

École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

Cours 9 / 27 novembre 2008

dans le cours d'aujourd'hui, on se focalise sur les **langages à objets**

bref historique :

- Simula I et Simula 67 (années 60)
- Smalltalk (1980)
- C++ (1983)
- Java (1995)

on utilise ici le langage Java à des fins d'illustration

brève présentation des concepts objets

le premier concept objet est celui de **classe** ; la déclaration d'une classe introduit un nouveau type

en toute première approximation, une classe peut être vue comme un enregistrement

```
class Polar {  
    double rho ;  
    double theta ;  
}
```

ici rho et theta sont les deux **champs** de la classe Polar, de type double

Objet

on crée une **instance** particulière d'une classe, appelée un **objet**, avec la construction `new`; ainsi

```
Polar p = new Polar();
```

déclare une nouvelle variable locale `p`, de type `Polar`, dont la valeur est une nouvelle instance de la classe `Polar`

l'objet est alloué sur le tas; ses champs reçoivent ici des valeurs par défaut (en l'occurrence 0)

on peut accéder aux champs de `p`, et les modifier, avec la notation usuelle

```
p.rho = 2;  
p.theta = 3.14159265;  
double x = p.rho * Math.cos(p.theta);  
p.theta = p.theta / 2;
```

Constructeur

on peut introduire un ou plusieurs **constructeurs**, dans le but d'initialiser les champs de l'objet

```
class Polar {  
    double rho, theta;  
    Polar(double r, double t) {  
        if (r < 0) throw new Error("Polar : negative length");  
        rho = r;  
        theta = t;  
    }  
}
```

ce qui permet alors d'écrire

```
Polar p = new Polar(2, 3.14159265);
```

Encapsulation

supposons maintenant que l'on veuille maintenir l'**invariant** suivant pour tous les objets de la classe Polar

$$0 \leq \text{rho} \quad \wedge \quad 0 \leq \text{theta} < 2\pi$$

pour cela on déclare les champs rho et theta **privés**, de sorte qu'ils ne sont plus visibles à l'extérieur de la classe Polar

```
class Polar {  
    private double rho, theta;  
    Polar(double r, double t) { /* garantie l'invariant */ }  
}
```

```
p.rho = 1;
```

```
complex.java :19 : rho has private access in Polar
```

Méthode

la valeur du champ `rho` peut néanmoins être fournie par l'intermédiaire d'une **méthode**, c'est-à-dire d'une fonction fournie par la classe `Polar` et applicable à tout objet de cette classe

```
class Polar {  
    private double rho, theta;  
    ...  
    double norm() { return rho; }  
}
```

pour un objet `p` de type `Polar`, on appelle la méthode `norm` ainsi

`p.norm()`

que l'on peut voir naïvement comme l'appel `norm(p)` d'une fonction

```
double norm(Polar x) { return x.rho; }
```

les objets remplissent donc un premier rôle d'**encapsulation**

l'équivalent en Caml serait obtenu grâce aux types abstraits

Champs et méthodes statiques

il est possible de déclarer un champ comme **statique** et il est alors lié à la classe et non aux instances de cette classe ; dit autrement, il s'apparente à une variable globale, dont la valeur est unique

```
class Polar {  
    double rho, theta ;  
    static double two_pi = 6.283185307179586 ;  
}
```

de même, une **méthode** peut être **statique** et elle s'apparente alors à une fonction traditionnelle

```
static double normalize(double x) {  
    while (x < 0) x += two_pi ;  
    while (x >= two_pi) x -= two_pi ;  
    return x ;  
}
```

ce qui n'est pas statique est appelé **dynamique**

le second concept objet est celui d'**héritage** : une classe B peut être définie comme héritant d'une classe A

```
class B extends A { ... }
```

les objets de la classe B héritent alors de tous les champs et méthodes de la classe A, auxquels ils peuvent ajouter de nouveaux champs et de nouvelles méthodes

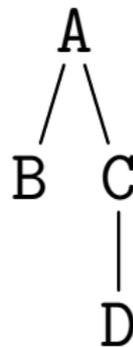
la notion d'héritage s'accompagne d'une notion de **sous-typage** : toute valeur de type B peut être vue comme une valeur de type A

en Java, chaque classe hérite d'au plus une classe ; on appelle cela l'**héritage simple**, par opposition à l'héritage multiple

Relation d'héritage

la relation d'héritage forme un arbre

```
class A { ... }  
class B extends A { ... }  
class C extends A { ... }  
class D extends C { ... }
```



Exemple

considérons une classe `Graphical` pour des objets graphiques (cercles, rectangles, etc.)

```
class Graphical {  
    int x, y;           /* centre */  
    int width, height;  
  
    void move(int dx, int dy) { x += dx; y += dy; }  
    void draw() { /* ne fait rien */ }  
}
```

Exemple

pour représenter un rectangle, on hérite de la classe Graphical

```
class Rectangle extends Graphical {
```

on hérite donc des champs `x`, `y`, `width` et `height` et des méthodes `move` et `draw`

on peut écrire un constructeur qui prend en arguments deux coins du rectangle

```
Rectangle(int x1, int y1, int x2, int y2) {  
    x = (x1+x2)/2;  
    y = (y1+y2)/2;  
    width = Math.abs(x1-x2);  
    height = Math.abs(y1-y2);  
}
```

Redéfinition (*overwriting*)

on peut utiliser directement toute méthode héritée de Graphical

```
Rectangle p = new Rectangle(0, 0, 100, 50);  
p.move(10, 5);
```

pour le dessin, en revanche, on va **redéfinir** la méthode draw dans la classe Rectangle (*overwriting*)

```
class Rectangle extends Graphical {  
    ...  
    void draw() { /* dessine le rectangle */ }  
}
```

et le rectangle sera alors effectivement dessiné quand on appelle

```
p.draw();
```

Ajout de champs

on procède de même pour les cercles ; ici on ajoute un champ radius pour le rayon, afin de le conserver

```
class Circle extends Graphical {
    int radius ;
    Circle(int cx, int cy, int r) {
        x = cx ;
        y = cy ;
        radius = r ;
        width = height = 2 * radius ;
    }
    void draw() { /* dessine le cercle */ }
}
```

Type statique et type dynamique

la construction `new C(...)` construit un objet de classe `C`, et la classe de cet objet ne peut être modifiée par la suite ; on l'appelle le **type dynamique** de l'objet

en revanche, le **type statique** d'une expression, tel qu'il est calculé par le compilateur, peut être différent du type dynamique, du fait de la relation de sous-typage introduite par l'héritage

exemple

```
Graphical g = new Rectangle(0, 0, 100, 50) ;  
g.draw() ; // dessine le rectangle
```

pour le compilateur, `g` a le type `Graphical`, mais le rectangle est effectivement dessiné : c'est donc bien la méthode `draw` de la classe `Rectangle` qui est exécutée

Des groupes

introduisons enfin un troisième type d'objet graphique, qui est simplement la réunion de plusieurs objets graphiques

on commence par introduire des listes chaînées de `Graphical`

```
class GList {
    Graphical g;
    GList next;
    GList(Graphical g, GList next) {
        this.g = g;
        this.next = next;
    }
}
```

(`this` désigne l'objet dont on appelle la méthode; il est utilisé ici pour distinguer le paramètre formel `g` du champ `g` de même nom)

un groupe hérite de `Graphical` et contient une `GList`

```
class Group extends Graphical {
    GList group;

    Group() { group = null; }

    void add(Graphical g) {
        group = new GList(g, group);
        // + mise à jour de x,y,width,height
    }
}
```

il reste à redéfinir les méthodes draw et move

```
void draw() {
    for(GList l = group; l != null; l = l.next)
        l.g.draw();
}

void move(int dx, int dy) {
    x += dx; y += dy;
    for(GList l = group; l != null; l = l.next)
        l.g.move(dx, dy);
}
}
```

note : il est clair sur cet exemple que le compilateur ne peut inférer le type dynamique de `l.g`

Surcharge

en Java, plusieurs méthodes d'une même classe peuvent porter le même nom, pourvu qu'elles aient des arguments en nombre et/ou en nature différents ; c'est ce que l'on appelle la **surcharge** (*overloading*)

```
class Polar {  
    void mult(Polar p) {  
        rho *= p.rho; theta = normalize(theta + p.theta);  
    }  
    void mult(double f) { rho *= f; }
```

il en va de même pour les constructeurs

```
class Rectangle extends Graphical {  
    Rectangle(int x1, int y1, int x2, int y2) { ... }  
    Rectangle(int x1, int y1, int w) { /* construit un carré */ }  
    ...
```

Classe abstraite

remarque : il n'y a pas lieu de créer d'instance de la classe `Graphical` ; c'est ce que l'on appelle une **classe abstraite**

certaines méthodes, comme `draw`, peuvent alors n'être définies que dans les sous-classes

```
abstract class Graphical {  
    int x, y;  
    int width, height;  
  
    void move(int dx, int dy) { x += dx; y += dy; }  
    abstract void draw();  
}
```

il est alors obligatoire de définir `draw` dans toute sous-classe (non abstraite) de `Graphical`

la notion de classe remplit plusieurs fonctions

- **encapsulation**, à travers des règles de visibilité
- organisation de l'**espace de noms**, à travers la possibilité d'utiliser le même nom dans des classes différentes ou pour des profils différents
- **factorisation de code**, à travers l'héritage et la redéfinition

ce sont des objectifs essentiels du **génie logiciel**, atteints par des moyens différents dans d'autres langages (exemple : polymorphisme, ordre supérieur, ou encore système de modules, dans le cas d'Ocaml)

Brève comparaison fonctionnel / objet

qu'est-ce qui distingue

```
type graphical = Circle of ... | Rectangle of ...
```

```
class Graphical {...} class Circle extends Graphical {...}
```

en Ocaml, le code de move est à un seul endroit et traite tous les cas

```
let move = function Circle _ -> ... | Rectangle _ -> ...
```

en Java, il est éclaté dans l'ensemble des classes

conséquence :

	extension horizontale = ajout d'un cas	extension verticale = ajout d'une fonction
Java	facile (un seul fichier)	pénible (plusieurs fichiers)
Ocaml	pénible (plusieurs fichiers)	facile (un seul fichier)

compilation des langages à objets

Représentation des objets

un objet est un bloc alloué sur le tas, contenant

- sa classe
- les valeurs de ses champs (comme pour un enregistrement)

```
class Graphical {
    int x, y, width, height; ... }
class Rectangle extends Graphical {
    ... }
class Circle extends Graphical {
    int radius; ... }

new Rectangle(0, 0, 100, 50)
new Circle(20, 20, 10)
```

Rectangle
x
y
width
height

Circle
x
y
width
height
radius

la valeur d'un objet est le pointeur vers le bloc

Représentation des objets

on note que l'héritage simple permet de stocker la valeur d'un champ x à un emplacement constant dans le bloc ; les champs propres viennent après les champs hérités

le calcul de la valeur droite ou gauche de $e.x$ est donc aisé

exemple : on compile $e.width$ par

```
# on compile la valeur de e dans $t0  
lw  $t1, 12($t0)
```

on compile $e.width = e'$ par

```
# on compile la valeur de e dans $t0  
# on compile la valeur de e' dans $t1  
sw  $t1, 12($t0)
```

Appel de méthode

en Java, le mode de passage est **par valeur** (mais la valeur d'un objet n'est qu'un pointeur sur le tas)

une méthode statique est compilée de manière traditionnelle

que ce soit pour les constructeurs, les méthodes statiques ou dynamiques, la surcharge est résolue à la compilation, et des noms distincts sont donnés aux différents constructeurs et méthodes

```
class A {  
    A() {...}  
    A(int x) {...}  
  
    void m() {...}  
    void m(A a) {...}  
    void m(A a, A b) {...}
```

```
class A {  
    A() {...}  
    A_int(int x) {...}  
  
    void m() {...}  
    void m_A(A a) {...}  
    void m_A_A(A a, A b) {...}
```

la surcharge est cependant délicate

```
class A {  
    void m(B b) {...}  
}  
class B extends A {  
    void m(A a) {...}  
}
```

la classe B possède deux méthodes

```
void m(B b); // héritée de A  
void m(A a);
```

```
{ ... B b = new B(); b.m(b); ... }
```

la première est appelée, car **plus précise** du point de vue de l'argument

il peut y avoir ambiguïté

```
class A {...}
class B extends A {
    void m(A a, B b) {...}
    void m(B b, A a) {...}
}
{ ... B b = new B(); b.m(b,b); ... }
```

surcharge1.java :13 : reference to m is ambiguous,
both method m(A,B) in B and method m(B,A) in B match

Algorithme de résolution de la surcharge

à chaque méthode définie dans la classe C

$$\tau \text{ m}(\tau_1 \times_1, \dots, \tau_n \times_n)$$

on associe le profil $(C, \tau_1, \dots, \tau_n)$

on **ordonne** les profils : $(\tau_0, \tau_1, \dots, \tau_n) \sqsubseteq (\tau'_0, \tau'_1, \dots, \tau'_n)$ si et seulement si τ_i est un sous-type de τ'_i pour tout i

pour un appel

$$e.m(e_1, \dots, e_n)$$

où e a le type statique τ et e_i le type statique τ_i , on considère l'ensemble des éléments **minimaux** dans l'ensemble des profils compatibles

- aucun élément \Rightarrow aucune méthode ne s'applique
- plusieurs éléments \Rightarrow ambiguïté
- un unique élément \Rightarrow c'est la méthode à appeler

toute la subtilité de la compilation des langages à objets est dans l'**appel d'une méthode dynamique** $e.m(e_1, \dots, e_n)$

pour cela, on construit pour chaque classe un **descripteur de classe** qui contient les adresses des codes de méthodes dynamiques de cette classe

comme pour les champs, l'héritage simple permet de ranger l'adresse du code de la méthode m à un emplacement constant dans le descripteur

les descripteurs de classes peuvent être construits dans le segment de données ; chaque objet contient dans son premier champ un pointeur vers le descripteur de sa classe

Exemple

```
class A          { void f() {...} }  
class B extends A { void g() {...} }  
class C extends B { void g() {...} }  
class D extends C { void f() {...}  
                  void h() {...} }
```

descr. A

A_f

descr. B

A_f
B_g

descr. C

A_f
C_g

descr. D

D_f
C_g
D_h

en pratique, le descripteur de la classe C contient également l'indication de la classe dont C hérite, appelée **super classe** de C

la super classe est représentée par un pointeur vers son descripteur

on peut le ranger dans le premier champ du descripteur

Exemple complet

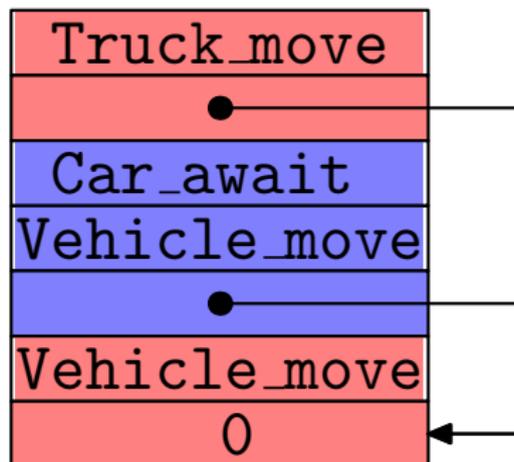
```
class Vehicle {
    static int start = 10;
    int position;
    Vehicle() { position = start; }
    void move(int d) { position += d; } }

class Car extends Vehicle {
    int passengers;
    Car() { super(); }
    void await(Vehicle v) {
        if (v.position < position)
            v.move(position - v.position);
        else
            move(10); } }

class Truck extends Vehicle {
    int load;
    Truck() { super(); }
    void move(int d) {
        if (d <= 55) position += d; else position += 55; } }
```

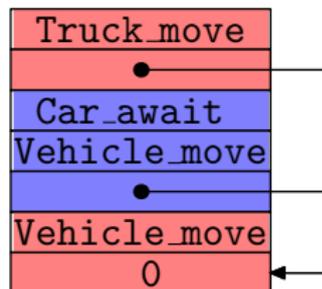
Descripteurs de classes

on construit les descripteurs suivants dans le segment de données



Descripteurs de classes

```
.data
descr_Vehicle:
    .word    0
    .word    Vehicle_move
descr_Car:
    .word    descr_Vehicle
    .word    Vehicle_move
    .word    Car_await
descr_Truck:
    .word    descr_Vehicle
    .word    Truck_move
```



et le champ statique de `Vehicle` est également dans le segment de données

```
static_start:
    .word    10
```

Constructeurs

le code d'un constructeur est une fonction qui suppose l'objet déjà alloué et son adresse dans \$a0, le premier champ déjà rempli (descripteur de classe) et les arguments du constructeur dans \$a1, \$a2, \$a3

```
class Vehicle {  
    Vehicle() { position = start; }  
}
```

new_Vehicle:

```
lw    $t0, static_start  
sw    $t0, 4($a0)  
jr    $ra
```

pour Car, le constructeur se contente d'appeler celui de la super classe, à savoir Vehicle

```
class Car extends Vehicle {  
    Car() { super(); }  
}
```

on reconnaît là un appel terminal, ce qui donne tout simplement

```
new_Car:  
    j          new_Vehicle
```

de même pour le constructeur de Truck

pour les méthodes, on adopte la même convention : l'objet est dans \$a0 et les arguments de la méthode dans \$a1, \$a2, \$a3

```
class Vehicle {  
    void move(int d) { position += d; }  
}
```

```
Vehicle_move:  
    lw     $t0, 4($a0)  
    add   $t0, $t0, $a1  
    sw     $t0, 4($a0)  
    jr    $ra
```

(de même pour la méthode move de Truck)

plus intéressant est

```
class Car extends Vehicle {  
    void await(Vehicle v) {  
        if (v.position < position)  
            v.move(position - v.position);  
        else  
            move(10);  
    }  
}
```

(`jr` et non `jalr` car l'appel est terminal)

Car.await:

```
lw    $t0, 4($a1)  
lw    $t1, 4($a0)  
bge   $t0, $t1, L1  
move  $a0, $a1  
sub   $a1, $t1, $t0  
lw    $t0, 0($a0)  
lw    $t0, 4($t0)  
jr    $t0
```

L1:

```
li    $a1, 10  
lw    $t0, 0($a0)  
lw    $t0, 4($t0)  
jr    $t0
```

Programme principal

```
class Main {
    public static void main(String arg[]) {
        Truck t = new Truck();
        Car c = new Car();
        c.passengers = 2;
        Vehicle v = c;
        System.out.println(c.position);
        c.move(60);
        System.out.println(c.position);
        v.move(70);
        System.out.println(c.position);
        c.await(t);
        System.out.println(t.position);
        System.out.println(c.position);
    }
}
```

Création d'un objet

```
Truck t = new Truck();
```

on commence par allouer un bloc de
12 octets sur le tas

on stocke le descripteur de Truck
dans le premier champ

on décide de mettre t dans \$s1

on appelle le code du constructeur

```
li    $a0, 12
```

```
li    $v0, 9
```

```
syscall
```

```
la    $t0, descr_Truck
```

```
sw    $t0, 0($v0)
```

```
move  $s1, $v0
```

```
move  $a0, $v0
```

```
jal   new_Truck
```

(de même pour `c = new Car()`, stockée dans \$s2)

la déclaration de variable

```
Vehicle v = c ;
```

ne fait que créer un alias (un autre nom pour le même objet)

si v est stockée dans $\$s3$, le code produit est simplement

```
move    $s3, $s2
```

(on pourrait même faire l'économie de cette instruction)

Appel de méthode

l'appel

```
c.move(60);
```

est compilé en

```
move    $a0, $s2  
li      $a1, 60  
lw      $t0, 0($s2)  
lw      $t0, 4($t0)  
jalr    $t0
```

on se sert enfin de `jalr` !

etc.

Optimisation de l'appel

pour plus d'efficacité, on peut chercher à remplacer un appel dynamique (*i.e.* calculé pendant l'exécution) par un appel statique (*i.e.* connu à la compilation)

pour un appel $e.m(\dots)$, et e de type statique C , c'est notamment possible lorsque la méthode m n'est redéfinie dans aucune sous-classe de C

une autre possibilité, plus complexe, consiste à propager les types connus à la compilation (*type propagation*)

```
B b = new B();    // la classe de b est B
A a = b;         // la classe de a est B
a.m();          // on sait exactement de quelle
                // méthode il s'agit
```

comme on l'a vu, le type statique et le type dynamique d'une expression désignant un objet peuvent différer (à cause du sous-typage)

il est parfois nécessaire de « forcer la main » au compilateur, en prétendant qu'un objet e appartient à une certaine classe C , ou plus exactement à l'une des super classes de C ; on appelle cela le **transtypage** (*cast*)

la notation de Java est

$$(C)e$$

le type statique d'une telle expression est C

considérons l'expression

$$(C)e$$

soit

- D le type statique de l'expression e
- E le type dynamique de (l'objet désigné par) e

il y a trois situations

- C est une super classe de D : on parle d'**upcast** et le code produit pour $(C)e$ est le même que pour e (mais le *cast* a une influence sur le typage puisque le type de $(C)e$ est C)
- C est une sous-classe de D : on parle de **downcast** et le code contient un **test dynamique** pour vérifier que E est bien une sous-classe de C
- C n'est ni une sous-classe ni une super classe de D : le compilateur refuse l'expression

Exemple

```
void m(Vehicule v, Vehicule w) {  
    ((Car)v).await(w);  
}
```

rien ne garantit que l'objet passé à `m` sera bien une voiture ; en particulier il pourrait ne même pas posséder de méthode `await` !

le test dynamique est donc nécessaire

Tester l'appartenance à une classe

pour permettre une programmation un peu plus défensive, il existe une construction booléenne

`e instanceof C`

qui détermine si la classe de `e` est bien une sous-classe de `C`

on trouve souvent le schéma

```
if (e instanceof C) {  
    C c = (C)e;  
    ...  
}
```

dans ce cas, le compilateur effectue typiquement une optimisation consistant à ne pas générer de test pour le *cast*

Compilation de instanceof

compilons la construction

`e instanceof C`

en supposant la valeur de `e` dans `$a0` et le descripteur de `C` dans `$a1`

```
L:    lw      $a0, 0($a0)
      beq    $a0, $a1, Ltrue
      lw      $a0, 0($a0)
      beq    $a0, 0, Lfalse
      b      L
```

le compilateur peut optimiser les constructions $(C)e$ et $e \text{ instanceof } C$ dans certains cas

- si C est l'unique sous-classe de D alors un unique test d'égalité plutôt qu'une boucle
- si D est une sous-classe de C alors $e \text{ instanceof } C$ vaut true

une autre optimisation est possible si l'ensemble des classes est connu à la compilation ; soit n la profondeur maximale dans la hiérarchie des classes

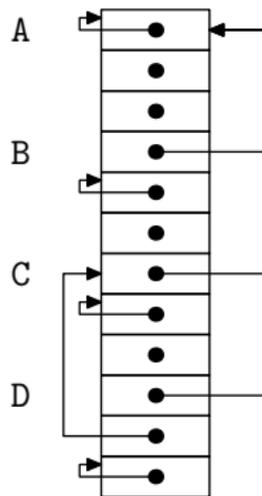
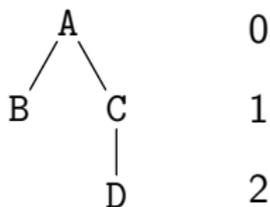
le descripteur d'une classe C de profondeur k contient un tableau de taille n où les cases $0..k$ contiennent les pointeurs vers les descripteurs des super classes de C ; les autres cases contiennent le pointeur nul

pour tester si x est une instance de D on considère la profondeur j de D (connue statiquement) et on regarde dans la case j du descripteur de x s'il y a un pointeur vers le descripteur de D

Exemple

```
class A {...}  
class B extends A {...}
```

```
class C extends A {...}  
class D extends C {...}
```



supposons que l'on veuille ajouter de l'**héritage multiple** à Java

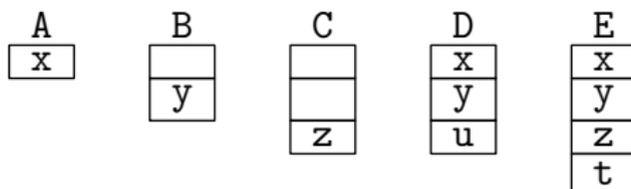
on ne peut plus utiliser le principe que

- la représentation d'un objet d'une super classe de C est un préfixe de la représentation d'un objet de la classe C
- de même pour les descripteurs de classes

Exemple

```
class A { int x; ... }  
class B { int y; ... }  
class C { int z; ... }  
class D extends A,B { int u; ... }  
class E extends A,B,C { int t; ... }
```

une représentation possible des objets est



problèmes :

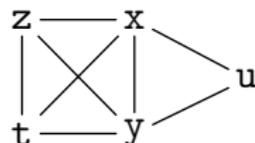
- il faut déterminer ces emplacements
- la représentation n'est pas compacte

Interférence

supposons que l'on connaît l'ensemble des classes au moment de la compilation

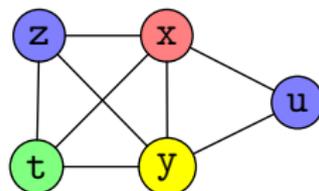
alors on peut construire un **graphe d'interférence**, dans lequel les champs f et g sont reliés dès lors que f et g sont deux champs de la même classe (directement ou par l'intermédiaire de l'héritage)

```
class A { int x; ... }
class B { int y; ... }
class C { int z; ... }
class D extends A,B { int u; ... }
class E extends A,B,C { int t; ... }
```

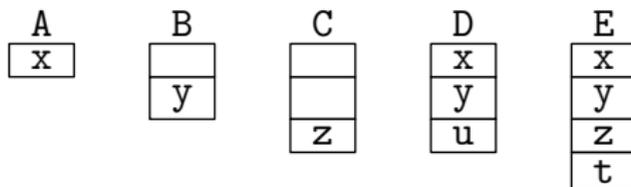


on cherche alors à affecter un nombre minimal d'emplacements aux sommets de ce graphe en respectant les interférences \Rightarrow c'est un problème de **coloriage de graphe**

ici il est possible de 4-colorier ce graphe



ce qui correspond (entre autres) à ce choix de représentation

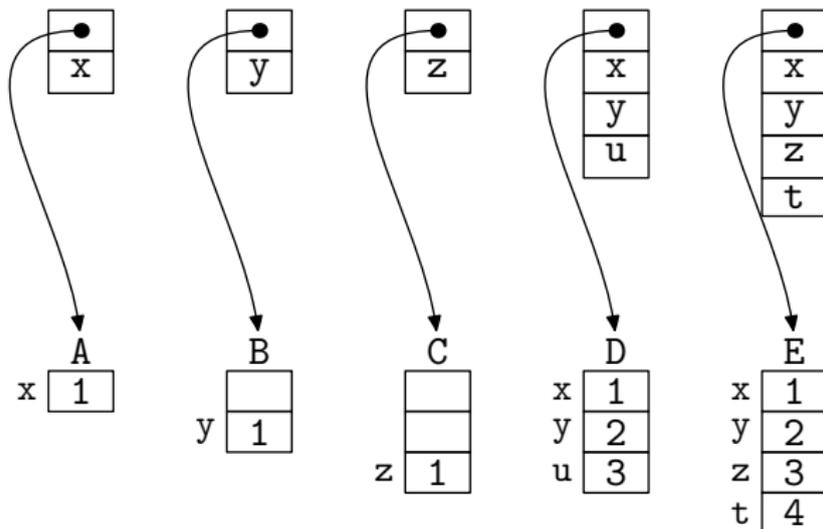


la même idée peut être utilisée pour l'appel de méthode

Représentation compacte des objets

on souhaite cependant une représentation compacte des objets

pour cela, on ajoute une indirection : ce sont les descripteurs de classes qui utilisent les emplacements disjoints, pour indiquer les emplacements réels qui sont maintenant contigus



en pratique, connaître l'ensemble des classes au moment de la compilation n'est pas réaliste, car on veut

- la compilation séparée
- voire même le chargement dynamique de classes

on ne peut plus calculer les emplacements à la compilation \Rightarrow chaque descripteur de classe contient une table (de hachage) donnant le déplacement pour chaque champ / méthode

(cf Appel chapitre 14)

la semaine prochaine

- TD 9 le mercredi 3 décembre
 - aide au projet
- Cours 10 le jeudi 4 décembre
 - production de code efficace