M2 Informatique - HAI931I
TD-TP : Métaprogrammation en Pharo, Clos, Python

1 Rappels pour l'utilisation de Pharo

Voir : `https://pharo.org/`.

1.1 Téléchargez Pharo

Téléchargement : suivez "download latest version" puis "Télécharger Pharo Launcher". Une fois téléchargé, exécutez le programme *PharoLauncher*. Le Launcher est un programme qui permet de choisir la version des bibliothèques du langage ou même différentes applications connexes comme le cours en ligne MOOC.

Cliquez sur *new* (étoile orange) et choisir la version stable "8.0 stable 64 bits", puis téléchargez la *(bouton create-image)*. Une fois l'image téléchargée, elle apparait dans la fenêtre résultante ; sélectionnez la et cliquez sur la fleche verte pour télécharger et installer la machine virtuelle correspondante et les bibliothèques (on appelle cela l'image) et ouvrir l'environnement *Pharo*. Vous pouvez commencer à travailler.

Pour réouvrir l'environnement (pour votre travail personnel puis le second TP) relancer le *launcher*, sélectionner directement l'image contenant votre travail et recliquez sur la fleche verte (bouton *launch*).

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". 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.
```

```
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''
```

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 `HAI931`.

1. Définissez la classe *Pile* selon la spécification ci-dessous, ou bien si vous savez déjà la programmer, rechargez la directement à partir de la page du cours (`http://www.lirmm.fr/~dony/enseig/MR/index.html`). Si vous ne savez pas, faites l'exercice.
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: ' HAI731 '
```

   — 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 ].
```

```
8       aStream nextPut: $).
9       aStream nextPut: $..
```

— Signalez les exeptions, en première version, 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

Pharo 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-side" du brower.

— Avant de passer côté *class*, ajouter une variable de classe `tailleDefaut` à la classe PIle (cela se fait côté instance -).
— 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 **??**.
— Redéfinir sur `Pile` la méthode de classe `new:`, à 1 paramètre, pour qu'elle appelle la méthode d'instance `initialize:` en lui transmettant l'argument qu'elle a reçu.
— Définir une méthode de classe `exemple` réalisant un programme utilisant une pile ;

2 Utilisation des méta-objets

2.1 Listes et Arbres - Implantation avec le méta-objet "UndefinedObject"

1. `nil` est la valeur par défaut contenue dans tout mot mémoire géré par la machine virtuelle *Pharo*. Toute variable ou attribut ou case de tableau non initialisée contient `nil`. En *Pharo*, `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 `HAI931` est le nom de votre package de travail en TP, définissez la classe `LinkedList` dans ce package. Puis, dans un autre *browser* définissez les méthodes relative aux listes sur la classe `UndefinedObject` dans un protocole (catégorie de méthode - voir l'item de menu "classify" dans le menu des méthodes) quelconque. Puis choisissez "convert-to-extension" puis `HAI931` pour transformer ce protocole en une extension du package HAI931. En synthèse, on ajoute ce protocole de la classe UndefinedObject au package `HAI931`.

   Ainsi ce protocole devient visible quand on affiche ce package.

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 sur la classe `Object`, une méthode `exoInspect` qui réalise un **un inspecteur d'objet** (non graphique), capable de rendre une chaîne de caractères, indiquant pour tout objet, les noms et valeurs de ses attributs.

   Exemple :

```
Pile new initialize: 4; push: 33; exoInspect. "doit rendre"
'contenu: #(33 nil nil nil)
index: 1
capacite: 4
'
```

2.4 Les méta-Objets pour l'accès aux classes, 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 `RefactorPile`.
- Créez sur `RefactorPile` une méthode de classe *do* 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.

Pour réaliser l'exercice, il est conseillé d'avoir une variable de classe `saveGrow` qui contiendra une copie de la méthode grow initialement définie sur `Pile`.

Voici l'initilisation de cette variable dans une méthode de classe de `RefactorPile` :

```
initialize
    saveGrow
        ifNil: [ saveGrow := [ Pile compiledMethodAt: #grow ]
                on: KeyNotFound
                do: [ self error: 'Definir grow sur Pile avant de commencer svp' ] ]
```

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 *Pharo*. 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`. Dans la chaîne, il peut y avoir des références aux cellules. Dans la formule `=C4+1`, `C4` doit donner accès à la valeur de la cellule correspondante.

Le code donné dans le cours est issu d'une ancienne version ; il faut l'adapter avec les nouveaux méta-objets présents dans Pharo.

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 Pharo 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 Pharo

Au delà de l'utilisation de base des méta-classes en Pharo (cf. paragraphe **??**), 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.
3. Définir la classe *SalleCours* d'une université. Modifier sa classe afin qu'il soit impossible d'en créer plus de n instances (n étant le nombre de salles disponibles).
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.
6. Mettez un point d'arrêt dans la méthode `subclass: t instanceVariableNames: f classVariableNames: d poolDictionaries: s package: cat` de la classe `Class`.

   Créez une nouvelle classe et observez tous le processus de création d'une classe. Essayez de trouver où et comment la métaclasse automatiquement associée à cette nouvelle classe est créée.

4 Les méta-classes et la méta-programmation en Clos

Exercices à réaliser en utilisant l'interpréteur clisp installé sur vos machines.

1. Reprenez les exemples de base présentés dans le cours.
2. Pour tester les fonctions génériques et les multi-méthodes, vous pouvez refaire l'exercice (partie A) de location de produits vu en master1 `http://www.lirmm.fr/~dony/enseig/IL/TD-TP-DD-Decorateur.pdf`. Le *dispatch* multiple et les multi-méthodes de CLOS réalisent le double-*dispatch* sans qu'il soit besoin d'envoyer 2 messages.
3. Implanter la méta-classe `MemoClass` en **Clos** (donnée en cours), créez une mémo-classe, par exemple créez `Pile` comme instance de `MemoClass`. Vérifiez le bon fonctionnement du résultat en récupérant la liste des instances de Pile après 2 instantiations.
4. On reprend l'exercice MemoClass en le complexifiant (voir dernière partie du cours). On supposera qu'il existe une classe `MemoObject`, sous-classe de *Object* et que toute `MemoClass` doit être une sous-classe de `MemoObject`. Gérer la compatibilité.
5. reprendre l'exercice des classes `Animal`, `Chien` et `Chat`. `Chat` est une classe concrète, sous-classe de `Animal` qui est abstraite. Avec les méta-classes explicites, ceci ne pose plus de problème.
6. reprendre l'exercice de l'inspecteur : en utilisant les protocoles d'introspection, écrivez **un inspecteur d'objet** (non graphique) capable d'afficher, pour tout objet, les noms et valeurs de ses attributs.

... à suivre ...