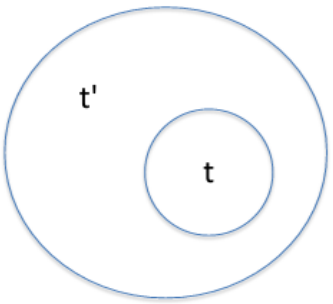
Subtipos y herencia

1. Introducción



Diferentes definiciones de subtipo:

1. t es subtipo de t' si todos los elementos de t son elementos de t'
2. t es subtipo de t' si cualquier función que se puede 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)
3. 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 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
```

2. ¿Qué es un subtipo?

Las definiciones 1 y 2 no son equivalentes:

- Si t es subtipo de t' de acuerdo con la definición 1, entonces también lo es de acuerdo con la definición 2.

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

Definición 3: t es subtipo de t' si los objetos de t se pueden convertir implícitamente a objetos de t' (type cast o coerción)

3. Herencia y subclases

La herencia y la relación de subclase permiten:

- estructurar mejor el código.
- tener una mayor reutilización de código.
- simplificar el diseño.

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++).

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 clases, 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)

4. Comprobación e inferencia de tipos con subtipos

La estructura de tipos:

- Si e :: t y t <= t', entonces e :: t'
- Si e :: t, t <= t' y f :: t' -> t'', entonces f e :: t''

Consecuencia:

- Si e :: t, t <= t' y f :: t' -> t', entonces f e :: t'

  NO podemos asegurar que f e :: t

Por ejemplo, si x: par y par <= int y

bool es_positivo (int n);

int incre_1 (int n);

entonces:

- es_positivo(x) :: bool
- incre_1 (x) :: int, pero no incre_1 (x) :: par

  Por ejemplo, si x == 6, incre_1 (x) no sería par

En el caso de la asignación, si x :: t, e :: t' y t' <= t, entonces:

- x = e, ¿es correcto? SI

  Ej.: x :: int y z :: par, es correcto x = z;

En el caso de la asignación, si x :: t, e :: t' y t <= t', entonces:

- x = e, ¿es correcto? NO

  Ej.: si x :: par no es correcto x = 7;

5. Covarianza y contravarianza

Si s <= t y s' <= t': ¿sería correcto decir (s -> s') <= (t -> t')?

NO

Supongamos que f :: par -> par y g :: int -> int.

Se supone que si (s -> s') <= (t -> t'), siempre que podamos usar g, podemos usar f en su lugar. Sin embargo, si tenemos la expresión g 5, e intentamos sustituir 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.

En cambio, si t <= s y s' <= t', entonces sí podemos asegurar que (s -> s') <= (t -> t')

Si s <= t, ¿podemos asegurar que List<s> <= 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?
```

- Un constructor de tipos C, C es **covariante** si s <= t implica que C<s> <= C<t>.
- C es **contravariante** si s <= t implica que C<t> <= C<s>.
- C es **invariante** si s <= t no implica C<s> <= C<t> ni lo contrario.
- El constructor -> es contravariante en el primer argumento y covariante en el segundo.

6. Subclases y Herencia en Python, C++ y Java

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.

C++:

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

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()
```

Java:

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

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

```
Interface Empleado {...}

Interface Comisionista {...}

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

Python:

```
class Empleado:
    ...

class Vendedor (Empleado)
    ...
```

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.

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.

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.

Si se quiere tener una vinculación similar a la de Java, se debe:

- Declarar como virtual el método sueldo de la clase empleado. Al hacer eso, la clase empleado se convierte en abstracta.

- 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.

Resumen:

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.

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.

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

- En tiempo de compilación, se ve cual es la clase C de e.
- Si C tiene declarada la operación op(…), entonces se ejecuta la operación de su clase.
- 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.

Ámbitos y visibilidad en C++ y Java

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

```
class Vendedor: Empleado { … }
```

En Vendedor son visibles todos los atributos y métodos declarados como public o protected en Empleado.

Los atributos y métodos privados en Empleado no son visibles en Vendedor.

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

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++.

```
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.

```
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.