École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

compilation des langages objets

dans le cours d'aujourd'hui, on se focalise sur les **langages 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
(mais quelques mots sur C++ à la fin du cours)

plan

1. les concepts objets
2. leur compilation

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`
(on parle aussi parfois d'**attributs**)

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 recoivent 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;
```

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);
```

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

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

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) { /* garantit l'invariant */ }
}
```

depuis une autre classe :

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

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

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 OCaml 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

```
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 { ... }
```

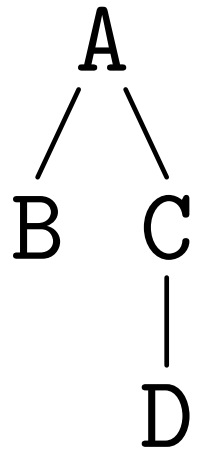
les objets de la classe `B` héritent alors de tous les champs et méthodes de la classe `A`, auxquels ils peuvent ajouter de nouveaux champs et de nouvelles méthodes

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

la relation d'héritage forme donc un arbre

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



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 */ }
}
```

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 (*overriding*)

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** (*overriding*) la méthode `draw` dans la classe Rectangle

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

et le rectangle sera alors effectivement dessiné quand on appelle

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

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

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 / le constructeur ; 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 {
  private 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) {
    super.move(dx, dy);
    for (GList l = group; l != null; l = l.next)
      l.g.move(dx, dy);
  }
}
```

il est clair sur cet exemple que le compilateur
ne peut pas connaître le type dynamique de `l.g`

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`

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 Rectangle extends Graphical {
  ...
  void draw() {
    ...
  }
  void draw(String color) {
    ...
  }
}
```

puis

```
  r.draw() ... r.draw("red") ...
```

la surcharge est résolue au typage

tout se passe comme si on avait écrit

```
class Rectangle extends Graphical {
  ...
  void draw() {
    ...
  }
  void draw_String(String color) {
    ...
  }
}
```

puis

```
  r.draw() ... r.draw_String("red") ...
```

on peut également surcharger les constructeurs

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

programme principal

```
class Main {
  public static void main(String arg[]) {
    Rectangle g1 = new Rectangle(0, 0, 100, 50);
    g1.move(10, 5);
    g1.draw();
    Graphical g2 = new Circle(10,10,2);
    Group g3 = new Group(); g3.add(g1); g3.add(g2);
    g3.draw();
    g3.move(-5,-7);
    g3.draw();
  }
}
```

```
$ javac *.java
$ java Main
```

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)

qu'est-ce qui distingue le code Java

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

du code OCaml

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

?

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

un point commun : une information **dynamique** (constructeur OCaml / classe Java) permet de déterminer le code à exécuter (filtrage OCaml / appel de méthode Java)

brève comparaison fonctionnel / objet

	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 objets

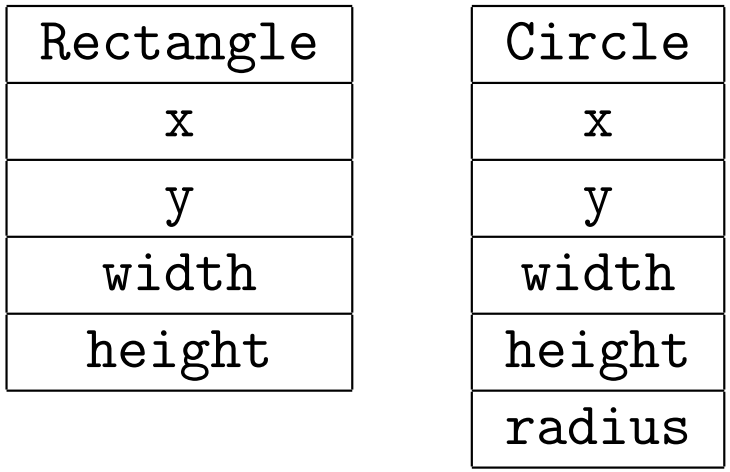
le langage Java est traditionnellement compilé vers une machine virtuelle, la JVM (*Java Virtual Machine*)

ici, cependant, on illustre une compilation de Java vers x86-64, qui pourrait s'appliquer à d'autres langages 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)
```



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

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 %rcx, puis
    movq 24(%rcx), %rax
```

on compile `e.width` = e' par

```
    # on compile la valeur de e dans %rcx
    # on compile la valeur de e' dans %rax
    movq %rax, 24(%rcx)
```

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) {...}
```

résoudre la surcharge peut être délicat

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

dans le code

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

les deux méthodes s'appliquent potentiellement

c'est la méthode `m(B b)` qui 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`

$$\tau\ \mathtt{m}(\tau_1\ x_1, ..., \tau_n\ x_n)$$

on associe le profil $(\mathtt{C}, \tau_1, \ldots, \tau_n)$

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

pour un appel

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

où e a le type statique τ_0 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

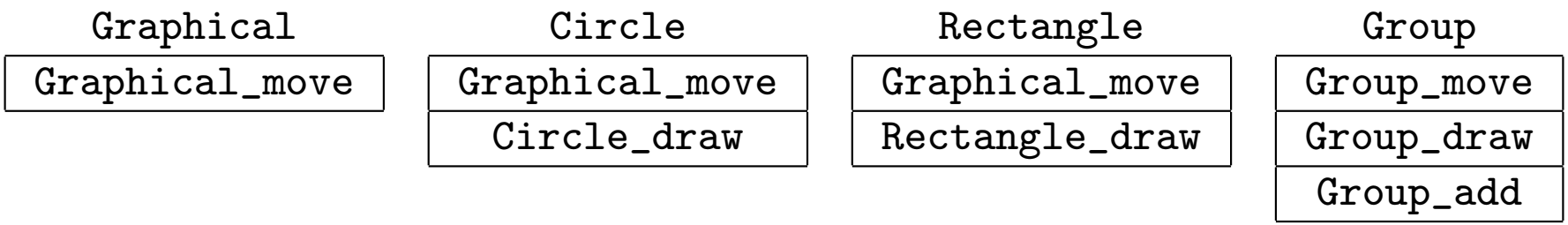
toute la subtilité de la compilation des langages objets est dans l'**appel d'une méthode dynamique** $e.m(e_1, \ldots, e_n)$

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

```
class Graphical                          { move... draw...          }
class Circle    extends Graphical { move... draw...          }
class Rectangle extends Graphical { move... draw...          }
class Group     extends Graphical { move... draw... add... }
```

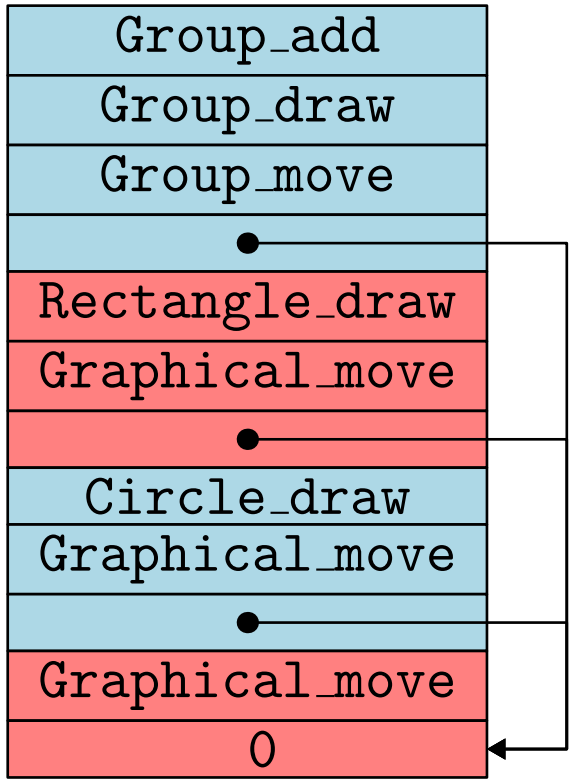


en pratique, le descripteur de la classe C contient également l'indication de la classe dont C hérite, appelée **super classe** de C

la super classe est représentée par un pointeur vers son descripteur

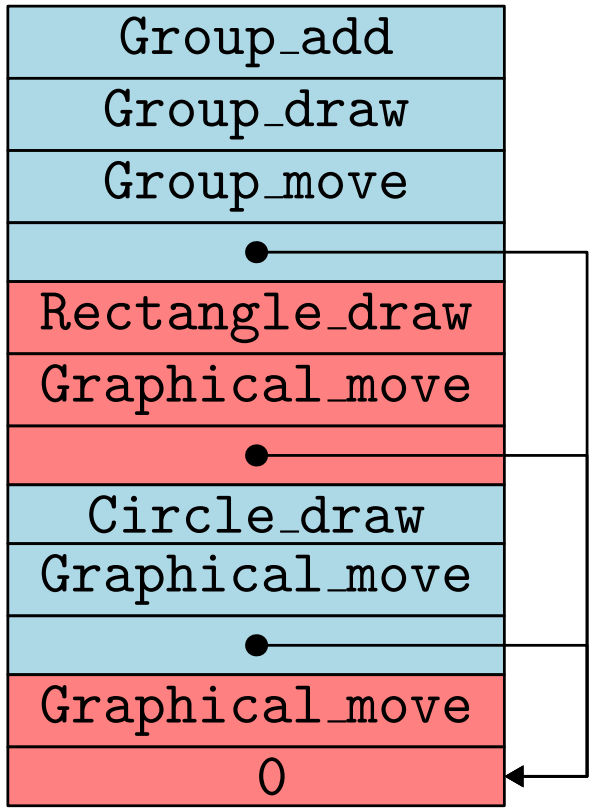
on peut le ranger dans le premier champ du descripteur

on construit les descripteurs suivants dans le segment de données



descripteurs de classes

```
        .data
descr_Graphical:
        .quad 0
        .quad Graphical_move
descr_Circle:
        .quad descr_Graphical
        .quad Graphical_move
        .quad Circle_draw
descr_Rectangle:
        .quad descr_Graphical
        .quad Graphical_move
        .quad Rectangle_draw
descr_Group:
        .quad descr_Graphical
        .quad Group_move
        .quad Group_draw
        .quad Group_add
```



le code d'un constructeur est une fonction qui suppose l'objet déjà alloué et son adresse dans `%rdi`, le premier champ déjà rempli (descripteur de classe) et les arguments du constructeur dans `%rsi`, `%rdx`, etc., et la pile

exemple :

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

```
new_GList:
        movq %rsi, 8(%rdi)
        movq %rdx, 16(%rdi)
        ret
```

en réalité, le constructeur commence par appeler le constructeur de la super classe

on peut le rendre explicite

```
class Group {
  Group() {
    super();       // était implicite
    group = null;
  }
}
```

notamment pour passer des arguments ou appeler un autre constructeur de la même classe (avec alors la syntaxe `this(...);`)

```
Rectangle g1 = new Rectangle(0, 0, 100, 50);
```

on commence par allouer un bloc de 40 octets sur le tas

```
movq $40, %rdi
call malloc
movq %rax, %r12
```

on stocke le descripteur de `Rectangle` dans le premier champ

```
movq $descr_Rectangle,(%r12)
```

on appelle le code du constructeur

```
movq %r12, %rdi
movq $0, %rsi
movq $0, %rdx
movq $100, %rcx
movq $50, %r8
call new_Rectangle
```

pour les méthodes, on adopte la même convention : l'objet est dans `%rdi` et les arguments de la méthode dans `%rsi`, `%rdx`, etc., et la pile si besoin

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

```
Graphical_move:
        addq %rsi, 8(%rdi)      # x += dx
        addq %rdx, 16(%rdi)     # y += dy
        ret
```

l'appel

```
g.move(10, 5);
```

est compilé en

```
... valeur de g dans %rdi
movq $10, %rsi
movq $5, %rdx
movq (%rdi), %rcx    # descripteur dans %rcx
call *8(%rcx)        # move est rangée en +8
```

c'est un saut à une adresse calculée
(comme la semaine dernière pour les fermetures)

autre exemple

```
class Group extends Graphical {
  void draw() {
    GList l = group;
    while (l != null) {
      l.g.draw();
      l = l.next;
    }
  }
}
```

```
Group_draw:
    pushq %rbx
    movq  8(%rdi), %rbx
    jmp   2f
1:  movq  8(%rbx), %rdi
    movq  (%rdi), %rcx
    call  *16(%rcx)
    movq  16(%rbx), %rbx
2:  testq %rbx, %rbx
    jnz   1b
    popq  %rbx
    ret
```

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

$$(C)e$$

il y a trois types en jeu

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

le compilateur ne connaît que C et D

il y a alors 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 (upcast)

```
class A {
  int x = 1;
}

class B extends A {
  int x = 2;
}
```

```
  B b = new B();
  System.out.println(b.x);       // 2
  System.out.println(((A)b).x); // 1
```

exemple (downcast)

```
void m(Graphical g, Graphical x) {
  ((Group)g).add(x);
}
```

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

le test dynamique est donc nécessaire

l'exception `ClassCastException` est levée si le test échoue

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

$$e \texttt{ instanceof } C$$

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 second test pour le *cast*

compilons la construction

$$e \texttt{ instanceof } C$$

en supposant la valeur de e dans %rdi et le descripteur de C dans %rsi

```
instanceof:
        movq  (%rdi), %rdi
1:      cmpq  %rdi, %rsi        # même descripteur ?
        je    Ltrue
        movq  (%rdi), %rdi      # on passe à la super classe
        testq %rdi, %rdi
        jnz   1b                # on a atteint Object ?
Lfalse: movq  $0, %rax
        ret
Ltrue:  movq  $1, %rax
        ret
```

le compilateur peut optimiser les constructions $(C)e$ et e `instanceof` C dans certains cas

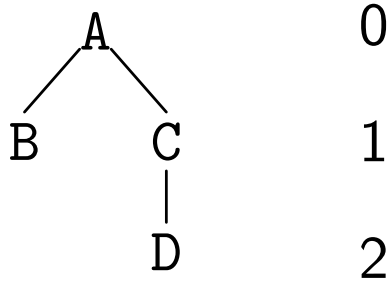
- si C est l'unique sous-classe de D alors un unique test d'égalité plutôt qu'une boucle
- si D est une sous-classe de C alors e `instanceof` C vaut `true`

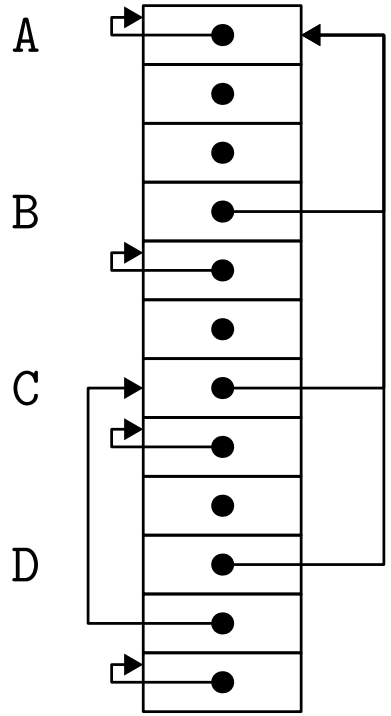
une autre optimisation est possible si l'ensemble des classes est connu à la compilation ; soit n la profondeur maximale dans la hiérarchie des classes

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

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





quelques mots sur C++

on reprend le même exemple

```
class Graphical {
public:
  int x, y;
  int width, height;

  virtual void move(int dx, int dy) { x += dx; y += dy; }
  virtual void draw() = 0;
};
```

`virtual` signifie que la méthode pourra être redéfinie

la présence d'une méthode non définie (`= 0`) implique une classe abstraite

```
class Circle : public Graphical {
public:
  int radius;
  Circle(int cx, int cy, int r) {
    x = cx; y = cy; radius = r;
    width = height = 2 * radius;
  }
  void draw() {
    ...
  }
};
```

(idem pour Rectangle)

```
class GList {
public:
  Graphical* g;
  GList* next;
  GList(Graphical* g, GList* next) {
    this->g    = g;
    this->next = next;
  }
};
```

noter que les pointeurs sont maintenant explicites

```
class Group : public Graphical {
  GList* group; // privé
public:
  Group() { group = NULL; }
  void move(int dx, int dy) {
    Graphical::move(dx, dy);
    for (GList* l = group; l != NULL; l = l->next)
      l->g->move(dx, dy);
  }
  void draw() {
    ...
  }
  void add(Graphical* g) {
    group = new GList(g, group);  // alloué sur le tas
    ...
  }
};
```

```
#include <iostream>
using namespace std;

int main() {
  Rectangle g1(0, 0, 100, 50);
  g1.move(10, 5);
  g1.draw();
  Circle g2(10, 10, 2);
  Group g3; // appelle le constructeur Group()
  g3.add(&g1); g3.add(&g2);
  g3.draw();
  g3.move(-5,-7);
  g3.draw();
}
```

les objets g1, g2 et g3 sont ici alloués **sur la pile**

sur cet exemple, la représentation d'un objet n'est pas différente de Java

descr. Circle
x
y
width
height
radius

descr. Rectangle
x
y
width
height

descr. Group
x
y
width
height
group

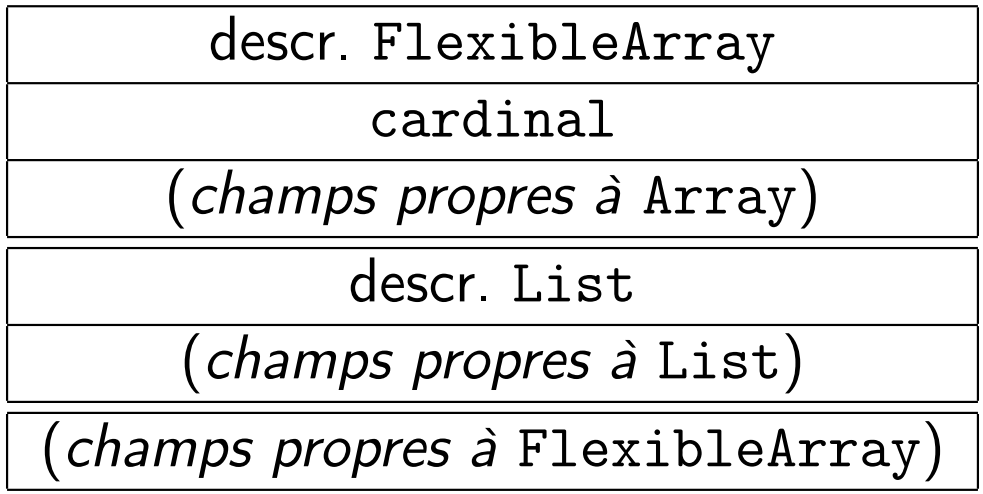
mais en C++, on trouve aussi de l'**héritage multiple**

conséquence : on ne peut plus (toujours) utiliser le principe selon lequel

- 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

```
class Collection {
  int cardinal;
};
class Array : public Collection {
  int nth(int i) ...
};
class List {
  void push(int v) ...
  int pop() ...
};
class FlexibleArray : public Array, public List {
};
```

dans un objet de la classe `FlexibleArray`, les représentations de `Array` et `List` sont juxtaposées

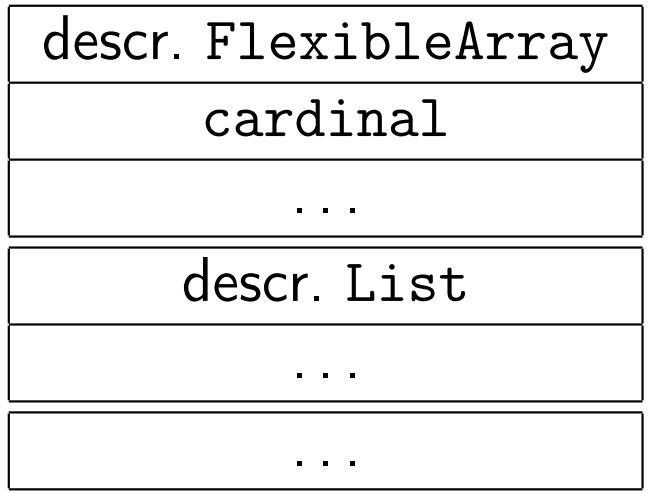


en particulier, un transtypage comme

```
FlexibleArray fa;
List l = (List)fa;
```

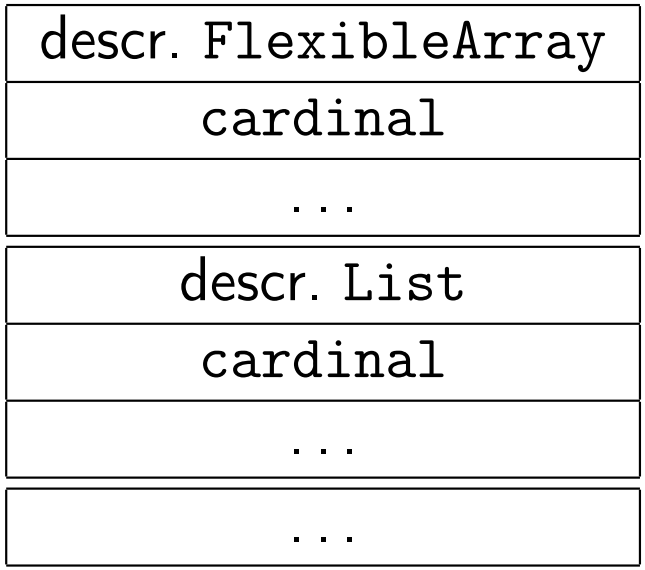
est traduit par une arithmétique de pointeur

```
l <- fa + 24
```



supposons maintenant que `List` hérite également de `Collection`

```
class Collection {
  int cardinal;
};
class Array : public Collection {
  int nth(int i) ...
};
class List : public Collection {
  void push(int v) ...
  int pop() ...
};
class FlexibleArray
  : public Array, public List {
};
```



on a maintenant **deux** champs `cardinal`

et donc une ambiguïté potentielle

```
int main() {
  FlexibleArray fa;
  cout << fa.cardinal << endl;
};
```

```
test.cc: In function ‘int main()’:
test.cc:42:14: error: request for member ‘cardinal’ is ambiguous
```

il faut préciser de quel champ `cardinal` il s'agit

```
int main() {
  FlexibleArray fa;
  cout << fa.Array::cardinal << endl;
};
```

pour n'avoir qu'une seule instance de `Collection`, il faut modifier la façon dont `Array` et `List` héritent de `Collection` (héritage virtuel)

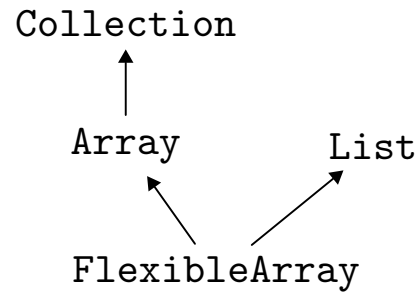
```
class Collection { ... };

class Array : public virtual Collection { ... };

class List : public virtual Collection { ... };

class FlexibleArray : public Array, public List {
```

il n'y a plus d'ambiguïté quant à `cardinal` :

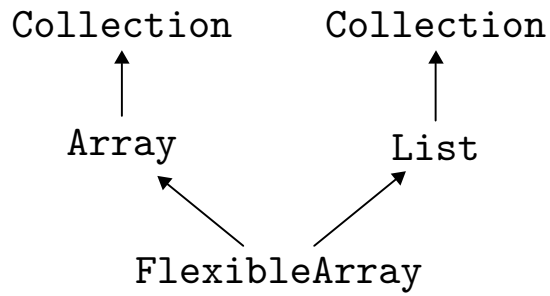
```
  FlexibleArray fa;
  cout << fa.cardinal << endl;
```

trois situations différentes

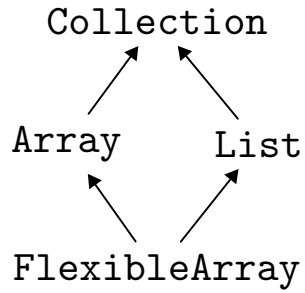
```
class Collection { ... };
class Array : Collection { ... };
class List { ... };
class FlexibleArray : Array, List { ... };
```




```
class Collection { ... };
class Array : Collection { ... };
class List : Collection { ... };
class FlexibleArray : Array, List { ... };
```




```
class Collection { ... };
class Array : virtual Collection { ... };
class List : virtual Collection { ... };
class FlexibleArray : Array, List { ... };
```



(le diamant)

la suite

- TD 9
  - production de code
    (on ajoute des fonctions au mini langage **Arith**)
- prochain cours
  - production de code efficace, partie 1