Abstraction et liaison dans les langages de programmation

LIN2: Paradigmes de programmation

N. Hameurlain

Abstraction: introduction

- L'importance de l'abstraction découle de sa capacité de cacher les détails non pertinents et de l'utilisation des noms pour référencer des objets.
- Exemples: l'abstraction par des procédures, des fonctions, et des modules qui permettent au programmeur de distinguer :
  - entre ce que fait le programme (ceci concerne l'utilisateur du programme) et comment il est implémenté (ceci concerne celui qui écrit le programme).
- essentielle dans la construction des programmes en permettant de gérer la complexité des gros programmes.

Abstraction: principe

- Un objet abstrait est une entité *nommée* qui peut être appelée en mentionnant le nom.
  - Donner un nom à un objet, donne la permission de substituer le nom à l'entité nommée (ou vice versa) sans changer la signification.
- *Exemple*: Les déclarations et les définitions dans les langages de programmation;
- "l'abstrait" est *encapsulé*, c.-à-d., les détails de l'abstrait sont cachés de sorte que le nom soit suffisant pour représenter l'entité (paradigme objets).

Abstraction: terminologie

- Le nommage dans l'abstraction est capturé par les concepts de *liaison*, de définitions et de déclarations, alors que "cacher des détails non pertinents" est capturé par le concept de *l'encapsulation*.
- Parmi les termes utilisés pour faire allusion a un objet « abstrait », on trouve : module, package, sous-programme, fonction, procédure, types de données abstrait, Objet, etc.

Liaison: définition

- Une *liaison* est une association de deux entités.
  - Une définition est une liaison d'un nom à une entité,
  - une déclaration est une définition de ce qui lie un nom à une variable,
  - et une affectation est une liaison d'une valeur et d'une variable.
- Nous pourrions également dire *identificateur* au lieu de *nom*.
  - Une variable est une entité dont la valeur n'est pas fixe mais peut changer. Des noms sont liés aux variables dans la partie déclaration d'un programme.

Liaison: exemples

- Le concept est commun à tous les LP. Les objets qui peuvent être liés aux noms peuvent inclurent:
  - valeurs primitives, valeurs composées, références à des variables, types (niveau définition et déclaration);
  - entre les valeurs et les emplacements mémoire (niveau matériel).
- Paradigme impératif, permet à des noms d'être liés successivement à différents objets,
  - ceci est accompli par l'instruction d'affectation (souvent de la forme *nom: = objet*).
- Programmation fonctionnel et logique pures, « L.P à affectation unique »:
  - les noms ne peuvent pas être réattribués à nouveau.

Types et composition de liaisons

1. **Co-latérale**: a pour principe d'effectuer les liaisons indépendamment les unes des autres et combiner ensuite les liaisons pour produire l'ensemble des liaisons à réaliser.
   - Jamais une liaison peut mettre en référence un nom utilisé dans n'importe quelle autre liaison.
   - pas très utilisées mais se produisent en Scheme et ML.
2. **Séquentiell**e: la plus utilisée pour composer des liaisons ; elle doit exécuter les liaisons dans l'ordre dans lequel elles se produisent.
   - L'effet doit permettre aux liaisons postérieures d'utiliser des liaisons déjà produites plutôt dans l'ordre.

Types et composition de liaisons

- Liaison séquentielle, Exemples :
  - En C/C++ et Pascal, les liaisons de constantes, de variables, et de procédures et de fonctions sont séquentielles.
- Problème (récursivité): pour traiter les définitions mutuellement récursives des fonctions et des procédures, et les construire
  - prévoir la séparation de la signature d'une fonction ou d'une procédure du corps par les moyens des prototypes de fonctions (C/C++ et Pascal).
  - les "déclaration avant référence " ou «forward».

Types et composition de liaisons

- Notes:
  - Les LP qui exigent la "déclaration avant référence " nécessitent des mécanismes spéciaux pour manipuler les références en avant. Pour les types dynamiques de données, la règle est relaxée pour permettre la définition des types pointeurs.
  - La "déclaration avant référence" est souvent choisie pour simplifier la construction du compilateur. Dans Modula-3 et Java le choix a été fait pour simplifier la tâche du programmeur plutôt que celle du compilateur en permettant l'utilisation des références en avant.

3. Récursive est une liaison dans laquelle le nom qui est en train d'être lié est utilisé (directement ou indirectement) dans sa propre liaison.

Liaison et temps obligatoire

Depuis le point de définition du langage jusqu'à l'exécution du programme, le moment où la liaison se produit est dit "temps obligatoire". On distingue quatre temps obligatoires:

1. Conception du Langage. Une grande partie de la structure d'un langage de programmation est fixée à la conception.

- Les types de données, les structures de données, les formes de commande et d'expression, et la structure de programmes
- La plupart des langages de programmation font le nécessaire pour étendre le langage en prévoyant que le programmeur définit des types de données, des expressions et des commandes.

Liaison et temps obligatoire

2. Mise en place du Langage. Quelques dispositifs du langage sont déterminés par l'implémentation.

- un programme qui marche sur un ordinateur peut ne pas fonctionner ou donne des résultats incorrects une fois exécuter sur une autre machine. Ceci se produit quand le matériel diffère dans sa représentation des nombres et des opérations arithmétiques.
- Exemple: le langage de programmation C permet d'accéder à la machine fondamentale et donc les programmes qui dépendent des caractéristiques de la machine fondamentale peuvent ne pas s'exécuter comme prévu une fois déplacer sur une autre machine.

Liaison et temps obligatoire

3. Traduction. La liaison entre le code source et le code objet se produit au moment de la phase de traduction.

- Les variables et les types définis par le programmeur sont un exemple de liaisons produites au moment de la translation du programme.

4. Exécution du programme**.** La liaison des valeurs aux variables, et des paramètres formels aux paramètres effectifs, se produisent pendant l'exécution du programme

Structuration des programmes en Blocs

1. **Monolitic:** Toutes les définitions sont regroupées dans un seul emplacement même si elles sont utilisées dans différents emplacements du programme.
   - La portée de chaque définition est tout le programme;
   - typique au Langage comme le Basic, et les premières version du COBOL;
   - La structure Monolitic est plutôt utilisée dans les cas des « petits » programmes.

Structuration des programmes en Blocs

2. **Linéaire:** un bloc externe entourant un ou plusieurs blocs internes. Des variables peuvent être déclarées à l'intérieur d'un sous-programme, et sont alors locales à ce sous-programme.

- Les noms de sous-programme font partie du bloc externe et leur portée est ainsi le programme entier avec des variables globales
  - Tous les noms de sous-programme et des variables globales doivent être uniques.
- Cette structure est typique du Fortran et C.

Structuration des programmes en Blocs

3. **Emboîtée:** des blocs peuvent être emboîtés à l'intérieur d'autres blocs, et il n'y a aucune restriction sur l'emboîtement des blocs dans le corps (exemple ALGOL).
   - Dans les blocs la visibilité est contrôlée par l'emboîtement.
     - Tous les noms sont visibles (implicitement exporté) aux blocs intérieurement emboîtés mais aucun nom n'est visible(exporté) aux blocs qui l'emboîtent.
     - Dans un bloc, les seuls noms visibles sont ceux qui sont déclarés dans tous les blocs qui l'emboîtent mais pas ceux déclarés dans les blocs emboîtés.

Partitions: définition

Une partition d'un ensemble est une collection d'ensemble disjoints dont l'union est égale a cet ensemble.

- Partionner un programme permet une compilation séparée et un traitement des donnes en Pipe line...
- Il existe un certain nombre de mécanismes pour diviser le code d'un programme. Dans tous les cas le résultat est toujours un seul fichier.
  - Procédure et fonctions sont les plus utilisées.
  - Lorsque la partition est dans des fichiers différents, on parle de Modules.

Partitions: exemples

- Déclarations des données et de code séparément;
- Procédures et fonctions;
- ADT (Abstract Data Type): séparer le "How to do" and le "What to do“;
- Modules (exemple package ADA).

Partitions: combinaison

- Il existe plusieurs mécanismes pour combiner des partitions en un seul programme pour le compiler et l'exécuter.
  - La clause <*include*> offerte dans un grand nombre de L.P permettant d'inclure textuellement le nom d'un fichier dans le programme source.
  - les partitions peuvent être compilées séparément, ensuite il y a la phase *d'éditions de liens* où les modules du programme sont liés ensemble pour l'exécution.
    - toute fonction ou procédure non retrouvée, résulte en une recherche pendant l'exécution: si le ss-programme est trouvé, il est donc exécuté, sinon erreur d'exécution.

Modules

- Unité (programme) de compilation indépendante.
- Consiste en un nombre de définition (de types, variables, fonctions, procédures), qui définit clairement ce qui exporté aux autres modules qui l'utilise.
- Les modules ont un certain nombre d'avantage:
  - facilite le développement parallèle et indépendant;
  - compilation séparée;
  - ré-utilisation de code;
- Les modules doivent répondre à des questions:
  - quel est le but;
  - comment l'atteindre (but).

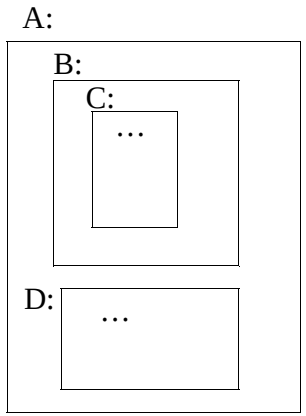
Modules :exemples

- Procédure et fonctions sont des modules simples;
  - leurs signatures décrit ce qu'elles font, et le corps décrit comment elles le réalisent.
- Les Modules sont utilisés pour construire:
  - des librairies, ADTs, classes, interfaces et des implémentations.
  - Dans le cas où un module contient seulement des abstractions de type, on parle de Module de spécification ou de module d'interface.

Modules: règles de portés

- La règle de porté pour les modules définit la relation existante entre les noms des différentes partitions (4 choix):
- Tous les noms locaux sont visibles globalement;
- Tous les noms externes sont visibles localement;
- Seulement les noms locaux explicitement exportés sont visibles globalement;
- Seulement les noms externes, explicitement importés sont visibles localement.

Blocs emboîtés, règles de portés



- A: peut voir tous les noms dans A incluant les noms A, B et D;
  A: peut pas voir les noms dans B, C, ou D
- B: peut voir tous les noms dans A et B incluant les noms A, B, C et D.
  B ne peut pas voir les noms dans C ou dans D;
- C: peut voir tous les noms dans A, B incluant les noms A et B et C;
  C: ne peut pas voir les noms dans D;
- D: peut voir tous les noms dans A et D incluant les noms A, B et D;
  D: ne peut pas voir les noms dans B, ou C.

Procédures et fonctions: Passage de paramètres

- Passage par valeur:
  - Transférer (communiquer) des infos de l'appelant vers un sous programme C, Pascal);
- Passage par résultat:
  - Transférer (recevoir) des infos d'un sous-programme appelé vers l'appelant (ADA);
- Passage par valeur-résultat:
  - Communiquer des infos à un sous-programme où elles seront mises à jour avant d'être ré-envoyées;
    - Valeur- résultat ADA (**IN OUT**),
    - **Par référence** (C++ par exemple).

Passage de paramètres par valeur

- Le paramètre formel joue le rôle d'une variable locale, initialisée à la valeur du par. effectif;

```
Procedure Lire_Ecrire (total: integer);
var symbole:char;
Begin while total > 0 do
      Begin read(symbole);
      write (symbole);
      total:= total-1
      End
End
```

  - Soit <quantité> une var. initialisée a 10; l'effet de l'appel Lire_Ecrire(quantité) sera de passer une copie de la valeur (10) au par. formel <total>;
  - A l'intérieur de cette proc., les modifs apportées à <total> n'affectent que cette copie locale, elles n'ont aucun effet sur le par. effectif <quantité>.

Passage par valeur constante

- Le par. formel est considéré comme une constante locale et ne doit pas varier dans le ss-programme (ADA):

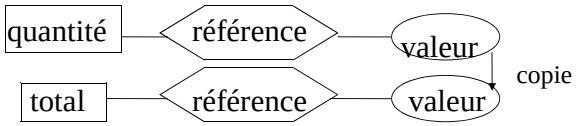
```
Procedure Lire-Ecrire (total:  in integer) is
symbole:character;
decompte:integer:=total;
begin
    while decompte> 0 loop
      get(symbole);
      put(symbole);
      decompte:= decompte-1;
      end loop;
EndLire-Ecrire;
```


Passage par valeur: inconvénients

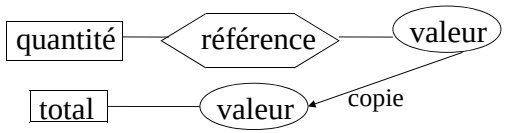
- Quand le par. effectif est un objet composé (tableaux, …), une copie de tout l'objet est crée:
  - Coûteux en mémoire et en temps;
- Pour contourner ce problème, Pascal par exemple passe les paramètres de types tableaux par référence au lieu par valeur.

Passage par valeur (/ constante)

- Passage par valeur:



- Passage par valeur constante :



Passage de paramètres par résultat

- En ADA, sert à implémenter le mode de passage **out** pour les par. de type scalaires;
- Le par. formel joue le rôle d'une var locale non initialisée, qui reçoit des val au cours de l'exécution de la procédure;
- Quand on quitte la proc. la val du par. formel est affectée au par. effectif qui doit être une variable.

Passage de paramètres par résultat: exemple

```
Procedure lire_négatif (nb_négatif: out integer) is
   Nombre: integer;
   begin
        get (nombre);
        while nombre >= 0 loop
        Put_line(« nbre non négatif, réessayer »);
        get(nombre);
        end loop;
   nb_négatif:=nombre;
   end lire_négatif;
```


Passage de paramètres par résultat : remarques

- Un par. passé en OUT ne peut être utilisé dans une expression: nécessité d'avoir une var. locale;
- Dans l'appel Lire-négatif (montant), la val de la var. montant ne reçoit la val de nb_négatif qu'au moment où l'on sort de la proc. Lire_négatif;
- En ADA, l'adresse du par. effectif est calculée à l'entrée de la proc, alors qu'en Algol W, elle n'est calculée qu'à la sortie;
- Ce mode de passage par résultat souffre des mêmes inconvénients que celui par valeur (objet composé, et création locale du par. formel).

Passage de paramètres par valeur/résultat (in out)

- Le paramètre formel joue le rôle d'une variable locale, initialisée à la valeur du par. effectif;
- A l'intérieur de la proc., les modifications apportées au par. formel n'affectent que la copie locale;
- Ce n'est qu'au moment de quitter la proc. que le par. effectif est mis à jour en recevant la valeur finale du par. formel.

Passage de paramètres par valeur/résultat: exemple

```
Procedure mise_à_jour (solde: in out integer) is
   transaction: integer;
   begin
        for j in 1..10 loop
        get(transaction);
        solde:=solde+transaction;
        end loop;
   end mise_à_jour;
```

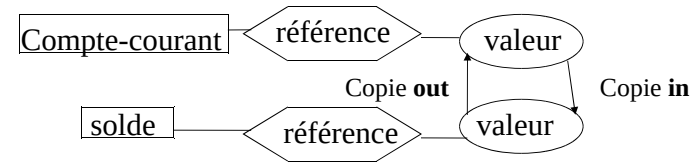
- Dans l'appel mise-à-jour(compte_courant) le par. effectif <compte_courant> n'est mis à jour qu'au moment où l'on quitte la procédure.

Passage de paramètres par référence

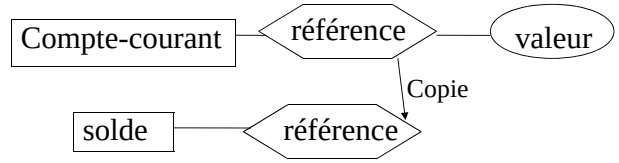
- L'adresse du par. effectif (c à d un pointeur vers l'emplacement contenant ce par) qui est passée au par. formel.
- Toute utilisation du par. formel, dans le ss-pgme, est traitée comme une référence indirecte au par. effectif.
- Exemple précédent: le paramètre effectif <compte_courant> est automatiquement modifié à chaque affectation de <solde>.

Passage par in/out Vs référence

in out:



Référence:



Passage de paramètres par nom

- Utilisé dans ALGOL 60, consiste à une substitution textuelle de chaque occurrence du par. formel par le par. effectif;
- Le par. effectif est passé sans être évalué, et il est recalculé à chaque fois que le par. formel est utilisé au cours de l'exécution;
- Cela introduit éventuellement une (nouvelle) adresse pour différentes utilisations du même par. formel.