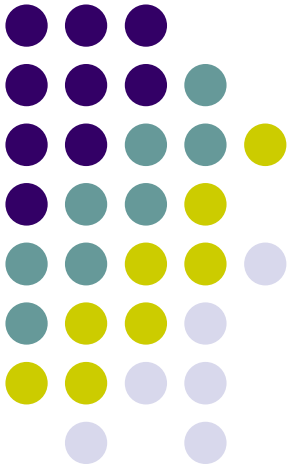
MI-GLSD-M1 -UEM213 : Programming languages paradigms

Chapitre III: paradigme orienté objets

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of Sciences & Technology
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Ce qu'on va voir aujourd'hui

- Résumé du dernier cours
  - Pourquoi la règle "l'appel récursif doit être le dernier appel" marche
  - Le paradigme logique en OZ plus performant.
- L'état explicite
  - L'utilité pour la modularité
- L'abstraction de données
  - Le type abstrait et l'objet

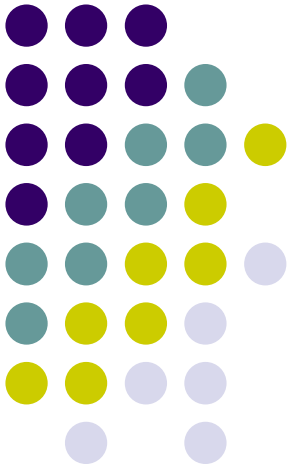
A.HARICHE a.hariche@univ-dbkm.dz

Résumé du dernier cours

paradigme Orienté Objets

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of Sciences & Technology
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Pourquoi l'optimisation du dernier appel marche

- Avec la sémantique, on peut démontrer que l'optimisation du dernier appel marche
- Nous allons prendre deux versions de la factorielle, une avec accumulateur (Fact2) et l'autre sans accumulateur (Fact)
- Nous allons les exécuter avec la sémantique
- Ces exemples se généralisent facilement pour toute fonction récursive

A.HARICHE a.hariche@univ-dbkm.dz

Factorielle avec accumulateur

- Voici une définition (partiellement) en langage noyau

```
proc {Fact2 I A F}
        if I==0 then F=A
        else I1 A1 in
                I1=I-1
                A1=I*A
                {Fact2 I1 A1 F}
        end
end
```

- Nous allons exécuter cette définition avec la sémantique
- Nous allons démontrer que la taille de la pile est la même juste avant chaque appel de Fact2

A.HARICHE a.hariche@univ-dbkm.dz

Début de l'exécution de Fact2

- Voici l'instruction qu'on va exécuter:

```
local N A F in
        N=5 A=1
        {Fact2 N A F}
end
```

- En fait, pour être complet il faut ajouter la définition de Fact2

A.HARICHE a.hariche@univ-dbkm.dz

Début de l'exécution (complet)

- Voici la vraie instruction qu'on va exécuter

```
local Fact2 in
        proc {Fact2 I A F}
                if I==0 then F=A
                else I1 A1 in
                        I1=I-1
                        A1=I*A
                        {Fact2 I1 A1 F}
                end
        end
        local N A F in
                N=5 A=1
                {Fact2 N A F}
        end
end
```


A.HARICHE a.hariche@univ-dbkm.dz

Premier appel de Fact2

- Voici l'état juste avant le premier appel:
  ([({Fact2 N A F}, {Fact2→*p*,N→*n*,A→*a*,F→*f* })],
  {*n*=5, *a*=1, *f*, *p*=(…)})
- Un pas plus loin (l'exécution de l'appel commence):
  ([(**if** I==0 **then** F=A **else** I1 A1 **in**
  I1=I-1 A1=A*I {Fact2 I1 A1 F} **end**,
  {Fact2→*p*,I→*n*,A→*a*,F→*f* })],
  {*n*=5, *a*=1, *f*, *p*=(…)})

A.HARICHE a.hariche@univ-dbkm.dz

Deuxième appel de Fact2

- Juste avant le deuxième appel:
  ([({Fact2 I1 A1 F},
  {Fact2$\rightarrow p$,I$\rightarrow n$,A$\rightarrow a$,F$\rightarrow f$,I1$\rightarrow i_1$,A1$\rightarrow a_1$})],
  {n=5, a=1, i_1=4, a_1=5, f, p=(…)})
- On voit que la pile ne contient qu'un seul élément, tout comme le premier appel
- On peut facilement voir que tous les appels de Fact2 seront le seul élément sur la pile
- QED!

A.HARICHE a.hariche@univ-dbkm.dz

Factorielle sans accumulateur

- Voici une définition (partiellement) en langage noyau

```
proc {Fact N F}
        if N==0 then F=1
        else N1 F1 in
                N1=N-1
                {Fact N1 F1}
                F=N*F1
        end
end
```

- Nous allons exécuter cette définition avec la sémantique, avec le premier appel {Fact 5 F}
- Nous allons démontrer que la taille de la pile augmente d'un élément pour chaque nouvel appel récursif

A.HARICHE a.hariche@univ-dbkm.dz

Début de l'exécution de Fact

- Voici l'exécution juste avant le premier appel:
  ([({Fact N F}, {Fact→*p*,N→*n*,F→*f* })],
  {*n*=5, *f*, *p*=(…)})
- La partie **else** de l'instruction **if**:
  ([(N1=N-1 {Fact N1 F1} F=N*F1,
  {Fact→*p*,N→*n*,F→*f*,N1→n_1,F1→f_1 })],
  {*n*=5, *f*, n_1, f_1, *p*=(…)})
- Juste avant le second appel de Fact:
  ([({Fact N1 F1},
  {Fact→*p*,N→*n*,F→*f*,N1→n_1,F1→f_1 })
  (F=N*F1,
  {Fact→*p*,N→*n*,F→*f*,N1→n_1,F1→f_1 })],
  {*n*=5, *f*, n_1=4, f_1, *p*=(…)})

Appel récursif

Après l'appel

A.HARICHE a.hariche@univ-dbkm.dz

Plus loin dans l'exécution

- Un des appels suivants de Fact:
  ([({Fact N1 F1}, {…}),
  (F=N*F1, {F→f_2,N→n_2,F1→f_3, …}),
  (F=N*F1, {F→f_1,N→n_1,F1→f_2, …}),
  (F=N*F1, {F→f,N→n,F1→f_1, …})],
  {n=5, f, n_1=4, f_1, n_2=3, f_2, …, p=(…)})
- A chaque appel successif, une autre instruction "F=N*F1" se met sur la pile
  - Avec un autre environnement bien sûr!
- La pile contient toutes les multiplications qui restent à faire

A.HARICHE a.hariche@univ-dbkm.dz

Généraliser ce résultat

- Pour que le résultat tienne pour toutes les fonctions récursives, il faut définir un schéma pour l'exécution d'une fonction récursive
  - Schéma = une représentation de l'ensemble de toutes les exécutions possibles des fonctions récursives
  - On redéfinit la sémantique pour marcher sur le schéma
- Est-ce que l'exécution de l'exemple concret tiendra toujours pour toutes les exécutions du schéma?
  - Oui!
  - La vérification de ce fait est hors de portée pour ce cours, mais si vous avez un esprit mathématique vous pouvez le démontrer rigoureusement

A.HARICHE a.hariche@univ-dbkm.dz

Conclusion

- Quand l'appel récursif est le dernier appel, la taille de la pile reste constante
- Quand l'appel récursif n'est pas le dernier appel, la taille de la pile grandit d'un élément pour chaque appel récursif
  - La pile contient toutes les instructions qui restent à faire
- La sémantique nous montre exactement ce qui se passe!

A.HARICHE a.hariche@univ-dbkm.dz

L’état

paradigme Orienté Objets

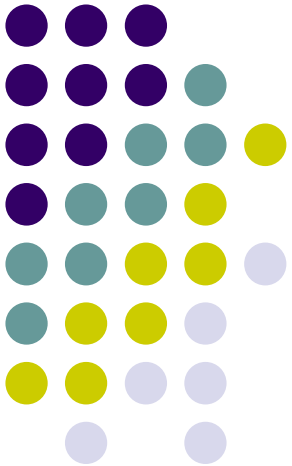
A. HARICHE

University of Djilali Bounaama, Khemis Meliana (UDBKM)

Faculty of Sciences & Technology

Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



La notion de temps

- Dans le modèle déclaratif, il n'y a pas de temps
  - Les fonctions sont des fonctions mathématiques, qui ne changent jamais
- Dans le monde réel, il y a le temps et le changement
  - Les organismes changent leur comportement avec le temps, ils grandissent et apprennent
  - Comment est-ce qu'on peut modéliser ce changement dans un programme?
- On peut ajouter une notion de temps abstrait dans les programmes
  - Temps abstrait = une séquence de valeurs
  - Un état = un temps abstrait = une séquence de valeurs

L’état est une séquence dans le temps

- Un état est une séquence de valeurs calculées progressivement, qui contiennent les résultats intermédiaires d’un calcul
- Le modèle déclaratif peut aussi utiliser l’état selon cette définition!
- Regardez bien la définition ci-jointe de Sum

```
fun {Sum Xs A}
   case Xs
   of nil then A
   [] X|Xr then
         {Sum Xr A+X}
   end
end

{Browse {Sum [1 2 3 4] 0}}
```


A.HARICHE a.hariche@univ-dbkm.dz

L’état implicite

- Les deux arguments Xs et A représentent un état implicite

Xs	A
[1 2 3 4]	0
[2 3 4]	1
[3 4]	3
[4]	6
nil	10

- Implicite parce qu’on n’a pas changé le langage de programmation
- C’est purement une interprétation de la part du programmeur

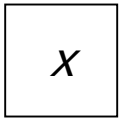
```
fun {Sum Xs A}
   case Xs
   of nil then A
   [] X|Xr then
         {Sum Xr A+X}
   end
end

{Browse {Sum [1 2 3 4] 0}}
```


A.HARICHE a.hariche@univ-dbkm.dz

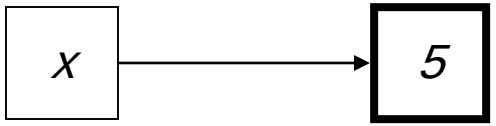
L’état explicite

- Un état peut aussi être explicite, c’est-à-dire, on fait une extension au langage
- Cette extension nous permettra d’exprimer directement une séquence de valeurs dans le temps
- Notre extension s’appellera une cellule
- Une cellule a un contenu qui peut être changé
- La séquence de contenus dans le temps est un état

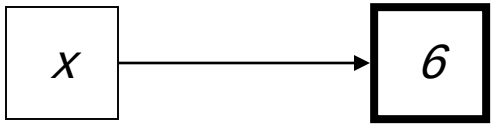


Une variable libre

Création d’une cellule avec contenu initiale 5



Changement du contenu qui devient 6

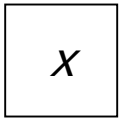


A.HARICHE a.hariche@univ-dbkm.dz

Une cellule

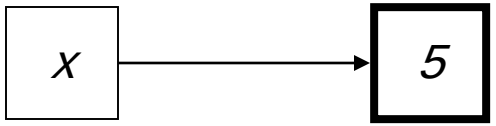
- Une cellule est un conteneur avec une identité et un contenu
  - L'identité est constante (le "nom" de la cellule)
  - Le contenu est une variable (qui peut être liée)
- Le contenu de la cellule peut être changé

```
X={NewCell 5}
{Browse @X}
X:=6
{Browse @X}
```

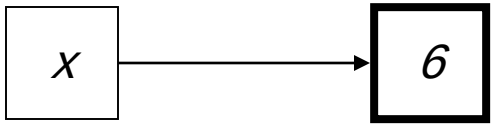


Une variable libre

Création d'une cellule avec contenu initiale 5



Changement du contenu qui est maintenant 6



A.HARICHE a.hariche@univ-dbkm.dz

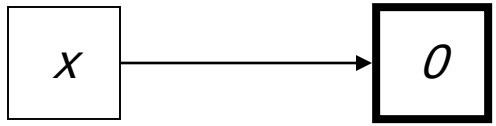
Opérations sur une cellule

- On ajoute le concept de la cellule au langage noyau
  - Il y a trois opérations de base
- X={NewCell I}
  - Créé une nouvelle cellule avec contenu initial I
  - Lie X à l'identité de la cellule
- X:=J
  - Suppose X est lié à l'identité d'une cellule
  - Change le contenu de la cellule pour devenir J
- Y=@X
  - Suppose X est lié à l'identité d'une cellule
  - Affecte Y au contenu de la cellule

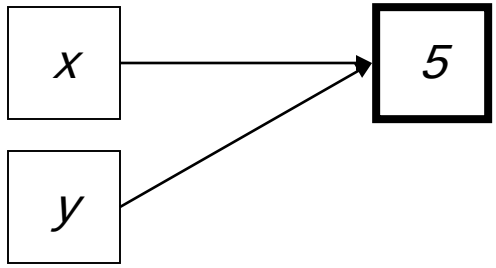
A.HARICHE a.hariche@univ-dbkm.dz

Quelques exemples (1)

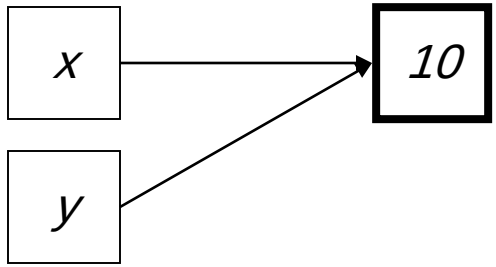
- X={NewCell 0}



- X:=5
- Y=X



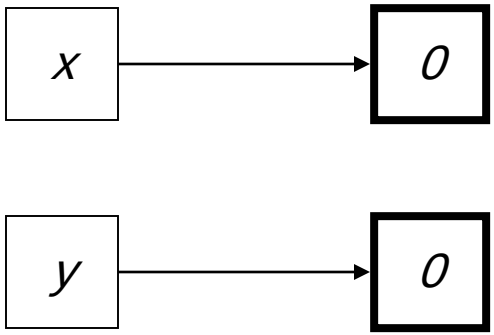
- Y:=10
- @X==10 % true
- X==Y % true



A.HARICHE a.hariche@univ-dbkm.dz

Quelques exemples (2)

- X={NewCell 0}
- Y={NewCell 0}



- X==Y % false
- Parce que X et Y font référence à des cellules différentes, avec identités différentes

- @X==@Y % true

A.HARICHE a.hariche@univ-dbkm.dz

Egalité de structure et égalité d'identité

- Deux listes sont égales si leurs structures sont égales (égalité de structure)
  - Même si les structures ont été créé séparément
  - A=[1 2] B=[1 2]
    {Browse A==B} % true
- Deux cellules sont égales s'il s'agit de la même cellule (égalité d'identité)
  - Deux cellules créées séparément sont toujours différentes
  - C={NewCell 0} D={NewCell 0}
    {Browse C==D} % false
    {Browse @C==@D} % true

A.HARICHE a.hariche@univ-dbkm.dz

Sémantique des cellules (1)

- Extension de la mémoire de la machine abstraite
- La mémoire a maintenant deux parties:
  - Mémoire à affectation unique (variables)
  - Mémoire à affectation multiple (cellules)
- Une cellule est une paire, un nom et un contenu.
  - Le contenu est une variable!
  - Le nom est un constant (une variable liée)
- Affectation de la cellule
  - Changez la paire: faire en sorte que le nom référencie un autre contenu
  - Ni le nom ni le contenu changent!

A.HARICHE a.hariche@univ-dbkm.dz

Sémantique des cellules (2)

- La mémoire $\sigma = \sigma_1 \cup \sigma_2$ a maintenant deux parties
  - Mémoire à affectation unique
    $\sigma_1 = \{t, u, v, w, x=\zeta, y=\xi, z=10, w=5\}$
  - Mémoire à affectation multiple
    $\sigma_2 = \{x:t, y:w\}$
- Dans σ_2 il y a deux cellules, *x* et *y*
  - Le nom de *x* est la constante ζ
  - L'opération X:=Z transforme *x*:*t* en *x*:*z*
  - L'opération @Y donne le résultat *w*
  - (dans l'environnement $\{X \rightarrow x, Y \rightarrow y, Z \rightarrow z, W \rightarrow w\}$)

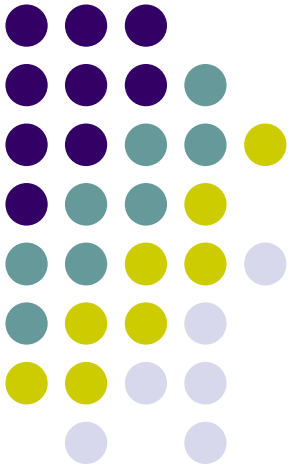
A.HARICHE a.hariche@univ-dbkm.dz

L’état et la modularité

paradigme Orienté Objets

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of Sciences & Technology
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



L'état est bénéfique pour la modularité

- On dit qu'un système (ou programme) est modulaire si des mises à jour dans une partie n'obligent pas à changer le reste
  - Partie = fonction, procédure, composant, …
- Nous allons vous montrer un exemple comment l'utilisation de l'état explicite nous permet de construire un système modulaire
  - Dans le modèle déclaratif ce n'est pas possible

A.HARICHE a.hariche@univ-dbkm.dz

Scénario de développement (1)

- Il y a trois personnes, P, U1 et U2
- P a développé le module M qui offre deux fonctions F et G
- U1 et U2 sont des développeurs qui ont besoin du module M

```
fun {MF}
   fun {F ...}
      ⟨Définition de F⟩
   end
   fun {G ...}
      ⟨Définition de G⟩
   end
in 'export'(f:F  g:G)
end
M = {MF}
```


A.HARICHE a.hariche@univ-dbkm.dz

Scénario de développement (2)

- Développeur U2 a une application très coûteuse en temps de calcul
- Il veut étendre le module M pour compter le nombre de fois que la fonction F est appelée par son application
- Il va voir P et lui demande de faire cela sans changer l'interface de M

```
fun {MF}
   fun {F ...}
      ⟨Définition de F⟩
   end
   fun {G ...}
      ⟨Définition de G⟩
   end
in 'export'(f:F  g:G)
end
```


A.HARICHE a.hariche@univ-dbkm.dz

Dilemme!

- Ceci est impossible dans le modèle déclaratif parce que F ne se souvient pas de ses appels précédents!
- La seule solution est de changer l'interface de F en ajoutant deux arguments Fin et Fout:
  fun {F … Fin Fout} Fout=Fin+1 … **end**
- Le reste du programme doit assurer que la sortie Fout d'un appel de F soit l'entrée Fin de l'appel suivant de F
- Mais l'interface de M a changé
  - Tous les utilisateurs de M, même U1, doivent changer leur programme

A.HARICHE a.hariche@univ-dbkm.dz

Solution avec l'état explicite

- Créez une cellule quand MF est appelé
- A cause de la portée lexicale, la cellule X est cachée du programme: elle n'est visible que dans le module M
- M.f n'a pas changé
- Une nouvelle fonction M.c est disponible

```
fun {MF}
   X = {NewCell 0}
   fun {F ...}
      X:=@X+1
      ⟨Définition de F⟩
   end
   fun {G ...} ⟨Définition de G⟩
   end
   fun {Count} @X end
in 'export'(f:F  g:G c:Count)
end
M = {MF}
```


A.HARICHE a.hariche@univ-dbkm.dz

Conclusion: comparaison des modèles

- Modèle déclaratif:
  - + Un composant ne change jamais son comportement
  - – La mise à jour d'un composant implique les mises à jour de beaucoup d'autres composants
- Modèle avec état:
  - + Un composant peut être mise à jour sans changer le reste du programme
  - – Un composant peut changer son comportement à cause des appels précédents
- On peut parfois combiner les deux avantages
  - Utiliser l'état pour aider la mise à jour, mais quand même faire attention à ne jamais changer le comportement d'un composant

A.HARICHE a.hariche@univ-dbkm.dz

Un autre exemple: la mémoisation

- La mémoisation d'une fonction:
  - La fonction garde un tableau interne qui contient les arguments et les résultats des appels précédents
  - A chaque nouvel appel, si l'argument est dans le tableau, on peut donner le résultat sans faire le calcul de la fonction
- La mémoisation implique un état explicite dans la fonction
  - Mais la fonction garde un comportement déclaratif

A.HARICHE a.hariche@univ-dbkm.dz

Motivation pour l'abstraction

paradigme Orienté Objets

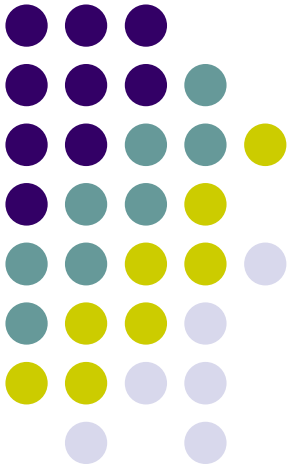
A. HARICHE

University of Djilali Bounaama, Khemis Meliana (UDBKM)

Faculty of Sciences & Technology

Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



L'importance de l'encapsulation

- Imaginez que votre télévision n'aurait pas de boîtier
  - Tous les circuits de l'intérieur seraient exposés à l'extérieur
- C'est dangereux pour vous: si vous touchez aux circuits, vous pouvez vous exposez à des tensions mortelles
- C'est problématique pour la télévision: si vous versez une tasse de café dans l'intérieur, vous pouvez provoquez un court-circuit
  - Vous pouvez être tenté à chipoter avec l'intérieur, pour soi-disant "améliorer" les performances de la télévision
- Il y a donc un intérêt à faire une encapsulation
  - Un boîtier qui empêcherait une interaction sauvage, et qui n'autoriserait que les interactions voulues (marche/arrêt, volume)

L'encapsulation dans l'informatique

- Supposez que votre programme utilise une pile avec l'implémentation suivante:

```
fun {NewStack} nil end
fun {Push S X} X|S end
fun {Pop S X} X=S.1 S.2 end
fun {IsEmpty S} S==nil end
```

- Cette implémentation n'est pas encapsulée!
  - Il y a les mêmes problèmes qu'avec la télévision
  - La pile est implémentée avec une liste et la liste n'est pas protégée
  - Le programmeur peut créer des piles autrement qu'avec les opérations voulues
  - On ne peut pas garantir que la pile marchera toujours!

A.HARICHE a.hariche@univ-dbkm.dz

Une pile encapsulée

- On utilise un “boîtier” qui isole l’intérieur de la pile (l’implémentation) de l’extérieur
  - On verra comment faire cela dans un programme!
- Le programmeur ne peut pas regarder à l’intérieur
- Le programmeur peut manipuler des piles seulement avec les opérations autorisées
  - On peut donc garantir que la pile marchera toujours
  - L’ensemble d’opérations autorisées = l’interface
- Le programmeur a la vie plus simple!
  - Toute la complexité de l’implémentation de la pile est cachée

A.HARICHE a.hariche@univ-dbkm.dz

Avantages de l'encapsulation

- La garantie que l'abstraction marchera toujours
  - L'interface est bien définie (les opérations autorisées)
- La réduction de complexité
  - L'utilisateur de l'abstraction ne doit pas comprendre comment l'abstraction est réalisée
  - Le programme peut être partitionné en beaucoup d'abstractions réalisées de façon indépendante, ce qui simplifie de beaucoup la complexité pour le programmeur
- Le développement de grands programmes devient possible
  - Chaque abstraction a un responsable: la personne qui l'implémente et qui garantie son comportement
  - On peut donc faire des grands programmes en équipe
  - Il suffit que chaque responsable connaît bien les interfaces des abstractions qu'il utilise

A.HARICHE a.hariche@univ-dbkm.dz

L’abstraction de données

paradigme Orienté Objets

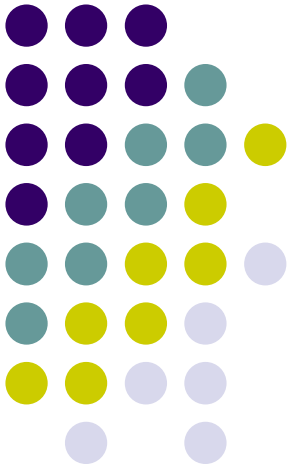
A. HARICHE

University of Djilali Bounaama, Khemis Meliana (UDBKM)

Faculty of Sciences & Technology

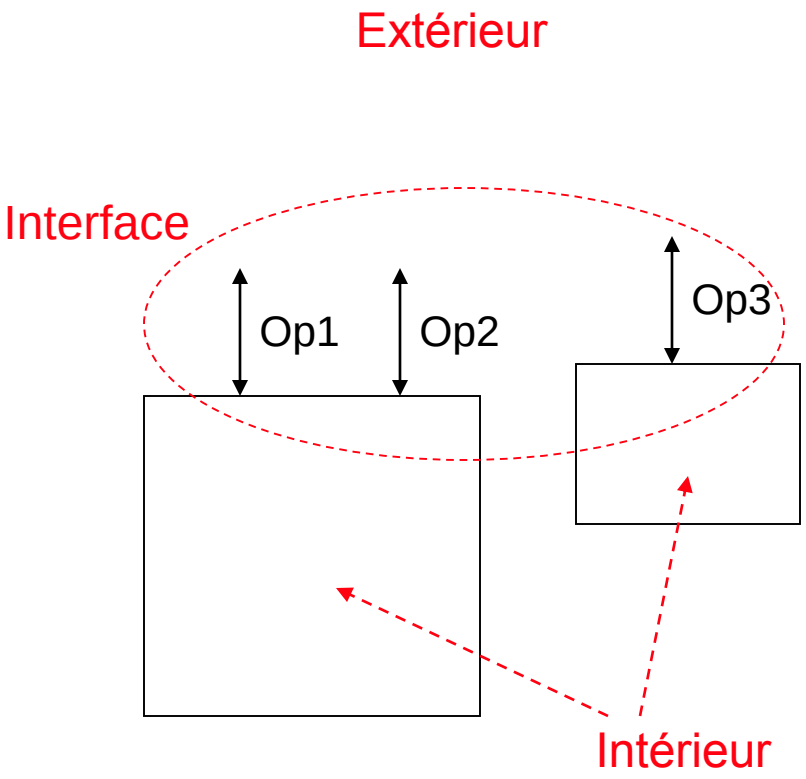
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



L'abstraction de données

- Une abstraction de données a un intérieur, un extérieur et une interface entre les deux
- L'intérieur est caché de l'extérieur
  - Toute opération sur l'intérieur doit passer par l'interface
- Cette encapsulation peut avoir un soutien du langage
  - Sans soutien est parfois bon pour de petits programmes
  - Nous allons voir comment le langage peut faire respecter l'encapsulation



A.HARICHE a.hariche@univ-dbkm.dz

Différentes formes d'abstractions de données

- Il y a plusieurs manières d'organiser une abstraction de données
- Les deux manières principales sont l'objet et le type abstrait
  - Un type abstrait a des valeurs et des opérations
  - Un objet contient en lui-même la valeur et le jeu d'opérations
- Regardons cela de plus près!

A.HARICHE a.hariche@univ-dbkm.dz

Une abstraction sans état: le type abstrait

- L'abstraction consiste en un ensemble de valeurs et des opérations sur ces valeurs
- Exemple: les entiers
  - Valeurs: 1, 2, 3, …
  - Opérations: +, -, *, div, …
- Il n'y a pas d'état
  - Les valeurs sont des constantes
  - Les opérations n'ont pas de mémoire interne

A.HARICHE a.hariche@univ-dbkm.dz

Autre exemple d'un type abstrait: une pile

- Il y a des valeurs et des opérations
  - Valeurs: toutes les piles possibles
  - Opérations: NewStack, Push, Pop, IsEmpty
- Les opérations prennent des piles comme arguments et résultats
  - S={NewStack}
  - S2={Push S X}
  - S2={Pop S X}
  - {IsEmpty S}
- Attention: les piles sont des valeurs, donc des constantes!

A.HARICHE a.hariche@univ-dbkm.dz

Implémentation d'une pile en type abstrait

- Opérations:
  - **fun** {NewStack} nil **end**
  - **fun** {Push S X} X|S **end**
  - **fun** {Pop S X} X=S.1 S.2 **end**
  - **fun** {IsEmpty S} S==nil **end**
- La pile est représentée par une liste!
- Mais la liste n'est pas protégée
- Comment est-ce qu'on peut protéger l'intérieur du type abstrait de l'extérieur?

A.HARICHE a.hariche@univ-dbkm.dz

Utilisation de la pile (en type abstrait)

S1={NewStack}

S2={Push S1 a}

S3={Push S2 b}

local X **in** S4={Pop S3 X} {Browse X} **end**

A.HARICHE a.hariche@univ-dbkm.dz

Implémentation protégée d'une pile en type abstrait

- Pour protéger l'intérieur, nous utilisons un 'wrapper' (un emballage sécurisé)
- {NewWrapper ?Wrap ?Unwrap} crée un nouvel emballeur:
  - W={Wrap X} % W contient X
  - X={Unwrap W} % Retrouver X à partir de W
- Nouvelle implémentation:

```
local Wrap Unwrap in
      {NewWrapper Wrap Unwrap}
      fun {NewStack} {Wrap nil} end
      fun {Push W X} {Wrap X|{Unwrap W}} end
      fun {Pop W X} S={Unwrap W} in X=S.1 {Wrap S.2} end
      fun {IsEmpty W} {Unwrap W}==nil end
end
```

- On peut implémenter NewWrapper
  - Un autre moment

A.HARICHE a.hariche@univ-dbkm.dz

Implémentation des types abstraits

- Il existe des langages qui permettent au développeur de créer de nouveaux types abstraits
  - Le langage CLU, développé par Barbara Liskov et son équipe dans les années 1970, est l'exemple majeur
- Ces langages soutiennent une notion de protection similaire à Wrap/Unwrap
  - CLU a un soutien syntaxique qui facilite de beaucoup la définition de types abstraits

A.HARICHE a.hariche@univ-dbkm.dz

Une abstraction avec état: l'objet

- L'abstraction consiste en un ensemble d'objets
  - Pas de distinction entre valeurs et opérations
  - Les objets jouent le rôle des deux
- Exemple: une pile
  - S={NewStack}
  - {S push(X)}
  - {S pop(X)}
  - {S isEmpty(B)}
- L'état de la pile est dans l'objet S
- S soutient des opérations, on dit que l'on "envoie un message" à l'objet

A.HARICHE a.hariche@univ-dbkm.dz

Implémentation d'une pile en objet

- Les mêmes opérations, mais organisée en objet:

```
fun {NewStack}
      C={NewCell nil}
      proc {Push X} C:=X|@C end
      proc {Pop X} S=@C in C:=S.2 X=S.1 end
      proc {IsEmpty B} B=(@C==nil) end
in
      proc {$ M}
          case M of push(X) then {Push X}
          [] pop(X) then {Pop X}
          [] isEmpty(B) then {IsEmpty B} end
      end
end
```

- L'objet est protégé

A.HARICHE a.hariche@univ-dbkm.dz

Utilisation de la pile (en objet)

S={NewStack}

{S push(a)}

{S push(b)}

local X **in** {S pop(X)} {Browse X} **end**

A.HARICHE a.hariche@univ-dbkm.dz

Comparaison entre objets et types abstraits

- Quels sont les avantages et désavantages respectifs des objets et des types abstraits?
- L'avantage majeur des types abstraits est qu'on peut faire des opérations qui prennent plusieurs valeurs comme arguments
  - **fun** {Add Int1 Int2} … **end**
  - On ne peut pas faire cela avec uniquement des objets
- L'avantage majeur des objets est l'héritage
  - La définition incrémentale des objets qui se ressemblent
  - Nous allons voir l'héritage la semaine prochaine
- Il y a aussi le polymorphisme
  - La raison principale du succès des objets (à mon avis)
  - C'est aussi possible avec les types abstrait mais moins facile
  - Nous allons voir le polymorphisme la semaine prochaine

Utilisation des objets et types abstraits en Java

- Contrairement à l'opinion reçue, Java fait un savant mélange d'objets et de types abstraits
- Les entiers en Java sont des types abstraits
  - C'est indispensable, parce que les instructions machine prennent plusieurs entiers comme arguments!
- Si O1 et O2 sont deux objets de la même classe, alors O1 peut regarder les attributs privés de O2!
  - On peut faire des méthodes qui prennent plusieurs objets comme arguments; c'est une propriété "type abstrait" qui est greffée sur les objets
- Smalltalk, un langage "purement" orienté-objet, ne permet pas de faire ces choses

A.HARICHE a.hariche@univ-dbkm.dz

D’autres formes d’abstractions de données

- Il y a deux autres formes possibles d’abstractions de données protégées
- Un “objet déclaratif” (objet sans état) est possible
  - Cette forme est plutôt une curiosité!
- Un “type abstrait avec état” est possible
  - Cette forme peut être très utile!
  - Regardons de plus près cette forme

A.HARICHE a.hariche@univ-dbkm.dz

Un type abstrait avec état

- Voici la pile en type abstrait avec état:
  - S={NewStack}
  - {Push S X}
  - X={Pop S}
  - B={IsEmpty S}
- La pile est passée comme argument aux opérations
  - Comme un type abstrait
- La pile elle même est changée; elle a un état
  - Comme un objet
- Cette forme est intéressant si on veut étendre le jeu d'opérations de la pile
  - On peut définir une nouvelle opération indépendamment des autres; ce n'est pas possible avec un objet classique

A.HARICHE a.hariche@univ-dbkm.dz

Utilisation de la pile (en type abstrait avec état)

S={NewStack}

{Push S a}

{Push S b}

{Browse {Pop S}}

A.HARICHE a.hariche@univ-dbkm.dz

Résumé

paradigme Orienté Objets

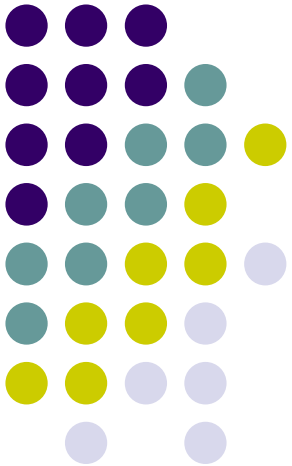
A. HARICHE

University of Djilali Bounaama, Khemis Meliana (UDBKM)

Faculty of Sciences & Technology

Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Résumé

- La sémantique
  - Pourquoi la règle “appel récursif en dernier” marche
- L’état
  - L’état explicite (la cellule)
  - L’avantage pour la modularité des programmes
  - La sémantique des cellules
- L’abstraction de données
  - Motivation: donner des garanties, la réduction de complexité, faire de grands programmes en équipe
  - Les deux formes principales: le type abstrait et l’objet
  - L’utilité de chaque forme et la mise en oeuvre en Java
  - Le type abstrait avec état

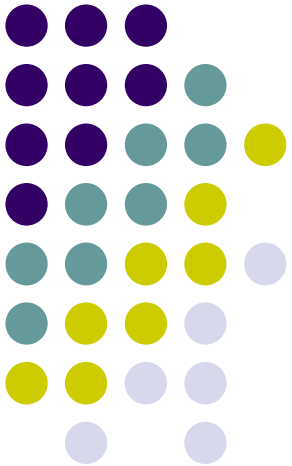
A.HARICHE a.hariche@univ-dbkm.dz

Résumé du dernier cours

paradigme Orienté Objets

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of Sciences & Technology
Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



La sémantique

- Il est important de comprendre comment exécute un programme
  - Celui qui ne comprend pas quelque chose est l'esclave de cette chose
- Il faut pouvoir montrer l'exécution d'un programme selon la machine abstraite
  - Concepts importants: environnement contextuel ("le lieu de naissance"), pile sémantique ("ce qu'il reste à faire"), appel de procédure
- Il ne faut pas sombrer dans les détails
  - Il suffit de les donner une fois, après vous pouvez faire des raccourcis (comme par exemple, sauter des pas, ne montrer qu'une partie de la pile, de la mémoire et des environnements, utiliser des substitutions)

A.HARICHE a.hariche@univ-dbkm.dz

L’état

- L’état explicite (la cellule)
- L’avantage pour la modularité des programmes
  - Etendre une partie sans devoir changer le reste
- La sémantique des cellules
  - Une cellule est une paire
  - Le nom de la cellule est aussi appelé l’adresse
  - La mémoire à affectation multiple

L’abstraction de données

- Motivations
  - Donner des garanties
  - Réduire la complexité
  - Faire de grands programmes en équipe
- Les deux formes principales
  - Le type abstrait et l’objet
- L’utilité de chaque forme et la mise en oeuvre en Java
- Le type abstrait avec état

Les objets et les classes

paradigme Orienté Objets

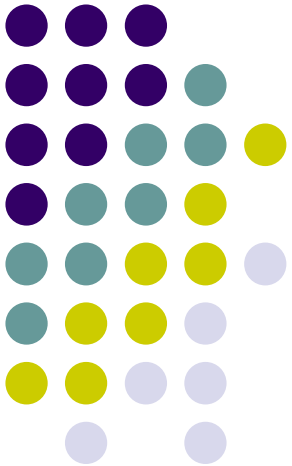
A. HARICHE

University of Djilali Bounaama, Khemis Meliana (UDBKM)

Faculty of Sciences & Technology

Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Programmation avec objets

- Le concept d'objet est devenu omniprésent dans l'informatique aujourd'hui
  - Une abstraction de données qui contient à la fois la valeur et les opérations
  - Originaire de Simula 67, partout via Smalltalk et C++
- Avantages
  - L'abstraction de données
  - Le polymorphisme
  - L'héritage
- Les types abstraits sont tout aussi omniprésents mais moins médiatisés!
  - Il est important de bien distinguer objets et types abstraits et comprendre comment ils sont mélangés dans chaque langage

A.HARICHE a.hariche@univ-dbkm.dz

Schéma général d'un objet

```
local
   A1={NewCell I1}
   …
   An={NewCell In}
in
   proc {M1 …} … end
   …
   proc {Mm …} … end
end
```

Ce fragment créé des nouvelles cellules locales A1, …, An, qui ne sont visibles que dans les procédures globales M1, …, Mm.

On appelle souvent A1, …, An des "attributs" et M1, …, Mm des "méthodes"

A.HARICHE a.hariche@univ-dbkm.dz

Exemple: un objet “compteur”

```
declare
local
   A1={NewCell 0}
in
   proc {Inc} A1:=@A1+1 end
   proc {Get X} X=@A1 end
end
```


A.HARICHE a.hariche@univ-dbkm.dz

Le compteur avec envoi procédural

```
declare
local
   A1={NewCell 0}
   proc {Inc} A1:=@A1+1 end
   proc {Get X} X=@A1 end
in
   proc {Counter M}
      case M of inc then {Inc}
      [] get(X) then {Get X}
      end
   end
end
```

Toutes les méthodes sont invoquées par l'intermédiaire d'un seul point d'entrée: la procédure Counter:

```
{Counter inc}
{Counter inc}
{Counter get(X)}
```

L'argument de Counter est appelé un "message"

A.HARICHE a.hariche@univ-dbkm.dz

Une usine pour créer des compteurs

```
declare
fun {NewCounter}
   A1={NewCell 0}
   proc {Inc} A1:=@A1+1 end
   proc {Get X} X=@A1 end
in
   proc {$ M}
      case M of inc then {Inc}
      [] get(X) then {Get X}
      end
   end
end
```

Il est souvent intéressant de faire plusieurs objets avec les mêmes opérations mais avec des états différents

On peut définir une fonction qui, quand on l'appelle, créé un nouvel objet à chaque fois

L'appel C={NewCounter} créé l'attribut A1 et rend un objet avec méthodes Inc et Get

L’utilisation de la fonction NewCounter

C1={NewCounter}

C2={NewCounter}

{C1 inc}

{C1 inc}

local X **in** {C1 get(X)} {Browse X} **end**

A.HARICHE a.hariche@univ-dbkm.dz

Syntaxe pour une classe

```
class Counter
   attr a1
   meth init a1:=0 end
   meth inc a1:=@a1+1 end
   meth get(X) X=@a1 end
end

C1={New Counter init}
{C1 inc}
local X in {C1 get(X)} {Browse X} end
```


A.HARICHE a.hariche@univ-dbkm.dz

C’est quoi une classe?

- La classe Counter est passée comme argument à la fonction New:
  - C={New Counter Init}
  - La classe Counter est une valeur (tout comme une procédure)!
  - La définition de la classe et la création de l’objet sont séparées
    - Pour les futés: les classes sont un type abstrait
- La fonction NewCounter fait les deux choses en même temps
  - Le résultat est le même
- Comment est-ce qu’on représente une classe comme une valeur?
  - Une classe est un enregistrement qui regroupe les noms des attributs et les méthodes
  - La fonction New prend l’enregistrement, créé les cellules et créé l’objet (une procédure qui référencie les cellules et les méthodes)
  - Exercice: lisez et comprenez la section 7.2, en particulier les figures 7.1, 7.2 et 7.3

A.HARICHE a.hariche@univ-dbkm.dz

Schéma général d'une classe

class C

  attr a_1 … a_n

  meth m_1 … **end**

  …

  meth m_m … **end**

end

A.HARICHE a.hariche@univ-dbkm.dz

Le polymorphisme

paradigme Orienté Objets

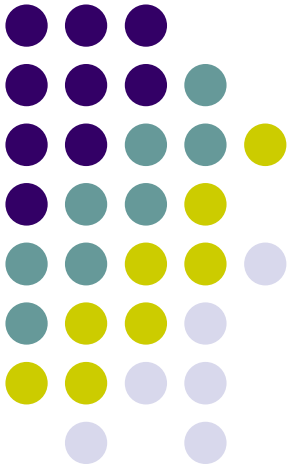
A. HARICHE

University of Djilali Bounaama, Khemis Meliana (UDBKM)

Faculty of Sciences & Technology

Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Le polymorphisme

- Dans le langage de tous les jours, une entité est polymorphe si elle peut prendre des formes différentes
- Dans le contexte de l'informatique, une opération est polymorphe si elle peut prendre des arguments de types différents
- Cette possibilité est importante pour que les responsabilités soient bien réparties sur les différentes parties d'un programme

A.HARICHE a.hariche@univ-dbkm.dz

Le principe de la répartition des responsabilités

- Le polymorphisme permet d'isoler des responsabilités dans les parties du programme qui les concernent
  - En particulier, une responsabilité doit de préférence être concentrée dans une seule partie du programme
- Exemple: un patient malade va chez un médecin
  - Le patient ne devrait pas être médecin lui-même!
  - Le patient dit au médecin: "guérissez-moi"
  - Le médecin fait ce qu'il faut selon sa spécialité
- Le programme "guérir d'une maladie" est polymorphe: il marche avec toutes sortes de médecins
  - Le médecin est un argument du programme
  - Tous les médecins comprennent le message "guérissez-moi"

A.HARICHE a.hariche@univ-dbkm.dz

Réaliser le polymorphisme

- Toutes les formes d'abstraction de données soutiennent le polymorphisme
  - Les objets et les types abstraits
  - C'est particulièrement simple avec les objets
    - Une des raisons du succès des objets
  - Pour ne par surcharger le cours, on ne parlera que des objets
- L'idée est simple: si un programme marche avec une abstraction de données, il pourrait marcher avec une autre, si l'autre a la même interface

A.HARICHE a.hariche@univ-dbkm.dz

Exemple de polymorphisme

```
class Figure
    …
end
class Circle
    attr x y r
    meth draw … end
    …
end
class Line
    attr x1 y1 x2 y2
    meth draw … end
    …
end
```

```
class CompoundFigure
    attr figlist
    meth draw
        for F in @figlist do
            {F draw}
        end
    end
    …
end
```

La définition de la méthode draw de CompoundFigure marche pour toutes les figures possibles: des cercles, des lignes et aussi d'autres CompoundFigures!

A.HARICHE a.hariche@univ-dbkm.dz

Exécution correcte d'un programme polymorphe

- Quand est-ce qu'un programme polymorphe est correct?
  - Pour une exécution correcte, l'abstraction doit satisfaire à certaines propriétés
  - Le programme marchera alors avec toute abstraction qui a ces propriétés
  - Pour chaque abstraction, il faut donc vérifier que sa spécification satisfait à ces propriétés
- Pour l'exemple des médecins, le programme exige que le médecin veut la guérisson du patient
  - Le polymorphisme marche si chaque médecin veut la guérisson du patient
  - Chaque abstraction ("médecin") satisfait la même propriété ("veut la guérisson du patient")

A.HARICHE a.hariche@univ-dbkm.dz

L’héritage

paradigme Orienté Objets

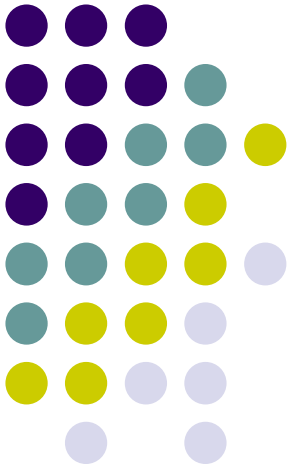
A. HARICHE

University of Djilali Bounaama, Khemis Meliana (UDBKM)

Faculty of Sciences & Technology

Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Définition incrémentale des abstractions de données

- Des abstractions de données sont souvent très similaires
- Par exemple, la notion de “collection” d’éléments a beaucoup de variations
  - Ensemble: des éléments sans ordre défini
  - Séquence: un ensemble d’éléments dans un ordre
    - Séquence = ensemble + ordre
  - Pile: une séquence où l’on ajoute et enlève du même côté
    - Pile = séquence + contraintes sur ajout/enlèvement
  - File: une séquence où l’on ajoute d’un côté et enlève de l’autre côté
    - File = séquence + contraintes sur ajout/enlèvement

A.HARICHE a.hariche@univ-dbkm.dz

L'héritage et les classes

- Il peut être intéressant de définir des abstractions sans répéter les parties communes
  - Parties communes = code dupliqué
  - Si une partie est changée, toutes les parties doivent être changées
    - Source d'erreurs!
- L'héritage est une manière de définir des abstractions de façon incrémentale
  - Une définition A peut "hériter" d'une autre définition B
  - La définition A prend B comme base, avec éventuellement des modifications et des extensions
- La définition incrémentale A est appelée une classe
  - Attention: le résultat est une abstraction de données complète

A.HARICHE a.hariche@univ-dbkm.dz

L'héritage et la sémantique

- Une classe A est définie comme une transformation d'une autre classe B
- L'héritage peut être vu comme transformation syntaxique
  - On prend le code source de B et on le modifie
- L'héritage peut aussi être vu comme transformation sémantique
  - La définition de A est une fonction f_A avec $c_A=f_A(c_B)$
  - f_A prend une abstraction de données B comme entrée (la valeur c_B) et donne comme résultat une autre abstraction de données (la valeur c_A)

A.HARICHE a.hariche@univ-dbkm.dz

Dangers de l'héritage

- L'héritage est parfois très utile, mais il faut l'utiliser avec beaucoup de précautions
- La possibilité d'étendre A avec l'héritage peut être vue comme une autre interface à A
  - Une autre manière d'interagir avec A
- Cette interface doit être maintenue pendant toute la vie de A
  - Une source supplémentaire d'erreurs!

A.HARICHE a.hariche@univ-dbkm.dz

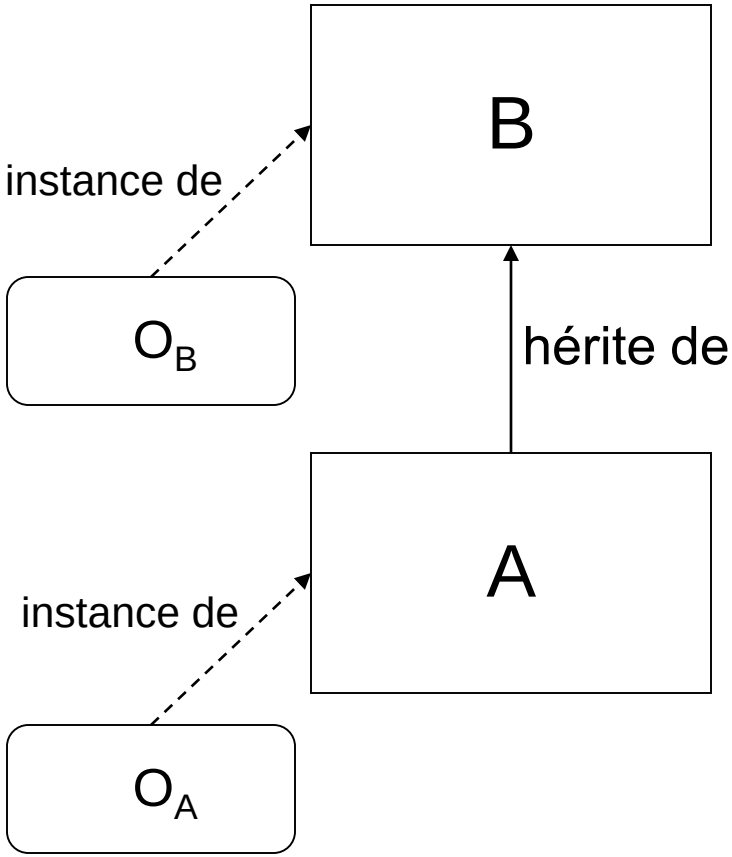
Notre recommandation

- Nous recommandons d'utiliser l'héritage le moins possible
  - C'est dommage que l'héritage est considéré comme tellement important par les mandarins de la programmation orienté-objet
- Quand on définit une classe, nous recommandons de la prendre comme "final" (non-extensible) par défaut
- Nous recommandons d'utiliser la composition de préférence sur l'héritage
  - La composition = une classe peut dans son implémentation utiliser d'autres classes (comme la liste de figures dans CompoundFigure)

A.HARICHE a.hariche@univ-dbkm.dz

Le principe de substitution

- La bonne manière d'utiliser l'héritage
  - Supposons que la classe A hérite de B et qu'on a deux objets, O_A et O_B
- Toute procédure qui marche avec O_B doit marcher avec O_A
  - L'héritage ne doit rien casser!
  - A est une extension conservatrice de B



A.HARICHE a.hariche@univ-dbkm.dz

Exemple: classe Account

```
class Account
   attr balance:0
   meth transfer(Amount)
        balance := @balance+Amount
   end
   meth getBal(B)
        B=@balance
   end
end
A={New Account transfer(100)}
```

Extension conservatrice (respecte le p. de s.)

VerboseAccount:
Un compte qui affiche toutes les transactions

```
class VerboseAccount
    from Account
    meth verboseTransfer(Amount)
            …
    end
end
```

La classe VerboseAccount a les méthodes transfer, getBal et verboseTransfer

A.HARICHE a.hariche@univ-dbkm.dz

Extension non-conservatrice (ne respecte pas le p. de s.)

AccountWithFee:
Un compte avec des frais

```
class AccountWithFee
    from VerboseAccount
    attr fee:5
    meth transfer(Amount)
            …
    end
end
```

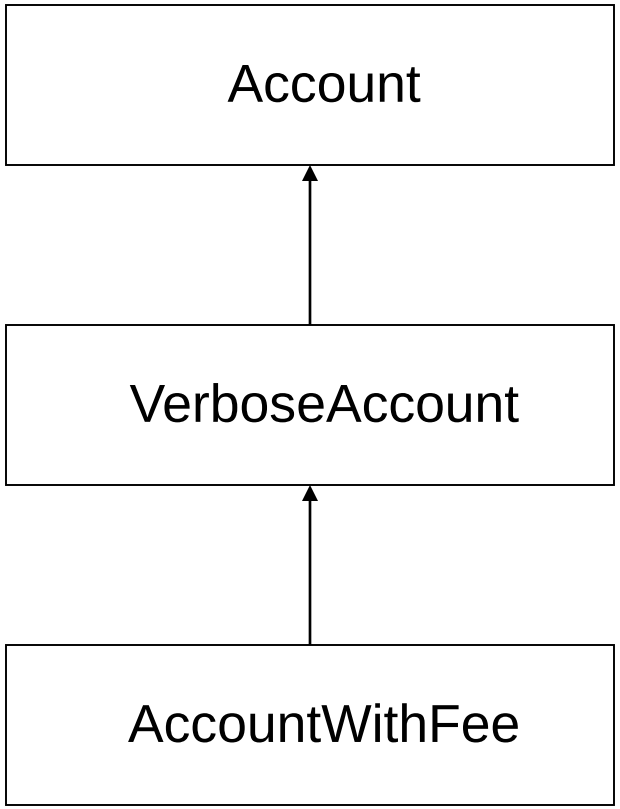
La classe AccountWithFee a les méthodes transfer, getBal et verboseTransfer. La méthode transfer a été redéfinie.

A.HARICHE a.hariche@univ-dbkm.dz

Hiérarchie de classe de l'exemple

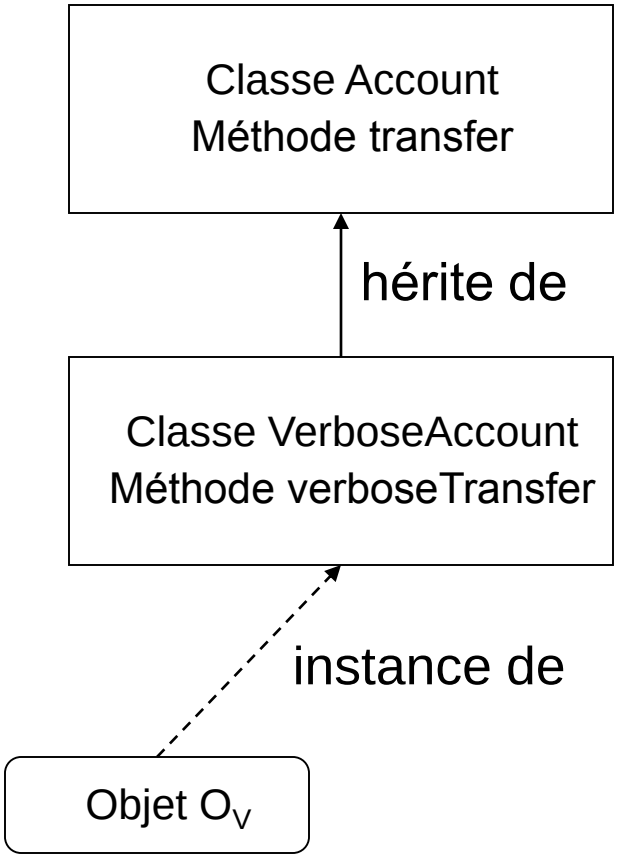
```
class VerboseAccount
    from Account
    meth verboseTransfer(Amount)
            …
    end
end

class AccountWithFee
    from VerboseAccount
    attr fee:5
    meth transfer(Amount)
            …
    end
end
```



Lien dynamique

- Nous allons maintenant définir la nouvelle méthode verboseTransfer
- Dans la définition de verboseTransfer, nous devons appeler transfer
- On écrit {**self** transfer(A)}
  - La méthode transfer est choisie dans la classe de l'objet lui-même O_V
  - **self** = l'objet lui-même, une instance de VerboseAccount



A.HARICHE a.hariche@univ-dbkm.dz

Définition de VerboseAccount

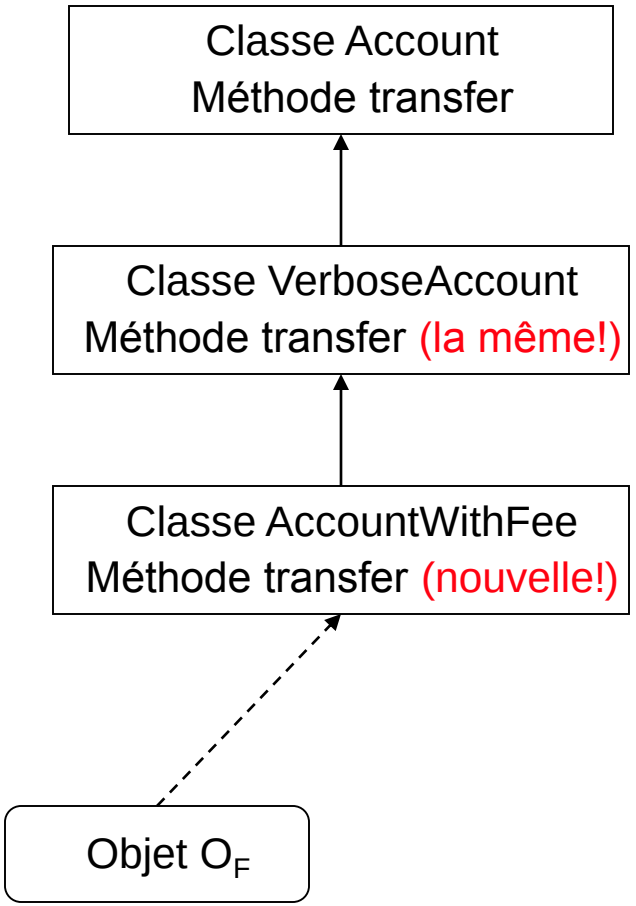
La classe VerboseAccount a les méthodes transfer, getBal et verboseTransfer

```
class VerboseAccount
    from Account
    meth verboseTransfer(Amount)
            {self transfer(Amount)}
            {Browse @balance}
    end
end
```


A.HARICHE a.hariche@univ-dbkm.dz

Lien statique

- Nous allons maintenant redéfinir l'ancienne méthode transfer dans AccountWithFee
- Dans la nouvelle définition de transfer, nous devons appeler l'ancienne définition!
- On écrit VerboseAccount,transfer(A)
  - Il faut spécifier la classe dans laquelle se trouve l'ancienne!
  - La méthode transfer est choisie dans la classe VerboseAccount



A.HARICHE a.hariche@univ-dbkm.dz

Définition de AccountWithFee

```
class AccountWithFee
    from VerboseAccount
    attr fee:5
    meth transfer(Amt)
            VerboseAccount,transfer(Amt-@fee)
    end
end
```

La classe AccountWithFee a les méthodes transfer, getBal et verboseTransfer. La méthode transfer a été redéfinie.

A.HARICHE a.hariche@univ-dbkm.dz

La magie du lien dynamique

- Regardez le fragment suivant:

  A={New AccountWithFee transfer(100)}
  {A verboseTransfer(200)}
- Question: qu'est-ce qui se passe?
  - Quelle méthode transfer est appelée par verboseTransfer?
  - Attention: au moment où on a définit VerboseAccount, on ne connaissait pas l'existence de AccountWithFee
- Réponse: !!

Extension non-conservatrice: danger, danger, danger!

Danger!
Les invariants deviennent faux.

```
class AccountWithFee
    from VerboseAccount
    attr fee:5
    meth transfer(Amt)
            VerboseAccount,transfer(Amt-@fee)
    end
end
```

Invariant:

{A getBal(B)}

{A transfer(S)}

{A getBal(B1)}

% B1=B+S ?

% **Faux!**

A.HARICHE a.hariche@univ-dbkm.dz

Le principe de substitution: une leçon coûteuse

- Dans les années 1980, une grande entreprise a initié un projet ambitieux basé sur la programmation orienté-objet
  - Non, ce n'est pas Microsoft!
- Malgré un budget de quelques milliards de dollars, le projet a échoué lamentablement
- Une des raisons principales était une utilisation fautive de l'héritage. Deux erreurs principales ont été commises:
  - Violation du principe de substitution. Une procédure qui marchait avec des objets d'une classe ne marchait plus avec des objets d'une sous-classe!
  - Création de sous-classes pour masquer des problèmes, au lieu de corriger ces problèmes à leur origine. Le résultat était une hiérarchie d'une grande profondeur, complexe, lente et remplie d'erreurs.

Liens statiques et dynamiques: recapitulatif

- Le but du lien dynamique et du lien statique est de sélectionner la méthode qu'on va exécuter
- Lien dynamique: {**self** M}
  - On sélectionne la méthode dans la classe de l'objet lui-même
  - Cette classe n'est connue qu'à l'exécution, c'est pourquoi on l'appelle un lien *dynamique*
  - C'est ce qu'on utilise par défaut
- Lien statique: SuperClass,M
  - On sélectionne la méthode dans la classe SuperClass
  - Cette classe est connue à la compilation (c'est SuperClass), c'est pourquoi on l'appelle un lien *statique*
  - C'est utilisé uniquement pour la redéfinition (overriding)
  - Quand une méthode est redéfinie, il faut souvent que la nouvelle méthode puisse accéder à l'ancienne

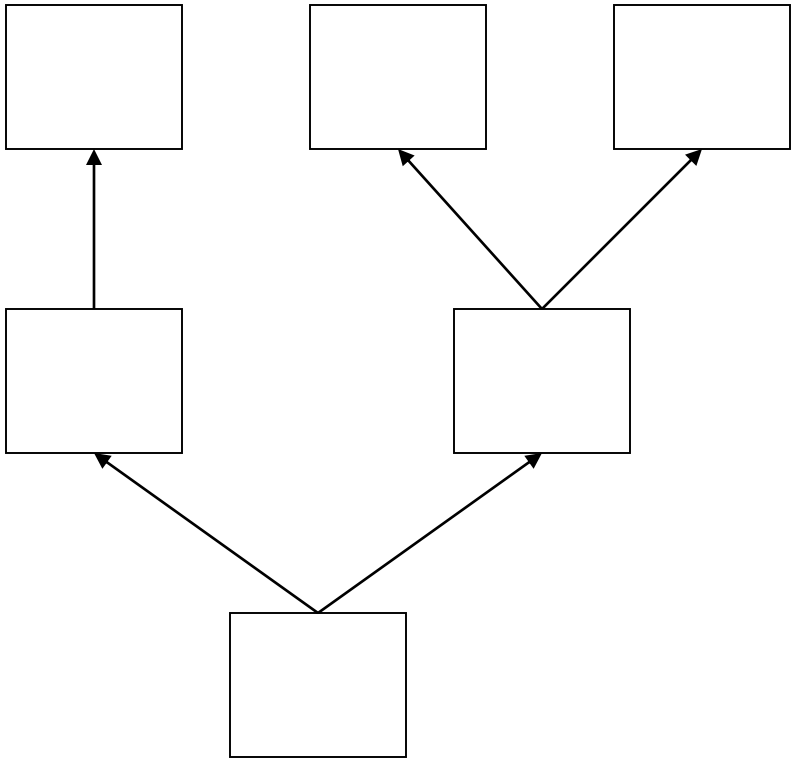
La relation de super-classe

- Une classe peut hériter d'une ou plusieurs autres classes, qui apparaissent après le mot-clé **from**
- Une classe B est appelée super-classe de A si:
  - B apparaît dans la déclaration from de A, ou
  - B est une super-classe d'une classe qui apparaît dans la déclaration from de A

A.HARICHE a.hariche@univ-dbkm.dz

La relation de super-classe

- La relation de super-classe est orienté et acyclique
- Une classe peut avoir plusieurs super-classes
  - Héritage multiple



L'héritage simple et l'héritage multiple

- L'héritage simple = une seule classe dans la clause **from**
  - Beaucoup plus simple à implémenter et à utiliser
  - Java ne permet que l'héritage simple des classes
- L'héritage multiple = plusieurs classes dans la clause **from**
  - Un outil puissant, mais à double tranchant!
  - Voir "Object-oriented Software Construction" de Bertrand Meyer pour une bonne présentation de l'héritage multiple

A.HARICHE a.hariche@univ-dbkm.dz

Résumé

paradigme Orienté Objets

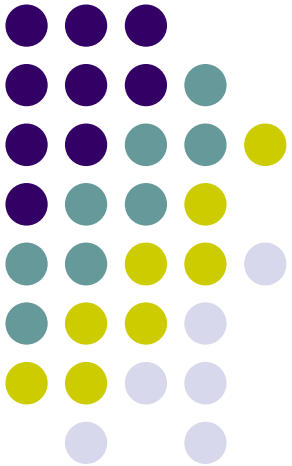
A. HARICHE

University of Djilali Bounaama, Khemis Meliana (UDBKM)

Faculty of Sciences & Technology

Mathematics & Computer Science Department

a.hariche@univ-dbkm.dz



Résumé

- Les objets et les classes
  - L'envoi procédural
  - Une classe comme une fonction pour créer des objets
  - Soutien syntaxique pour les classes
- Le polymorphisme
  - Le principe de la répartition des responsabilités
- L'héritage
  - Le principe de substitution
  - Lien dynamique et lien statique
  - L'héritage simple et l'héritage multiple

A.HARICHE a.hariche@univ-dbkm.dz

Idées à apprendre

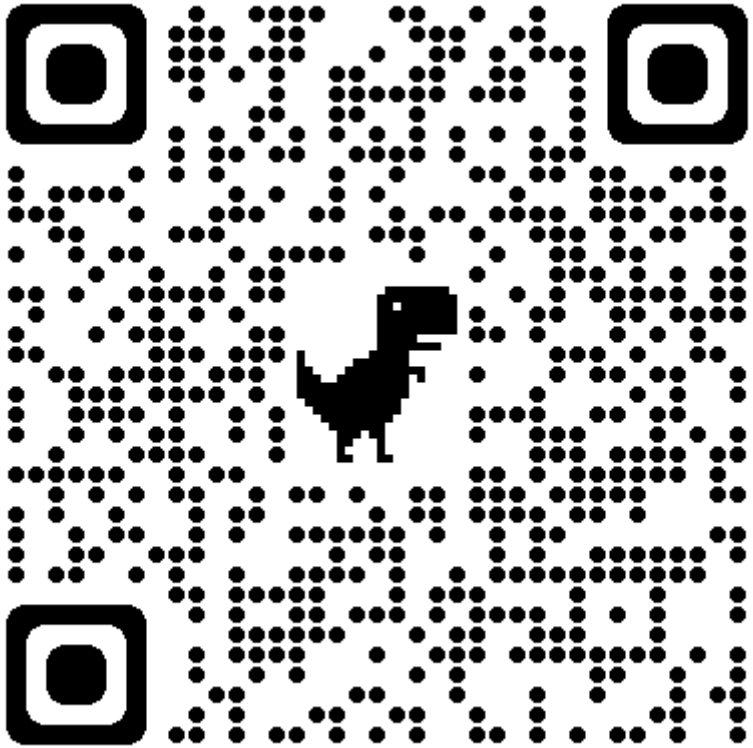
Louv1.1x

- Identifiants et environnements
- Programmation fonctionnelle
- Récursivité
- Programmation invariante
- Listes, arborescences et enregistrements
- Programmation symbolique
- Instanciation
- Généricité
- Programmation d'ordre supérieur
- Complexité et notation Big-O
- Loi de Moore
- Problèmes NP et NP-complets
- Langages du noyau
- Machines abstraites
- Sémantique mathématique

Louv1.2x

- État explicite
- Abstraction des données
- Types de données abstraites et objets
- Polymorphisme
- Héritage
- Héritage multiple
- Programmation orientée objet
- Gestion des exceptions
- Concurrence
- Non-déterminisme
- Ordonnancement et équité
- Synchronisation des flux de données
- Flux de données déterministe
- Agents et flux
- Programmation multi-agents

Lien



https://drive.google.com/drive/folders/1YBCIZzAldeiT19DIfDiREQwP-NAQ1qMN