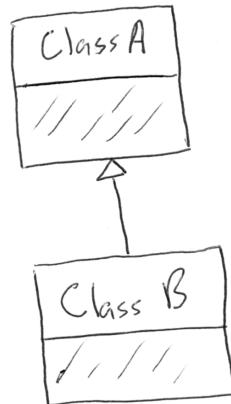


Llenguatges de Programació

Conceptes avançats



Jordi Petit, Fernando Orejas, Gerard Escudero

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona



Contingut

- Recursivitat
 - *Tail Call*
 - *Continuation-Passing Style*
 - *Trampolining*
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
- Programació asíncrona

Recursivitat

Tail Recursion: la crida recursiva es fa just abans de retornar el valor.

Factorial recursiu:

```
def f_rec(n):  
    if n == 0:  
        return 1  
    else:  
        return n * f_rec(n-1)
```

Factorial tail recursion:

```
def f_tailrec(n, resultat = 1):  
    if n == 0:  
        return resultat  
    else:  
        return f_tailrec(n-1, n*resultat)
```

- **Tail Recursion Optimization**: optimització en que el compilador substitueix la crida per recursiva per un salt.
- Python no ho suporta:

```
f_rec(1000)  
RecursionError: maximum recursion depth exceeded in comparison
```

- Altra opció és passar-la a **iterativa**.

Tail Recursion

Però encara és útil:

Recursivitat normal:

```
def slow_fib(n):  
    if n < 2:  
        return 1  
    else:  
        return slow_fib(n - 1) +  
               slow_fib(n - 2)
```

```
fib(40) = 165580141  
temps(s): 21.715350
```

Tail recursion:

```
def quick_fib(n, acc1=1, acc2=1):  
    if n < 2:  
        return acc1  
    else:  
        return quick_fib(n - 1,  
                          acc1 + acc2,  
                          acc1)
```

```
fib(40) = 165580141  
temps(s): 0.000117
```

* Hem utilitzat el mòdul `pytictoc` per mostrar el temps d'execució.

Tail Recursion

Per què funciona?

```
>>> slow_fib(4)
4 ##          # paràmetre i pila
3 ###
2 ####
1 #####
0 #####
1 #####
2 ###
1 #####
0 #####
5          # resultat
```

```
>>> quick_fib(4)
4 ##
3 ###
2 ####
1 #####
5
```

* Hem utilitzat el mòdul `traceback` per mostrar la mida de la pila.

Contingut

- Recursivitat
 - *Tail Call*
 - *Continuation-Passing Style*
 - *Trampolining*
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
- Programació asíncrona

Continuation-Passing Style

És una tècnica de la programació funcional en la que es retornen les funcions a aplicar al resultat, en lloc dels valors.

Exemple:

```
expr = lambda: (1 + 2) * 3 + 4  
expr() ➡ 13
```

Fem una funció per cada operació.

```
mes2 = lambda x, cont: cont(2 + x)  
per3 = lambda x, cont: cont(3 * x)  
mes4 = lambda x, cont: cont(4 + x)  
  
def expr_cps(x, cont):  
    mes2(x, (lambda y:  
        per3(y, (lambda z:  
            mes4(z, (lambda res:  
                cont(res)))))))  
  
expr_cps(1, print) ➡ 13
```

Com funciona la pila?

```
expr ##  
mes2 ###  
per3 #####  
mes4 #####  
13
```

Recorda una mica als *thunks* del Haskell.

CPS amb funcions recursives

Amb una funció recursiva és més natural.

Exemple:

```
identitat = lambda x: x

def fact_cps(n, cont):
    if n == 0:
        return cont(1)
    else:
        return fact_cps(n - 1,
                        lambda value:
                            cont(n * value))

fact_cps(6, identitat) ➡ 720
```

Pila?

```
fact_cps ##
fact_cps ###
fact_cps ####
fact_cps #####
fact_cps #####
fact_cps #####
fact_cps #####
fact_cps #####
720
```

Continuem tenint el problema de la pila:

```
fact_cps(1000, identitat)
RecursionError: maximum recursion depth exceeded while calling a Python object
```


Contingut

- Recursivitat
 - *Tail Call*
 - *Continuation-Passing Style*
 - *Trampolining*
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
- Programació asíncrona

Trampolining

És una tècnica que evita el creixement de la pila.

Funció trampolí:

```
def trampoline(f, *args):  
    v = f(*args)  
    while callable(v):  
        v = v()  
    return v
```

```
trampoline(fact_cps2, 6, identitat)  
👉 720
```

Crida a la funció mentre la continuació sigui de tipus funció (*callable*).

Adaptació per al trampolí:

```
def fact_cps1(n, cont):  
    if n == 0:  
        return cont(1)  
    else:  
        return fact_cps1(n - 1,  
                           lambda value:  
                               cont(n * value))  
  
def fact_cps2(n, cont):  
    if n == 0:  
        return cont(1)  
    else:  
        return lambda: fact_cps2(n - 1,  
                                   lambda value:  
                                       lambda: cont(n * value))
```

Trampolining

Funcionament de la pila:

```
trampoline(fact_cps2, 6, identitat)
```



```
trampoline ##  
fact_cps2 ###  
fact_cps2 ####  
fact_cps2 ####  
fact_cps2 ####  
fact_cps2 ####  
fact_cps2 ####  
fact_cps2 ####  
fact_cps2 ####
```

720

Funciona!

```
trampoline(fact_cps2, 1000, identitat)
```



4023872600770.....0000000000000000

Contingut

- Recursivitat
- Orientació a Objectes
 - Herència
 - Declaració de subclasses
 - Vinculació
- Subtipus i variància de tipus
- Clausures
- Programació asíncrona

Programació orientada a objectes

Elements principals de la POO:

- Reutilització de codi
- Modularitat
- Facilitat de manteniment
- Ampliació de funcionalitats
- Abstracció
- Encapsulació
- Herència

Herència i subclasses

L'herència i la relació de subclasses tenen per objectiu:

- Estructurar millor el codi.
- Reaprofitar millor el codi.
- Simplificar el disseny.

Herència i subclasses

Exemple:

```
class Empleat {...}  
  
function sou(e: Empleat): number {...}  
  
e = new Empleat()  
s = sou(e)
```

Amb programació "clàssica":

```
function sou(e: Empleat): number {  
  if (e.es_venedor()) {  
    ...  
  } else if (e.es_contable()) {  
    ...  
  } else if (e.es_executiu()) {  
    ...  
  }  
}
```

Amb POO:

```
class Empleat {  
  function sou(): number {...}  
  ...  
}  
  
class Venedor extends Empleat {  
  function sou(): number {...}  
  ...  
}  
  
class Comptable extends Empleat {  
  function sou(): number {...}  
  ...  
}
```

Herència i subclasses

A cada subclasse es poden re definir operacions de la classe base.

```
class Empleat {  
    function sou(): number {...}  
}  
  
class Venedor extends Empleat {  
    function sou(): number {...}  
}  
  
class Comptable extends Empleat {  
    function sou(): number {...}  
}
```


Herència i subclasses

A cada subclasse es poden definir noves operacions.

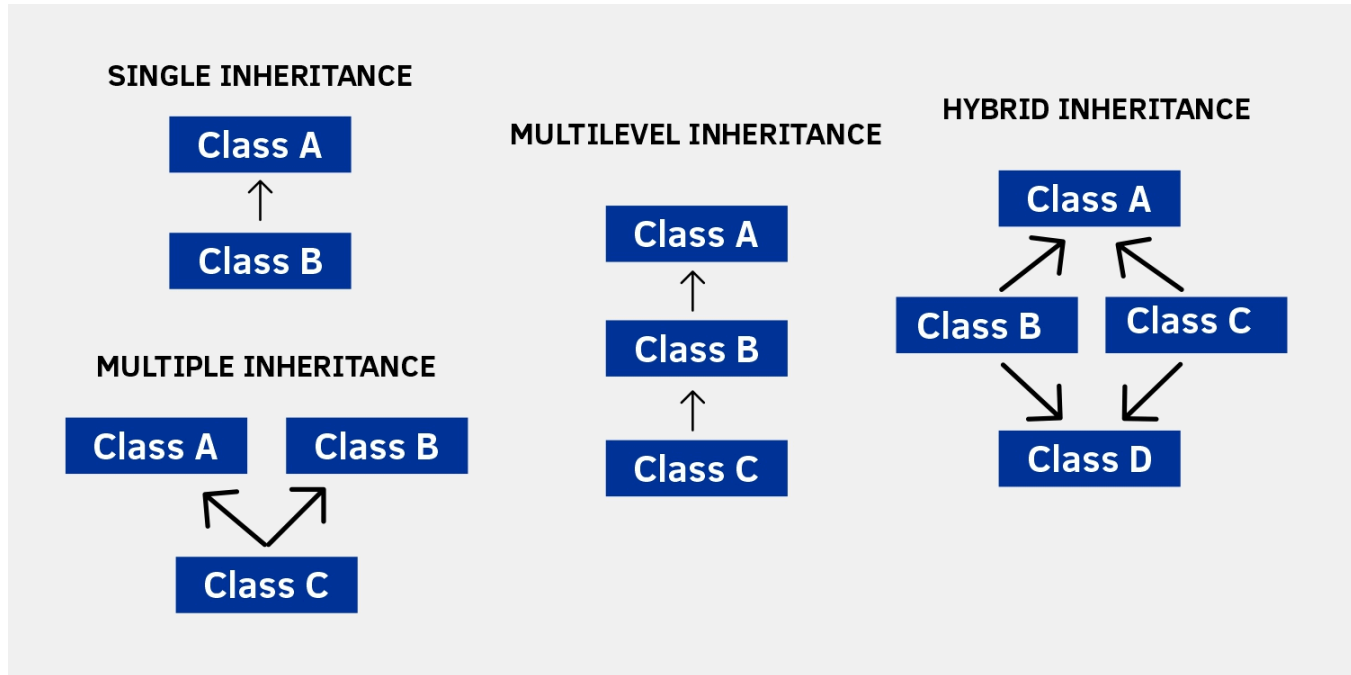
```
class Empleat {  
    function sou(): number {...}  
}  
  
class Venedor extends Empleat {  
    function comissio(): number {...}  
}  
  
class Comptable extends Empleat {  
    function fulls_de_calcul(): FullCalcul[] {...}  
}
```

Herència i subclasses

L'operació que es crida depèn de la (sub)classe de l'objecte en temps d'execució (*late binding*).

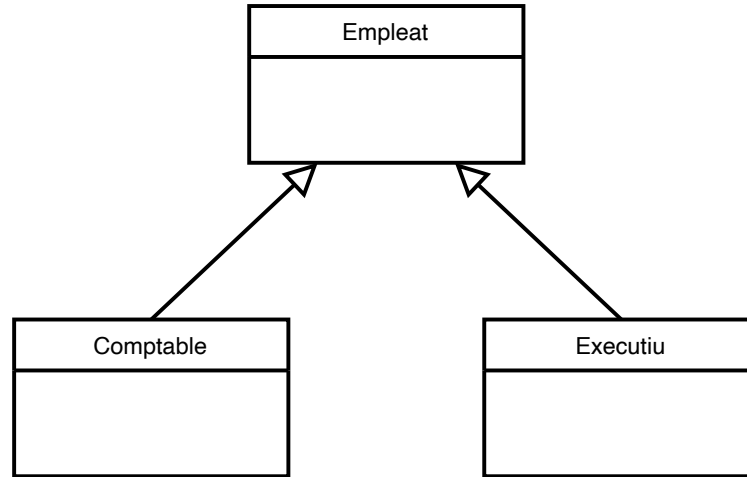
```
function escriure(e: Empleat) {  
    print(e.nom, e.sou())  
}  
  
Empleat e = new Empleat()  
Empleat v = new Venedor()  
Empleta c = new Comptable()  
  
escriure(e)      // usa el sou() d'Empleat  
escriure(v)      // usa el sou() de Venedor  
escriure(c)      // usa el sou() de Comptable
```

Herència



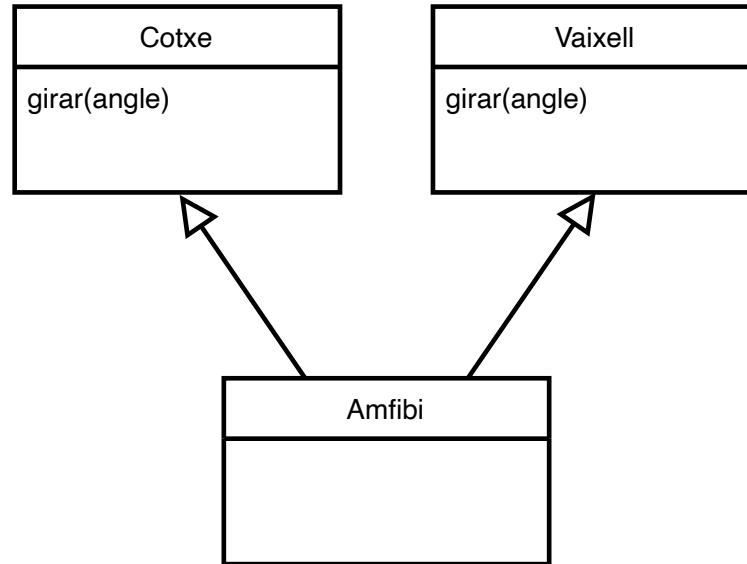
Herència simple

Una classe només pot ser subclasse d'una altra classe.



Herència múltiple

Una classe pot ser subclasse de més d'una classe.



📖 Vaixell amb Rodes d'en J. Petit: Oasi #25

Promesa de l'OO

Si es canvia l'estructura salarial:

- En programació "clàssica" cal refer del tot la funció `sou()` (i potser més operacions).
- En programació "OO", es canvien les classes i el mètode `sou()` d'algunes.

Contingut

- Recursivitat
- Orientació a Objectes
 - Herència
 - Declaració de subclasses
 - Vinculació
- Subtipus i variància de tipus
- Clausures
- Programació asíncrona

Declaració de subclasses en C++

```
class Empleat { ... };  
class Venedor: Empleat { ... };
```

O també:

```
class Venedor: public Empleat { ... };  
class Venedor: protected Empleat { ... };  
class Venedor: private Empleat { ... };
```

Amb herència múltiple:

```
class Cotxe { ... };  
class Vaixell { ... };  
class Hibrid: public Cotxe, public Vaixell { ... };
```

Resolució de conflictes:

```
hibrid.Cotxe::girar(90);  
hibrid.Vaixell::girar(90);
```


Declaració de subclasses en Java

```
class Empleat { ... }  
class Venedor extends Empleat { ... }
```

En Java no hi herència múltiple amb classes, però sí amb interfícies:

```
interface Cotxe { ... }  
interface Vaixell { ... }  
class Hibrid implements Cotxe, Vaixell { ... }
```

Les interfícies de Java són com les classes de Haskell (quin embolic!).

Declaració de subclasses en Python

```
class Empleat:  
    ...  
  
class Venedor(Empleat):  
    ...
```

Amb herència múltiple:

```
class Híbrid(Cotxe, Vaixell):  
    ...
```

Resolució de conflictes:

- Quan a les dues classes hi ha mètodes amb el mateix nom, s'hereta el de la primera.

Visibilitat dels membres

Els LPs limiten la visibilitat dels membres (atributs i mètodes) de les classes:

Ajuda a:

- Encapsular els objectes en POO.
- Definir una interfície clara i independent de la implementació.
- Prevenir errors en el codi.

Visibilitat en C++

Els **especificadors d'accés** defineixen la visibilitat dels membres d'una classe.

```
class Classe {  
    public:  
    ...  
    protected:  
    ...  
    private:  
    ...  
};
```

- **public:** els membres són visibles des de fora de la classe
- **privat:** no es pot accedir (ni veure) als membres des de fora de la classe
- **protegit:** no es pot accedir als membres des de fora de la classe, però s'hi pot accedir des de classes heretades.

```
class Classe {  
    // privat per defecte  
};
```

```
struct Estructura {  
    // public per defecte  
};
```

Visibilitat en C++

Els **especificadors d'accés** també defineixen la visibilitat dels membres quan es deriva una classe:

```
class SubClasse: public Classe { ... };
```

- Els membres protegits de **Classe** són membres protegits de **SubClasse**.
- Els membres públics de **Classe** són membres públics de **SubClasse**.

```
class SubClasse: protected Classe { ... };
```

- Els membres protegits i públics de **Classe** són membres protegits de **SubClasse**.

```
class SubClasse: private Classe { ... };  
class SubClasse: Classe { ... };           // 'private' per defecte en classes
```

- Els membres públics i protegits de **Classe** són membres privats de **SubClasse**.

Visibilitat en C++

```
class A {  
    public:  
        int x;  
    protected:  
        int y;  
    private:  
        int z;  
};  
  
class B : public A {  
    // x és public  
    // y és protegit  
    // z no és visible des de B  
};  
  
class C : protected A {  
    // x és protegit  
    // y és protegit  
    // z no és visible des de C  
};  
  
class D : private A {  
    // x és privat  
    // y és privat  
    // z no és visible des de D  
};
```

Visibilitat en Java

Els **nivells d'accés** defineixen la visibilitat dels membres (atributs i mètodes) d'una classe.

```
class Classe {  
    public ...  
    protected ...  
    private ...  
    ...  
}
```

- **public**: aquest membre és accessible des de fora de la classe
- **privat**: no es pot accedir (ni veure) en aquest membre des de fora de la classe
- **protegit**: no es pot accedir en aquest membre des de fora de la classe, però s'hi pot accedir des de classes heretades.
- *res*: només el codi en el **package** actual pot accedir aquest membre.

Visibilitat en Java

En Java no es pot limitar la visibilitat heretant classes (sempre és "public").

```
class SubClasse extends Classe { ... }
```

- Els membres protegits de Classe són membres protegits de SubClasse.
- Els membres públics de Classe són membres públics de SubClasse.

Visibilitat en Java

```
package p;

public class A {
    public int a;
    protected int b;
    private int c;
    int d;
}

class B extends A {
    // a és visible des de B
    // b és visible des de B
    // c no és visible des de B
    // d és visible des de B
}

// A.a és visible des de p
// A.b és visible des de p
// A.c no és visible des de p
// A.d és visible des de p
```

```
package q;

import p.*;

class C extends p.A {
    // a és visible des de C
    // b és visible des de C
    // c no és visible des de C
    // d no és visible des de C
}

// A.a és visible des de q
// A.b no és visible des de q
// A.c no és visible des de q
// A.d no és visible des de q
```

Visibilitat en Python

En Python no hi ha restriccions de visibilitat.

Tot és visible.

Per *convenció*, els membres que comencen per `_` (però no per `__`) són privats.

Contingut

- Recursivitat
- Orientació a Objectes
 - Herència
 - Declaració de subclasses
 - Vinculació
- Subtipus i variància de tipus
- Clausures
- Programació asíncrona

Tipatge estàtic i tipatge dinàmic

Tipatge estàtic: La verificació de tipus que es realitza durant la compilació del codi.


- El compilador comprova si les variables s'utilitzen de manera coherent amb el seu tipus durant la compilació del codi.
- Si hi ha un error de tipus, el compilador no genera codi.
- Ajuda a detectar i corregir errors abans d'executar el codi, evitant problemes durant l'execució.

Tipatge dinàmic: La verificació de tipus que es realitza durant l'execució del codi.


- El tipus de la variable es determina en temps d'execució
- Si hi ha un error de tipus, aquest no es detectarà fins que el codi s'executi.


Late binding (vinculació)


El **late binding** és el procés pel qual es determina (en temps d'execució) quin mètode cal cridar en funció del tipus dinàmic d'un objecte.

```
class Animal {  
    parlar() {  
        print("grr")  
    }  
}  
  
class Gat extends Animal {  
    parlar() {  
        print("mèu")  
    }  
}  
  
class Gos extends Animal {  
    parlar() {  
        print("bub")  
    }  
}  
  
function parlarN(animal: Animal,  
                 n: number) {  
    repeat (n) {  
        animal.parlar()  late binding  
    }  
}
```


```
animal: Animal = new Animal()  
gat: Gat = new Gat()  
gos: Gos = new Gos()
```


animal.parlar()  grr

gat.parlar()  mèu

gos.parlar()  bub

parlarN(animal, 3)  grr grr grr

parlarN(gat, 3)  mèu mèu mèu

parlarN(gos, 3)  bub bub bub

Vinculació en Java

En Java, els objectes tenen un tipus estàtic i un tipus dinàmic:

```
Animal animal;  
animal = new Gat();
```

- El tipus estàtic d'`animal` és `Animal`.
- El tipus dinàmic d'`animal` és `Gat`.

El tipus dinàmic ha de ser un subtipus del tipus estàtic.

En temps de compilació, es comprova que les crides es puguin aplicar al tipus estàtic.

En temps d'execució, la vinculació es fa en funció del tipus dinàmic.

Vinculació en Java

Donada una declaració `C c;` i una operació `c.m()`:

- En temps de compilació, es verifica que la classe `C` tingui el mètode `m` (directament o a través d'herència).
- En temps d'execució, es crida al `m` de la classe corresponent al tipus dinàmic de `c` o de la seva superclasse més propera que l'implementi.

Vinculació en Java

```
class Animal {  
    void parlar() {  
        print("grr");  
    }  
}  
  
class Gat extends Animal {  
    void parlar() {  
        print("mèu");  
    }  
    void filar() {  
        print("rum-rum");  
    }  
}  
  
void parlarN(Animal animal, int n) {  
    for (int i = 0; i < n; ++i) {  
        animal.parlar();  
    }  
}
```

```
Animal animal = new Animal();  
Gat gat = new Gat();
```

```
animal.parlar();      ➡ grr
```

```
gat.parlar();         ➡ mèu
```

```
parlarN(animal, 3);   ➡ grr grr grr
```

```
parlarN(gat, 3);      ➡ mèu mèu mèu
```

```
gat.filar();          ➡ rum-rum
```

```
animal.filar()        ❌ error compilació
```


Vinculació en Python

En Python, el tipus dels objectes és dinàmic.

```
>>> e = Empleat()
>>> v = Venedor()
>>> type(e)
<class '__main__.Empleat'>
>>> type(v)
<class '__main__.Venedor'>
>>> v = e
>>> type(v)
<class '__main__.Empleat'>
```

Donada una operació `c.m()`:

- En temps d'execució, es crida al `m` de la classe corresponent al tipus dinàmic de `c` o de la seva superclasse més propera que l'implementi.

Vinculació en Python

```
class Animal:
    def parlar(self):
        print("grr")

class Gat(Animal):
    def parlar(self):
        print("mèu")
    def filar(self):
        print("rum-rum")

def parlarN(animal, n):
    for _ in range(n):
        animal.parlar()
```

```
animal = Animal()
gat = Gat()
```

animal.parlar() 📌 grr

gat.parlar() 📌 mèu

parlarN(animal, 3) 📌 grr grr grr

parlarN(gat, 3) 📌 mèu mèu mèu

gat.filar(); 📌 rum-rum

animal.filar() ❌ error execució

Vinculació en C++

En C++, els objectes estàtics tenen un tipus estàtic.

```
Animal a = Gat();
```

- El tipus estàtic d'a és `Animal`: quan se li assigna un `Gat` es perd la part extra.
- (Recordeu: El pas per còpia fa una assignació)

Els objectes dinàmics (punters i referències) tenen un tipus estàtic i un tipus dinàmic.

```
Animal* a = new Gat();
```

- El tipus estàtic d'a és punter a `Animal`.
- El tipus dinàmic d'a és punter a `Gat`.

```
Animal& a = Gat();
```

- El tipus estàtic d'a és referència a `Animal`.
- El tipus dinàmic d'a és referència a `Gat`.

Vinculació en C++

Per a objectes estàtics, la vinculació és estàtica.

El tipus dinàmic ha de ser un subtipus del tipus estàtic.

En temps de compilació, es comprova que les crides es puguin aplicar al tipus estàtic.

En temps d'execució, la vinculació es fa en funció del tipus dinàmic, sobre els mètodes marcats `virtual`.

Vinculació en C++

```
class Animal {  
    virtual void parlar() {  
        print("grr");  
    }  
}  
  
class Gat: Animal {  
    virtual void parlar() {  
        print("mèu");  
    }  
    virtual void filar() {  
        print("rum-rum");  
    }  
}  
  
void parlarN(Animal animal, n: int) {  
    for (int i = 0; i < n; ++i) {  
        animal.parlar();  
    }  
}
```

```
Animal animal;  
Gat gat;
```

```
animal.parlar();
```

👉 grr

```
gat.parlar();
```

👉 mèu

```
parlarN(animal, 3);
```

👉 grr grr grr

```
parlarN(gat, 3);
```

👉 grr grr grr *

```
gat.filar();
```

👉 rum-rum

```
animal.filar();
```

❌ error compilació

* Com que `parlarN` rep un `Animal` per còpia, al cridar `parlarN(gat, 3)` es perd la part de `gat`.

Vinculació en C++

```
class Animal {  
    virtual void parlar() {  
        print("grr");  
    }  
}  
  
class Gat: Animal {  
    virtual void parlar() {  
        print("mèu");  
    }  
    virtual void filar() {  
        print("rum-rum");  
    }  
}  
  
void parlarN(Animal* animal, n: int) {  
    for (int i = 0; i < n; ++i) {  
        animal->parlar();  
    }  
}
```

```
Animal animal;  
Gat gat;
```

```
animal.parlar();    👉 grr
```

```
gat.parlar();       👉 mèu
```

```
parlarN(animal, 3);    👉 grr grr grr
```

```
parlarN(&gat, 3);      👉 mèu mèu mèu *
```

```
gat.filar();         👉 rum-rum
```

```
animal.filar()       ❌ error compilació
```

* Com que `parlarN` rep un punter a `Animal`, al cridar `parlarN(gat, 3)` el tipus dinàmic continua sent `Gat`.

Vinculació en C++

```
class Animal {  
    virtual void parlar() {  
        print("grr");  
    }  
}  
  
class Gat: Animal {  
    virtual void parlar() {  
        print("mèu");  
    }  
    virtual void filar() {  
        print("rum-rum");  
    }  
}  
  
void parlarN(Animal& animal, n: int) {  
    for (int i = 0; i < n; ++i) {  
        animal.parlar();  
    }  
}
```

```
Animal animal;  
Gat gat;
```

```
animal.parlar();    👉 grr
```

```
gat.parlar();       👉 mèu
```

```
parlarN(animal, 3);    👉 grr grr grr
```

```
parlarN(gat, 3);       👉 mèu mèu mèu *
```

```
gat.filar();         👉 rum-rum
```

```
animal.filar()       ❌ error compilació
```

* Com que `parlarN` rep un `Animal` per referència, al cridar `parlarN(gat, 3)` el tipus dinàmic continua sent `Gat`.

Vinculació en C++

```
class Animal {  
    void parlar() {  
        print("grr");  
    }  
}  
  
class Gat: Animal {  
    void parlar() {  
        print("mèu");  
    }  
}  
  
void parlarN(Animal& animal, n: int) {  
    for (int i = 0; i < n; ++i) {  
        animal.parlar();  
    }  
}
```

```
Animal animal;  
Gat gat;
```

```
parlarN(animal, 3);    🙌 grr grr grr  
parlarN(gat, 3);     🙌 grr grr grr *
```

* Com que `parlar` no és `virtual`,
`parlarN` no fa late binding.

Contingut

- Recursivitat
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
- Programació asíncrona

Noció de subtipus

Definició 1:

s és subtipus de t si tots els valors d' s són valors de t .

Exemple en Pearl:

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



➡ Aquesta mena de subtipus no són habituals en els LPs.

Noció de subtipus

Definició 2:

s és subtipus de **t** si qualsevol funció que es pot aplicar a un objecte de tipus **t** es pot aplicar a un objecte de tipus **s**.

Exemple en C++:

```
class Forma;  
class Quadrat: Forma;           // Forma és subtipus de Forma  
  
double area(const Forma& f);  
  
Forma f;                           
area(f);  
Quadrat q;                         
area(q);
```

➡ Aquesta és la definició en què es basa la programació orientada a objectes.

Noció de subtipus

Definició 2':

s és subtipus de t si en tot context que es pot usar un objecte de tipus t es pot usar un objecte de tipus s .

➡ Aquesta és la definició en què (a vegades es diu que) es basa la programació orientada a objectes.

Noció de subtipus

Les definicions 1 i 2 no són equivalents:

- Si s és subtipus de t segons la Def. 1, llavors també ho és d'acord amb la Def. 2.
- La inversa, en general, no és certa. És a dir, si s és subtipus de t d'acord amb la Def. 2, llavors no té perquè ser-ho d'acord amb la Def. 1.

Exemple:

```
class T {  
    int x;  
};  
  
class S : T {  
    int y;  
}
```

Els valors de S no es poden veure com un subconjunt dels valors de T , ja que tenen més elements.

Noció de subtipus

Definició 3:

s és subtipus de t si tots els objectes de s es poden convertir implícitament a objectes de t (*type casting* o coerció).

Comprovació i inferència amb subtipus

- Si $e :: s$ i $s \leq t$, llavors $e :: t$.
- Si $e :: s$, $s \leq t$ i $f :: t \rightarrow t'$, llavors $f e :: t'$.

La notació $e :: t$ indica que e és de tipus de t .

La notació $s \leq t$ indica que s és un subtipus de t .

Comprovació i inferència amb subtipus

- Si `e :: s` i `s <= t`, llavors `e :: t`.
- Si `e :: s`, `s <= t` i `f :: t -> t'`, llavors `f e :: t'`.

Per tant,

- Si `e :: s`, `s <= t` i `f :: t -> t`, llavors `f e :: t`.

Però no podem assegurar que `f e :: s`! Per exemple, si tenim

- `x :: parell`
- `parell <= int`
- `function es_positiu(int): boolean`
- `function incrementa(int): int`

Llavors

- `es_positiu(x) :: bool` ✓
- `incrementa(x) :: int` ✓
- `incrementa(x) :: parell` ✗

El cas de l'assignació

- Si `x :: t` i `e :: s` i `s <= t`, llavors `x = e` és una assignació correcta.
- Si `x :: s` i `e :: t` i `s <= t`, llavors `x = e` és una assignació incorrecta.

Exemples:

- Si `x :: Int` i `e :: Parell`, `x = e` no té problema.
- Si `x :: Parell` i `e :: Int`, `x = e` crearia un problema: `e` potser no és parell.

El cas de les funcions

- Si $s \leq t$ i $s' \leq t'$, llavors $(s \rightarrow s') \leq (t \rightarrow t')$?

El cas de les funcions

- Si $s \leq t$ i $s' \leq t'$, llavors $(s \rightarrow s') \leq (t \rightarrow t')$?

No!

Suposem que $f :: \text{parell} \rightarrow \text{parell}$ i que $g :: \text{int} \rightarrow \text{int}$.

Si $(s \rightarrow s') \leq (t \rightarrow t')$, llavors sempre que puguem usar g , podem usar f al seu lloc. Com que $g\ 5$ és legal, $f\ 5$ també seria legal. Però f espera un parell i 5 no ho és.

El cas de les funcions

- Si $s \leq t$ i $s' \leq t'$, llavors $(s \rightarrow s') \leq (t \rightarrow t')$?

No!

Suposem que $f :: \text{parell} \rightarrow \text{parell}$ i que $g :: \text{int} \rightarrow \text{int}$.

Si $(s \rightarrow s') \leq (t \rightarrow t')$, llavors sempre que puguem usar g , podem usar f al seu lloc. Com que $g\ 5$ és legal, $f\ 5$ també seria legal. Però f espera un `parell` i 5 no ho és.

- En canvi, si $s \leq t$ i $s' \leq t'$, llavors $(t \rightarrow s') \leq (s \rightarrow t')$ és correcte.

El cas dels constructors de tipus

- Si $s \leq t$, podem assegurar que $\text{List } s \leq \text{List } t$?

El cas dels constructors de tipus

- Si $s \leq t$, podem assegurar que $\text{List } s \leq \text{List } t$?

No!

```
class Animal
class Gos extends Animal
class Gat extends Animal

function f(animals: List<Animal>) {
  animals.push(new Gat())      // perquè no?
}

gossos: List<Gos> = ...
f(gossos)                     // ai, ai
```

El cas dels constructors de tipus

- Si $s \leq t$, podem assegurar que $\text{List } t \leq \text{List } s$?

El cas dels constructors de tipus

- Si $s \leq t$, podem assegurar que $\text{List } t \leq \text{List } s$?

No!

```
class Animal
class Gos extends Animal {
    function borda() {...}
}
class Gat extends Animal;

function f(gossos: List<Gos>) {
    for (var gos: Gos of gossos) gos.borda()
}

List<Animal> animals = [new Gos(), new Animal, new Gat()]
f(animals)           // alguns animals no borden 🐱
```


Variància de constructors de tipus

Sigui C un constructor de tipus i sigui $s \leq t$.

- Si $C\ s \leq C\ t$, llavors C és **covariant**.
- Si $C\ t \leq C\ s$, llavors C és **contravariant**.
- Si no és covariant ni contravariant, llavors C és **invariant**.

Variància de constructors de tipus

Sigui C un constructor de tipus i sigui $s \leq t$.

- Si $C\ s \leq C\ t$, llavors C és **covariant**.
- Si $C\ t \leq C\ s$, llavors C és **contravariant**.
- Si no és covariant ni contravariant, llavors C és **invariant**.

Hem vist doncs que:

- El constructor \rightarrow és contravariant amb el primer paràmetre.
- El constructor \rightarrow és covariant amb el segon paràmetre.
- El constructor `List` és invariant.

Contingut

- Recursivitat
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
 - *Partial*
 - Objectes sense classe
 - Memorització
 - Decoradors
- Programació asíncrona

Closures

Al retornar funcions es creen closures (*closure*):

- tanca l'abast (*scope*) lèxic del voltant i captura els seus valors.

```
def interna(x):  
    z = "!"  
    return lambda y: print(x, y, z)  
  
externa = interna("Hola")  
  
externa("món")  ➡  Hola món !
```

Fixeu-vos en que si la funció interna tornés només una funció, no funcionaria.

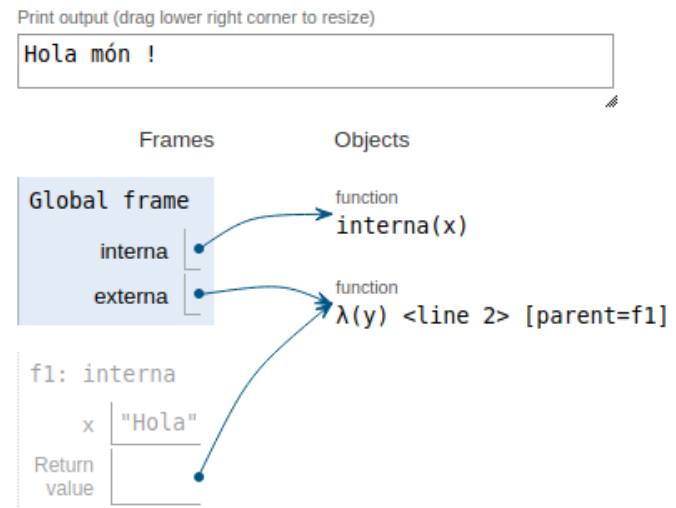


diagrama: [Python Tutor](#)

Contingut

- Recursivitat
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
 - *Partial*
 - Objectes sense classe
 - Memorització
 - Decoradors
- Programació asíncrona

Partial

Podem currificar en python?

```
def partial ( f , x ):  
    def g(* args ):  
        return f (*( ( x ,) + args ))  
    return g
```

Exemple:

```
multiplica = lambda x, y: x * y  
doble = partial(multiplica , 2)
```

```
doble(3) ➡ 6
```

Partial++

Més general:

```
def partialN (* args ):  
    def g(* args2 ):  
        f = args [0]  
        xs = args [1:] + args2  
        return f (* xs)  
    return g
```

Exemple:

```
from functools import reduce  
  
def sumaRed(*args):  
    suma = lambda x, y: x + y  
    return reduce(suma, list ( args ), 0)
```

sumaRed(1,2,3,4) 🙌 10

```
sumaRed10 = partialN(sumaRed, 1, 2, 3, 4)
```

print(sumaRed10(5, 5, 5)) 🙌 25

Contingut

- Recursivitat
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
 - *Partial*
 - Objectes sense classe
 - Memorització
 - Decoradors
- Programació asíncrona

Objete punt sense classe

```
from math import pi, atan2, degrees

def punt(x, y):
    def temp(*args):
        if args[0] == 'crt':
            return x, y
        elif args[0] == 'plr':
            return (x ** 2 + y ** 2) ** 0.5, \
                degrees(atan2(y, x))
        elif args[0] == 'dst':
            p2 = args[1]('crt')
            return ((x-p2[0])**2 + (y-p2[1])**2)**0.5
    return temp
```

```
punt(1, 0)('crt') ➡ (1, 0)
punt(1, 0)('plr') ➡ (1.0, 0.0)
punt(1, 1)('crt') ➡ (1, 1)
punt(1,1)('plr') ➡ (1.4142135623730951, 45.0)
punt(1, 1)('dst', punt(2, 0)) ➡ 1.4142135623730951
punt(1, 1)('dst', punt(1, 1)) ➡ 0.0
```

Contingut

- Recursivitat
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
 - *Partial*
 - Objectes sense classe
 - Memorització
 - Decoradors
- Programació asíncrona

Test de funcions

```
from pyticToc import TicToc

def test(n):
    def prec(x):
        return '{:.6f}'.format(x)

    def clausura(f):
        t = TicToc()
        t.tic()
        print('f(', n, ') = ', f(n), sep='')
        print('temps(s):', prec(t.tocvalue()))

    return clausura
```

```
def fib(n):
    if n in [0, 1]:
        return n
    return fib(n-1) + fib(n-2)
```

```
test40 = test(40)
test40(fib)
👉
f(40) = 102334155
temps(s): 23.586690
```

Memorització genèrica

```
def memoritza (f):  
    mem = {}          # la memòria  
  
    def f2 (x):  
        if x not in mem:  
            mem[x] = f(x)  
        return mem[x]  
    return f2
```

```
fib = memoritza(fib)  
test40(fib)
```



```
f(40) = 102334155  
temps(s): 0.000051
```

Contingut

- Recursivitat
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
 - *Partial*
 - Objectes sense classe
 - Memorització
 - Decoradors
- Programació asíncrona

Decoradors

Són un mètode per alterar quelcom invocable (*callable*).

Ho podem fer mitjançant les clausures.

```
def testDec(f):
    def wrapper(*args):
        valor = f(*args)
        print('fib(' + str(args[0]) + ') = ' + \
              str(valor))
        return valor

    return wrapper

@testDec
def fib(n):
    if n in [0, 1]:
        return n
    return fib(n-1) + fib(n-2)
```

```
test4 = test(4)
test4(fib)
👉
fib(1) = 1
fib(0) = 0
fib(2) = 1
fib(1) = 1
fib(3) = 2
fib(1) = 1
fib(0) = 0
fib(2) = 1
fib(4) = 3
f(4) = 3
temps(s): 0.000089
```

Funciona també aplicant `fib = memoritza(fib)`.

Decoradors parametritzats

Podem afegir arguments parametritzant els decoradors:

```
def testInterval(inici, fi):
    def decorador(f):
        def wrapper(*args):
            valor = f(*args)
            n = args[0]
            if inici <= n <= fi:
                print('fib(' + str(n) + ') = ' + \
                      str(valor))
            return valor
        return wrapper
    return decorador

@testInterval(35, 40)
def fib(n):
    if n in [0, 1]:
        return n
    return fib(n-1) + fib(n-2)
```

```
test40(fib)
```



```
fib(35) = 9227465
fib(36) = 14930352
fib(37) = 24157817
fib(38) = 39088169
fib(39) = 63245986
fib(40) = 102334155
f(40) = 102334155
temps(s): 0.000119
```

Memorització genèrica amb decoradors

```
def memoritza (f):  
    mem = {}  
    def f2 (x):  
        if x not in mem:  
            mem[x] = f(x)  
        return mem[x]  
    return f2  
  
@testInterval(38, 40)  
@memoritza  
def fib(n):  
    if n in [0, 1]:  
        return n  
    return fib(n-1) + fib(n-2)
```

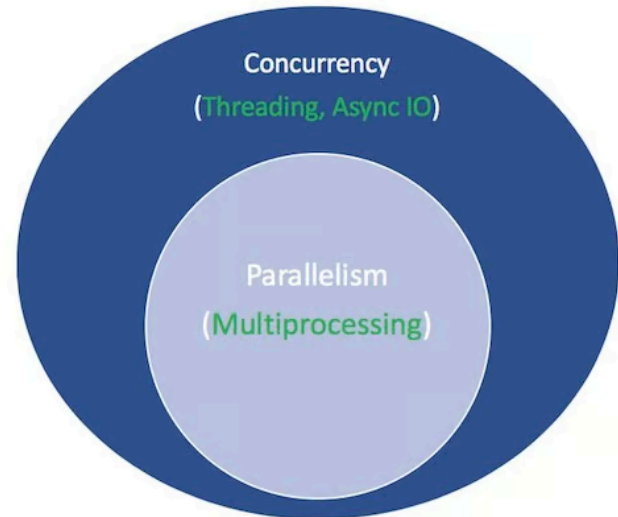
```
fib(38) = 39088169  
fib(39) = 63245986  
fib(38) = 39088169  
fib(40) = 102334155  
f(40) = 102334155  
temps(s): 0.000078
```


Contingut

- Recursivitat
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
- Programació asíncrona
 - Introducció
 - Exemples
 - Aplicacions

Introducció

- **Paral·lelisme i multiprocés:** programes que involucren diversos processadors al mateix temps. No és possible en python.
- **Concurrencia:** quan el sistema admet que hi hagi 2 o més tasques funcionant al mateix temps.
- Programació **asíncrona:** programació multitasca en front de la seqüencial (síncrona).



font: Real Python: Async IO in Python

- **Threading:** model asíncron clàssic molt útil per programació multitasca amb memòria compartida.
- **Async IO:** nou model de programació asíncrona alternatiu als *threads*, però que no el substitueix. **No soluciona els problemes de les *race conditions***

Model AsyncIO

Corutina: funció asíncrona: podem aturar-la i fer-la continuar de nou.

Hello World!

```
import asyncio

async def say_hello_async():
    await asyncio.sleep(3)
    print("Hola món!")

asyncio.run(say_hello_async())
```

Funcions de l'Async IO:

- **async def**: defineix la corutina.
- **await**: torna el control fins que s'acompleix la tasca encomanada.
- **asyncio.run**: crida.

Exemples d'ús:

- Mòbils: consulta d'urls.
- Chatbots de telegram.
- Motors de videojocs.

Contingut

- Recursivitat
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
- Programació asíncrona
 - Introducció
 - Exemples
 - Aplicacions

Gestió de diferents tasques

Codi:

```
import asyncio

async def task_one():
    print("Starting task one")
    await asyncio.sleep(1)
    print("Finishing task one")
    return 1

async def task_two():
    print("Starting task two")
    await asyncio.sleep(2)
    print("Finishing task two")
    return 2

async def main():
    # Wait for all the coroutines
    results = await
        asyncio.gather(task_one(),
                        task_two())
    print(results)

asyncio.run(main())
```

Sortida:

```
Starting task one
Starting task two
Finishing task one
Finishing task two
[1, 2]
```

font: **Hascker Culture: Python**
asyncio

Funcions:

asyncio.gather: crida a diverses corutines.

Espera i fallada

Codi:

```
import asyncio

async def might_fail():
    try:
        await asyncio.sleep(2)
        print("Success!")
    except asyncio.CancelledError:
        print("Operation cancelled")

async def main():
    task = asyncio.create_task(
        might_fail())

    try:
        await asyncio.wait_for(task,
                                timeout=1)
    except asyncio.TimeoutError:
        print("Operation timed out")
        task.cancel()
        await task

if __name__ == "__main__":
    asyncio.run(main())
```

Sortida:

Operation cancelled

font: Medium: Master asyncio in
Python

Funcions:

asyncio.wait_for

task.cancel

Obtenint URLs

Codi:

```
import aiohttp
import asyncio
import time

async def fetch_async(url, session):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        page1 = asyncio.create_task(fetch_async('http://example.com', session))
        page2 = asyncio.create_task(fetch_async('http://example.org', session))
        await asyncio.gather(page1, page2)

start_time = time.time()
asyncio.run(main())
print(f"Done in {time.time() - start_time} seconds")
```

font: Medium: Mastering Python's Asyncio

Contingut

- Recursivitat
- Orientació a Objectes
- Subtipus i variància de tipus
- Clausures
- Programació asíncrona
 - Introducció
 - Exemples
 - Aplicacions

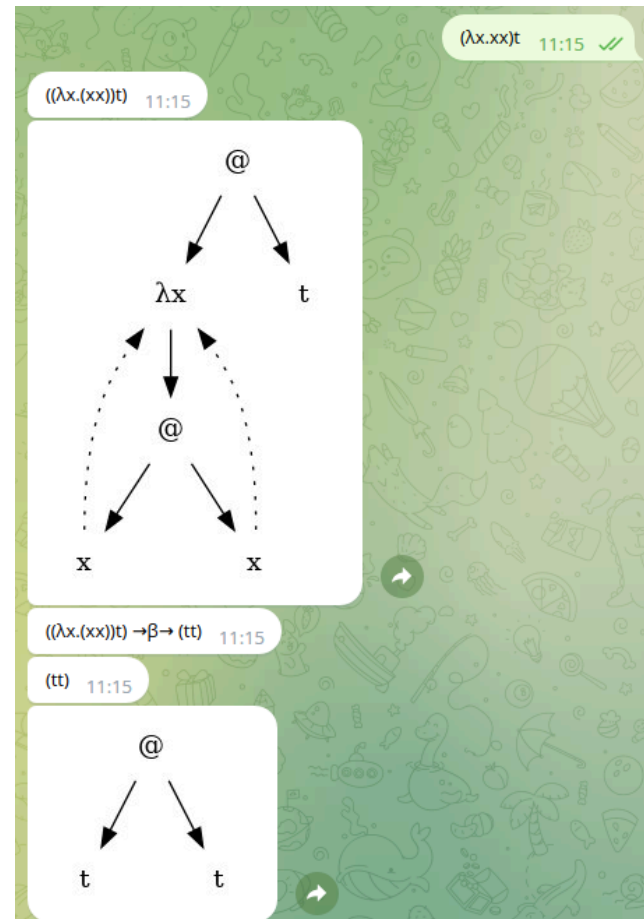
Telegram

Pràctica d'LP (primavera del 23): intèrpret de λ -càlcul.

```
async def start(update: Update,
                 context: ContextTypes
                 .DEFAULT_TYPE):
    context.user_data['visitor'] =
        EvalVisitor()
    user = update.effective_user
    msg = '''
AChurchBot!
Benvingut %s !
''' % user.first_name
    await update.message.reply_text(msg)
```

```
def main() -> None:
    TOKEN = open('token.txt').read()
    .strip()
    application = Application.builder()
    .token(TOKEN).build()
    application.add_handler(
        CommandHandler("start", start))
    ...
    application.run_polling()
```

python-telegram-bot funciona sobre *asyncio*.



Algorisme de Flocking

Algorisme asíncron per simular el moviment grupal d'animals com peixos, abelles, ocells... (Reynolds, 1999)

Ho veurem en C# i Unity (motor de videojocs) per apreciar millor l'efecte.

Algorisme bàsic:

- [Apunt de Flocking](#)
- [Vídeo demostració](#)

Implementació més complexa:

- Sebastian Lague. [Coding adventure: Boids](#), 2019.
- [Repositori github](#)