L3 Informatique - HLIN603
Partie III - Typage dynamique - Tout objet - Développement Agile - Introduction à la méta-programmation -
avec Smalltalk.
TDs/TPs

Les notes de cours sont en : http://www.lirmm.fr/~dony/notesCours/smalltalkOverview.s.pdf,

Version imprimable en : http://www.lirmm.fr/~dony/notesCours/smalltalkOverview.pdf.

1 Téléchargez Pharo Smalltalk

Allez sur le site http://pharo.org/.

Prenez le temps de vous balader vous un peu sur le site.

Téléchargement : suivez "download latest version" puis "Télécharger Pharo Launcher". Le Launcher est un programme qui va vous permettre de choisir la version du langage. Lancer le *launcher* (en arrière-plan) puis choisissez la version "7.0 stable 64 bits", téléchargez la (clic étoile orange). L'image téléchargée apparait dans la fenêtre droite, sélectionnez la et cliquez sur la fleche verte pour télécharger et installer la machine virtuelle correspondante et ouvrir l'environnement Pharo. Vous pouvez commencer à travailler.

Pour réouvrir l'environnement (par exemple pour le second TP) relancer le launcher et recliquez sur la fleche verte.

2 La syntaxe et les bases avec le tutorial

L'application s'ouvre avec une fenêtre ouverte : "Welcome to Pharo xxx".

Dans cette fenêtre repérer : "learn Pharo" puis "PharoTutorial go." ou "ProfStef go.". Placez le curseur derrière le point, ouvrez le menu contextuel ("command-Clic" ou "control-clic" ou "clic droit" (selon clavier et souris)), choisissez "doIt". Vous obtenez le même résultat avec le raccourci clavier "Cmd-d" ou "Control d" selon votre système. Idem pour "printIt" avec "Cmd-p" et "InspectIt" avec "Cmd-i", utiles partout et tout le temps.

Le tutorial va vous faire aller de fenêtre en fenêtre et vous présenter toute la syntaxe de base et quelques autres choses. Vous avez accès à un livre en ligne : http://www.pharobyexample.org/. Vous avez aussi un Mooc en ligne : http://mooc.pharo.org.

3 L'Environnement, premières indications

Comme indiqué en cours, l'environnement n'est pas un détail mais une part intégrante du concept, permettant dans la vraie vie de programmer "in the large" vite et bien.

- Menu *World* : clic gauche sur fond d'écran. Les items essentiels dans un premier temps sont *Tools-System Browser*, *Tools-Playground* et *Tools-Transcript* si vous voulez afficher des messages ; par exemple (`Transcript show: 'Hello World'; cr.`).
- Chaque sous-genêtre de chaque outil possède un menu contextuel, "Cmd-clic" ou "Ctrl-clic" ou clic bouton droit souris.

— Sauvegarde de vos travaux : *Pharo-save*. Nous ferons plus subtil ultérieurement.

— Ouvrez un `System Browser`, dans le menu contextuel de sa fenêtre en haut à gauche, faites “Find Class” et cherchez la classe *OrderedCollection*. Une fois sélectionnée, regardez la liste de ses méthodes et le classement en catégories. Les catégories sont un concept de l’environnement ; elles n’ont pas d’incidence sur l’exécution des programmes.

— Ouvrez un *Playground*, c’est comme un tableau de travail, entrez des expressions, choisissez “doIt”,“print It” ou “inspectIt” pour exécuter, exécuter et afficher le résultat ou exécuter et inspecter le résultat. Ceci vaut pour toute expression. Toute instruction est une expression.

```
t := Array new: 2.
t at: 1 put: #quelquechose.
t at: 1

c := OrderedCollection new: 4
1 to: 20 do: [:i | c add: i]
"even dit si un nombre est pair"
c count: [:each | each even]
"aller vous ballader sur la classe Collection pour regarder les
itérateurs disponibles"
```

4 Classes, instances, méthodes d’instance

Pour se familiariser, création d’une classe simple. Ouvrez un *System Browser*, dans le menu contextuel de sa fenêtre haut-gauche, faites “Add Package”, donnez lui un nom, par exemple *HLIN603*.

1. A définir la classe `Pile` implantée par composition avec un tableau (`Array`, ce sera ainsi dans le corrigé) ou une `OrderedCollection` qui va bien aussi.

   — Sélectionnez votre package et pour créer la classe, clickez dans la seconde fenêtre du brower, renseignez le *template*, puis “accept”. Ensuite dans le *playground* essayez `Pile new` “inspectIt”.

   ```
   Object subclass: #Pile
       instanceVariableNames: 'contenu index capacite'
       classVariableNames: 'tailleDefaut'
       category: 'HLIN603'
   ```

   — Définissez la méthode `initialize: taille`, équivalent d’un constructeur à 1 paramètre, qui initialise les 3 attributs (dits “variables d’instance”). Essayez ensuite : `Pile new initialize: 5`.

   ```
   initialize: taille
       "la pile est vide quand index = 0"
       index := 0.
       "la pile est pleine quand index = capacite"
       capacite := taille.
       "le contenu est stocké dans un tableau"
       contenu := Array new: capacite.
       "pour les tests, enlever le commentaire quand isEmpty est écrite"
       "self assert: (self isEmpty)."
   ```

— Ecrivez les méthodes : `isEmpty`, `isFull`, `push: unObjet`, `pop`, `top`. Testez les dans le *playground*.

— Pour la rendre compatible avec le *printIt* de l'environnement, définissez la méthode suivante sur la classe. C'est l'équivalent du `toString()` de Java. L'opérateur de concaténation des chaînes est "," (par exemple `'ab' , 'cd'`).

```
printOn: aStream
    aStream nextPutAll: 'une Pile, de taille: '.
    capacite printOn: aStream.
    aStream nextPutAll: ' contenant: '.
    index printOn: aStream.
    aStream nextPutAll: ' objets : ('.
    contenu do: [ :each | each printOn: aStream. aStream space ].
    aStream nextPut: $).
    aStream nextPut: $..
```

— Signalez les exeptions, en première approche, vous écrirez : `self error: 'pile vide'.`.

2. Apprenez à utilisez le débogueur. Insérer l'expression `self halt.` au début de la méthode `push:`. Après lancement, l'exécution s'arrête à ce point, choisissez "debug" dans le menu proposé. Vous voyez la pile d'exécution. Vous pouvez exécuter le programme en pas à pas (les items de menu importants sont "into" et "over" pour entrer, ou pas, dans le détail de l'évaluation de l'expression courante. Le débugger est aussi un éditeur permettant le remplacement "à chaud". Le debugger d'Eclipse a été construit sur le modèle de celui-ci.

5 Composition et méthodes (de type *keyword*) à plusieurs paramètres

Réaliser une classe distributeur de Bonbons (type foire foraine) à n colonnes modélisée comme un tableau de n piles (chaque colonne est représentée par une pile).

— Création d'un distributeur de 2 colonnes : `D:= (Distributeur new) colonnes: 2 taille: 5`.

— Créer une classe `Bonbon` et deux sous-classes `Carambar` et `Malabar` (par exemple).

— Ecrire la méthode `remplir:avec:` telle que `D remplir: 1 avec: Carambar.`, remplisse la colonne 1 avec des instances de la classe `Carambar` ; ceci suppose l'instantiation de la classe passée comme second argument de la méthode `remplir:avec:`.

— Ecrire la méthode `donner:` telle que `D donner: 1.` rende le premier élément de la colonne 1 s'il en reste (donc *aCarambar*, sinon *#YenAPlus*.

6 Jeux de Test

Typage dynamique et développement Agile ou Extrême vont de pair avec la réalisation systématique de test après chaque modification. La définition des tests est souvent préalables à l'écriture du code ; les tests faisant partie de la spécification. Pharo intègre une solution rationnelle pour organiser des jeux de tests dans l'espace (couverture du code) et le temps (rejouer les tests après une modification du code).

1. Appliquez le tutoriel ci-dessous au cas de la pile en créant une classe `TestPile`. Il faut pour cela créer, dans le même package que l'application une sous-classe de `TestCase`, comme indiqué en : http://pharo.gforge.inria.fr/PBE1/PBE1ch8.html.
2. Si vos tests ne passent pas (couleur rouge), le bon outil pour déboguer est le *TestRunner* (menu principal).

7 Méthodes de classes (Take a first walk on the wild side)

Les méthodes de classe s'exécutent en envoyant des messages aux classes (ainsi considérées comme des objets. Par exemple `Date today`. Pour observer ou définir des méthodes de classes, il faut faut cliquer sur le bouton "class" du brower.

- Avant de passer côté *class*, ajouter une variable de classe `tailleDefaut` à la classe PIle (cela se fait côté instance - demandez vous pourquoi ?).
- Définir une **méthode de classe** `initialize` qui fixe à 5 la taille par défaut des piles, valeur 5 stockée dans la variable de classe `tailleDefaut`. Vous devez exécuter cette méthode pour que la variable de classe soit initialisée. De par son nom (`initialize`) cette méthode est reconnue par le browser (voir flêche verte en face du nom). Si vous décidez de la nommer autrement, vous aurez à lancer cette exécution par : (`Pile initialize`).
- Définir sur `Pile` la méthode de classe `new:` pour qu'elle appelle la méthode **d'instance** `initialize:` définie en section 4.
- Définir sur `Pile` la méthode de classe `new` pour qu'elle appelle la méthode **de classe** `new:` définie juste avant.
- Définir une méthode de classe `example` réalisant un exemple de programme utilisant une pile.

8 Héritage - Redéfinitions

Pour expérimenter l'héritage, on va reprendre l'exercice des comptes bancaires fait en C++ et Ocaml.

On traite dans ces cection des classes `Account`, `InterestAccount` et `SecureAccount`.

On traitera de la classe *Bank* dont les instances possèdent une collection de comptes en section 9.

1. Définir `Account` puis ses méthodes `initialize`, `deposit:` , `withdraw:`, et `get`.
   Je vous donne la méthode `printOn:` (notez le joli `self class name`, pour la réutilisation).

```
printOn: aStream
    aStream nextPutAll: 'un ' , self class name , ' de solde : ' .
    balance printOn: aStream.
```

2. **Redéfinition 1**.
   Ouvrez un second browser, visualisez la méthode `=` de la classe `Object`. Repérez le petit triangle bleu à la gauche du nom qui permet d'identifier puis de visualiser toutes ses redéfinitions.
   Redéfinissez la méthode `=` sur `Account` (notez que c'est une méthode binaire, selon la définition du cours *Ocaml*). Notez que vous avez un problème pour accéder au `solde` du compte reçu en paramètre, car les attributs sont protected en *Smalltalk*; comment est-ce solutionné en Java, C++, Ocaml ? Ici il faut utiliser la ruse ou l'introspection (voir la méthode `instVarAt:` de `Object`). Je suggère la ruse.
3. **Redéfinition 2**.
   Redéfinissez `withdraw:` sur `SecureAccount` afin qu'il soit impossible de retirer d'un *secureAccount* si le solde est insuffisant.
4. **Redéfinition 3**.
   Redéfinissez `deposit:` sur `InterestAccount` avec le code ci-dessous, et définissez `depositInterest:` qui ajoute sur le compte receveur 5% de la somme déposée `n`.

```
"méthode d'instance de la classe InterestAccount"
deposit: n
    super deposit: n.
    self depositInterest: n.
```

9 Collection générique (collection de comptes)

1. Définissez la classe `Bank` qui définit les objets possédant un ensemble de comptes stockés dans un attribut `accounts` initialisé avec une *OrderedCollection*.
2. Définissez les méthodes.
   - `add:`, ajoute un compte à une banque ;
   - `balance`, calcule la somme des soldes des comptes du receveur ;
   - `deposit: n`, depose la somme n sur chacun des comptes du receveur.
3. Une bonne habitude à prendre : écrivez la méthode `printOn:`.
4. Définissez et exécutez la méthode de classe `example` ci-dessous.

```
example
    "une méthode de classe de la classe Bank"
    "Bank example" "Selectionnez le commentaire precedent puis doIt"
    | b |
    b := self new initialize.
    b add: (SecureAccount new: 200). "pourquoi les parenthèses ?"
    b add: InterestAccount new.
    b add: (SecureAccount new: 150).
    b deposit: 100.
    ^b
```

5. En l'exécutant, vérifiez que la liaison dynamique fonctionne correctement dans le cas d'envoi de messages a des éléments d'une collection hétérogène, donc que la méthode `depositInterest:` est bien invoquée pour chaque type de compte et qu'elle ajoute des intérêts sur le bon compte.
6. Pour voir plus d'itérateurs, ajoutez à la classe `Bank` les méthodes : `fees`, enlève 5% de frais à tous les comptes ; `min`, rendant le compte ayant le plus faible solde.

   Pour cela examinez la collection des itérateurs (protocole *enumetating)*) sur la classe `Collection`.

10 Contrôle de types à l'exécution

Typage dynamique ne signifie pas abence de types. Les types sont présents à l'exécution où tout objet possède un type défini par sa classe (tout est objet et tout objet est instance d'une classe). Il est donc possible de programmer des contrôles si on le souhaite [1].

1. Il est à noter que ce n'est pas dans la pratique avec de tels langages où la qualité des programmes découle des tests systématiques et du fait qu'il n'est pas si fréquent de placer par erreur un chien dans une collection de chats. L'intérêt de l'exercice est intéressant de voir comment manipuler les types dans un langage tout objet.

1. Exemple : écrire une classe `PileTypée`, sous-classe de la classe `Pile` qui permet d'imposer le type des éléments empilés. En pratique il faut pour cela utiliser un attribut pour stocker la classe des éléments à empiler, une méthode d'instance (et une de classe) pour l'initialiser, et une redéfinition de la méthode `push:`.
   Utilisation :

```
p := PileTypee de: Bonbon.
p push: Carambar new.
p push: 22 "--> exception, cette pile ne peut contenir que des Carambar"
```

2. (travail post TP) Etendre l'exercice précédent au cas du distributeur en le reprogrammant pour en faire une version non contrainte et une version contrainte utilisant des piles typées.

11 Tout est objet : `nil` comme la liste ou l'arbre vide

`nil` est la valeur par défaut contenue dans tout mot mémoire géré par la machine virtuelle *Smalltalk*. Toute variable ou attribut ou case de tableau non initialisée contient `nil`. En *Smalltalk*, `nil` est aussi un objet, l'unique instance de la classe `UndefinedObject` (dont le nom me semble faire peu de sens puisque `nil` est parfaitement défini mais c'est un point de vue personnel). On peut donc envoyer des messages à `nil` qui correspondront à des méthodes définies sur `UndefinedObject`.

Par ailleurs, cet exercice est l'occasion de bien noter que l'envoi de message réaliste un test de type implicite, à comparer à un test de type explicite réalisé dans un *case-switch*).

1. En utilisant cette information, programmez la classe `ArbreBinaireDeRecherche` (ou `ABR`), selon l'énoncé du TD2 Ocaml - question 4, en définissant toutes les méthodes relatives aux arbres vides sur la classe `UndefinedObject`. (pas d'obligation à traiter le cas du *remove* si vous n'avez pas de temps, il relève plus du cours d'algorithmique scricto sensu).
   NB : **Environnement**. Si HLIN603 est le nom de votre package de travail en TP, définissez la classe `ABR` dans le package nommé *HLIN603-ABR* et définissez les méthodes sur la classe *UndefinedObject* dans le protocole (ou catégorie) de méthodes nommé **HLIN603-ABR*. Ainsi vous pourrez tout visualisez au même endroit (le package HLIN603) dans le browser.
2. Définissez un itérateur `do:` pour les arbre binaires de recherche.

12 Méta-programmation et IDM

12.1 Inspections

Inspectez la classe `Pile`, puis son dictionnaire des méthodes, puis sa méthode `push:`.

Inspectez le résultat de l'expression : `Pile compiledMethodAt: #push:`.

Trouvez la classe sur laquelle est définie la méthode `compiledMethodAt:`.

12.2 Un programme qui fabrique un programme

Créer sur une classe *CreateCpt*, la méthode de classe `do` suivante, testez la et finissez la.

```
do
    "fabriquer une classe et ses méthodes par programme"
    "detruisez la classe Cpt si elle existe avant de relancer l'exécution"
    "CreateCpt do"

    | newClass unCpt initializeMethod |

    "creer la classe Cpt"
    Object
        subclass: #Cpt
        instanceVariableNames: 'val'
        classVariableNames: ''
        package: 'HLIN603'.

    "rérérencer la classe Cpt dans une variable"
    newClass := Smalltalk classNamed: #Cpt.

    "creer une instance"
    unCpt := newClass new.

    "fabriquer le code et compiler la méthode initialize de Cpt"
    initializeMethod := OpalCompiler new
                source: 'initialize\ ^val := 0' withCRs;
                class: newClass;
                compile.

    "ajouter la méthode à la classe"
    newClass addSelector: #initialize withMethod: initializeMethod.

    "executer la méthode et verifier la valeur rendue"
    self assert: (unCpt initialize == 0).

    "à vous de continuer"
    "commencer par programmer en début de méthode : détruire la classe Cpt si elle existe"
```

12.3 Accéder à la pile d'Exécution de la machine virtuelle

Implantez sur la classe Symbol, les méthodes `catch` et `returnToCatchwith:` suivantes qui accèdent à la pile d'exécution via la pseudo-variable *thisContext*.

```
!Symbol methodsFor: 'catch-throw'!

catch: aBlock
    "execute aBlock with a throw possibility"
    aBlock value.

returnToCatchWith: aValue
        | catchMethod currentContext |
        currentContext := thisContext.
        catchMethod := Symbol compiledMethodAt: #catch:.
```

```
        [currentContext method == catchMethod and: [currentContext receiver == self]]
                whileFalse: [currentContext := currentContext sender].
        currentContext return: aValue.
        ^aValue
```

Listing (1) *– version Pharo-6*

Exemple d'utilisation dans le *playground* :

```
^#Essai catch: [
        Transcript show: 'a';cr.
        Transcript show: 'b';cr.
        Transcript show: 'c';cr.
        #Essai returnToCatchWith: 22.
        Transcript show: 'd';cr.
        33]
```

Listing (2) *– should display a b c (not d) in the Transcript and return 22 (not 33)*

Vous pouvez ensuite lire la méthode `signal` (équivalent de `throw`) de la classe `Exception`.

12.4 Réflection de comportement, les méta-objets pour modifier l'exécution

```
"la méthode ast rend l'arbre de syntaxe abstraite du receveur"
node := (Pile compiledMethodAt: #push:) ast.

"Il est possible d'associer un méta-objet à tout noeud de l'arbre de syntaxe''
link := MetaLink new metaObject: (Object new); selector: #intercepte.
node link: link.

(Pile new: 3) push: 33.
```

Listing (3) *– "Reflection in Pharo : Beyond Smalltak", Marcus Denker*

12.5 Création d'une nouvelle méta-classe

Modifiez la classe `Pile` pour qu'elle mémorise la liste de ses instances dans une collection stockée dans une variable d'instance de la métaclasse, et dotée d'un accesseur sur cette collection.

13 Tester les fermetures

1. Testez les exemples du cours relatifs aux blocks.
2. Ecrivez sur une classe `Counter` une **méthode de classe** `create` :

```
create
    | x |
    x := 0.
    ^ [ x := x + 1 ]
```

3. Appelez deux fois la méthode et stockez les valeurs rendues dans 2 variables. Exécuter plusieurs fois les blocks contenus dans ces deux variables.

```
c1 := Counter create.
c1 value.
c1 value.
c2 := Counter create.
c2 value.
```

4. Inspectez ces deux variables en faisant le lien avec la définition de la classe `BlockCLosure`.

14 Héritage et Traits

Lisez la section 3.11 du cours.

On souhaite pouvoir trier la collection de comptes d'une banque par la balance de ses comptes, du plus bas au plus élevé. On définit la méthode `sort` sur Banque avec comme code : `accounts sort`.

Sachant que accounts est une `OrderedCollection`, que `OrderedCollection` hérite de `SequenceableCollection`, que `SequenceableCollection` utilise le trait `TSortable` dont le code est ci-dessous, déduisez en ce qu'il faut faire sur la classe `Account` pour obtenir le résultat demandé.

```
Trait named: #TSortable
    uses: {}
    category: 'Collections-Abstract'

!TSortable methodsFor: 'sorting' !
sort
    "Sort this collection into ascending order using the '<=' method."
    self sort: [:a :b | a <= b]
```