M2 Informatique - HMIN305
TD-TP : Métaprogrammation en Smalltalk et Clos

1 Rappels pour l'utilisation de Pharo-Smalltalk

1.1 Téléchargez Pharo-Smalltalk

Télécharger la dernière version de Pharo Smalltalk (v6) à partir de `http://pharo.org/` (balladez vous un peu sur le site), désarchivez selon le système que vous utilisez. La commande exécutable est `Pharo` dans le dossier obtenu. Exécutez la en ligne de commande dans un terminal ou avec l'interface graphique.

1.2 La syntaxe et les bases avec le tutorial (si besoin)

L'application s'ouvre avec une fenêtre ouverte : "Welcome to Pharo xxx". Dans cette fenêtre repérer : "PharoTutorial go." ou "ProfStef go" (avec *Pharo5*, c'est dans l'onglet "Learn Pharo Smalltalk". Placez le curseur derrière le point, ouvrez le menu contextuel"command-Clic" ou "control-clic" ou "clic droit" (selon 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 un Mooc en ligne : `http://mooc.pharo.org`.

1.3 L'Environnement, premières indications (si besoin)

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 sur fond d'écran. Les items essentiels dans un premier temps sont *System Browser*, *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"
- Sauvegarde de vos travaux : *menuWorld-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.

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

5  c := OrderedCollection new: 4
6  1 to: 20 do: [:i | c add: i]
7  "even dit si un nombre est pair''
8  c count: [:each | each even]
9  "aller vous ballader sur la classe Collection pour regarder les
10 itérateurs disponibles''
```

1.4 Classes, instances, méthodes d'instance

Ouvrez un *System Browser*, dans le menu contextuel de sa fenêtre haut-gauche, créez un package "Add Package", donnez lui le nom `HMIN305`.

1. Redéfinissez la classe *Pile* selon la spécification ci-dessous, ou bien rechargez la directement à partir de la page du cours (`http://www.lirmm.fr/~dony/enseig/MR/index.html`).
2. A définir la classe `Pile` implantée avec un `Array` (ce sera ainsi dans le corrigé) ou une `OrderedCollection` qui va bien aussi.
   — Sélectionnez votre package et pour créer la classe, cliquez dans la seconde fenêtre du brower, renseignez le template (code ci-dessous), puis "accept". Ensuite dans le *playground* essayez `Pile new` "inspectIt".

```
Object subclass: #Pile
    instanceVariableNames: 'contenu index capacite'
    classVariableNames: 'tailleDefaut'
    category: 'HMIN305'
```

   — Définissez la méthode `initialize:`, équivalent d'un constructeur à 1 paramètre, qui initialise les 3 attributs (dites 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*, définissez la méthode suivante sur la classe. C'est l'équivalent du `toString()` de Java.L'opérateur de concaténation 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'.`.
3. 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.
4. Ecrire une méthode `grow` qui double la capacité d'une pile.

1.5 Jeux de Test

Pharo intègre une solution rationnelle pour organiser des jeux de tests systématique dans l'espace (couverture du code) et le temps (rejouer les tests après une modification du code).

1. Appliquez ce tutoriel aux 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).

1.6 Définition de Méthodes de classes

Smalltalk permet l'utilisation de méta-classes pour programmer le niveau de base, il ne s'agit donc pas conceptuellement de méta-programmation, même si cela en est de facto. C'est en premier lieu une façon rationnelle de réaliser une version claire des "static" de C++ et Java.

Les méthodes de classe sont définies sur la classe de la classe et 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 exécutée. 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 (`Pile initialize`).
— Redéfinir sur `Pile` la méthode de classe `new` pour qu'elle appelle la méthode **d'instance** `initialize` définie en section 1.4.
— Définir une méthode de classe `example` réalisant un exemple de programme utilisant une pile ;

2 Utilisation des méta-objets

2.1 Listes et Arbres - Implantation avec UndefinedObject

1. `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`.

   En utilisant cette information, programmez la classe `LinkedListe` (ou `List`), en définissant toutes les méthodes relatives aux listes vides sur la classe `UndefinedObject`.
2. Définissez un itérateur `do:` pour les listes.
3. Si vous le souhaitez refaites l'exercice pour Arbre Binaire de Recherche.
4. **Environnement**. Si HMIN305 est le nom de votre package de travail en TP, définissez la classe `LinkedList` dans le package nommé *HMIN305-List* et définissez les méthodes sur UndefinedObject dans le protocole (ou catégorie) **HMIN305-List*. Ainsi vous pourrez tout visualisez au même endroit dans le browser.

2.2 S'ouvrir aux fermetures

2.2.1 Vérifier qu'une fermeture est accessible en lecture/écriture

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.
4. Exécuter plusieurs fois les blocks contenus dans ces deux variables.
5. Inspectez ces deux variables en faisant le lien ave la définition de la classe `BlockCLosure`.

2.2.2 Implantez de nouvelles structures de contrôle

Ajouter au système les méthodes `ifNotTrue:` et `ifNotFalse:` sur les classes de booléens, `repeatUntil:` sur la classe `BlockClosure`.

2.3 Les méta-Objets de base et l'Introspection

1. En utilisant l'inspecteur, inspectez la classe `Pile`, puis son dictionnaire des méthodes, puis sa méthode `push:`.
2. Etudier les classes `Object`, `Behavior`, `ClassDescription` et `Class` et leurs méthodes pour l'introspection (protocole *accessing*).
3. Inspectez par exemple le résultat de l'expression : `Pile compiledMethodAt: #push:`.
   Trouvez la classe sur laquelle est définie la méthode `compiledMethodAt:`.
4. En utilisant les protocoles d'introspection, écrivez **un inspecteur d'objet** (non graphique) capable d'afficher dans le `Transcript`, pour tout objet, les noms et valeurs de ses attributs.

2.4 Les méta-Objets pour l'accès au compilateur et aux méthodes compilées

2.4.1 Programmer une transformation de modèle

La classe `Pile` que je vous ai passée possède une méthode `grow`. On souhaiterait la refactoriser sur une sous-classe `PileGrossissante`, les piles standard n'étant alors plus capables de grossir.

— Créez une classe prétexte à l'exercice nommée `IDM`.
— Créez sur `IDM` une méthode de classe *idmPile* qui :
  — créé unesous-classe de `Pile` nommée `PileGrossissante`,
  — enlève la méthode `grow` de la classe `Pile` et la met sur la classe `PileGrossissante` (voir la méthode `addSelector:withMethod:` de la classe `Behavior`),
  — créé une méthode `push:` sur `PileGrossissante` dont le code appelle `grow` si la pile est pleine.

2.4.2 Programmer la classe Cellule d'un tableur

Implanter la classe `Cellule` donnée en cours. Une cellule représente une case d'une feuille de calcul d'un tableur. Une cellule possède un attribut `valeur` et un attribut `formule`. Une formule peut-être n'importe quelle expression *Smalltalk*. La méthode `formule:` reçoit une chaîne en argument, la compile puis stocke le

résultat dans dans l'attribut `formule`. La méthode `executeFormule`, exécute la formule et stocke la valeur dans l'attribut `valeur`.

2.5 Méta-objets pour accéder à la pile d'exécution

— Implantez sur la classe `Symbol` les méthodes `catch` et `returnToCatchwith:` données dans le cours.
— Etudiez l'implantation en Smalltalk du système de gestion des exceptions, en premier lieu la classe `Exception` et sa méthode `signal` (équivalent conceptuel du *throw* de *Java*), mais bien sûr ici throw est une vraie méthode.

3 Les méta-classes en Smalltalk

Au delà de l'utilisation de base des méta-classes en Smalltalk (cf. paragraphe 1.6), qui ne nécessitent en fait pas une utilisation du terme "méta-classe", nous allons maintenant les considérer en tant que telles.

1. Définissez la classe Citoyen. Comment définiriez vous l'attribut `Président` ? variable d'instance, variable d'instance partagée (variable de classe), ou variable d'instance de la méta-classe.
2. Faites en sorte qu'une classe, par exemple la classe `Pile`, devienne une *MemoClass*. Ceci revient à définir sur la métaclasse `Pile class` (automatiquement créée) le comportement adéquat pour que son instance `(Pile)` soit une MemoClass. Ainsi il sera possible de demander à `Pile` la liste de ses instances (`Pile instances`).

   A noter la différence subtile avec la question qui serait "Définissez une nouvelle méta-classe MemoClass", ce qui n'est pas simple du tout avec le système de création automatisé des méta-classes de Smalltalk.
3. Définir la classe *Dieu*. Modifier sa classe afin qu'il soit impossible d'en créer plus de n instances (n : 0..n selon votre opinion).
4. Définissez les classes `Chien` et `Chat` comme sous-classes de `Animal`. Faites de `Animal` une classe abstraite.
5. Après avoir fait de `Animal` une classe abstraite, créez une instance de `Chat`. Résolvez le problème et discuter en conséquenses des avantages et inconvénients des méta-classes explicites.

4 Les méta-classes en Clos

1. Implanter la méta-classe `MemoClass` en **Clos**.

   On supposera qu'il existe une classe `MemoObject`, sous-classe de *Object* et que toute `MemoClass` doit être une sous-classe de `MemoObject`.
2. La compatibilité des méta-classes ... à suivre

... à suivre ...