M2 Informatique - HAI931I
TD-TP séances 1 à 9 : Métaprogrammation en Pharo, Clos, Python

1 Séance 1 : Bases de Pharo-Smalltalk

1.1 Rappels (ou pas) pour l'utilisation de Pharo

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

1.1.1 Téléchargez Pharo

Téléchargement : suivez le bouton"download" 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.

Ces TPs ont été écrits pour la version "8.0 stable 64 bits". Cliquez sur *new* (étoile orange) et choisir la version deprecated "8.0 stable 64 bits", puis téléchargez la *(bouton create-image)*.

A chaque numéro d'image correspond une nouvelle version du système. Les nouvelles versions changent l'environnement et les outils associés, les concepts que nous étudions dans ce cours ne changent pas d'une version à l'autre. Si vous voulez, vous pouvez prendre la dernière image de Pharo (10 ou 11), tous les code donnés dans l'énoncé et dans les corrigés fonctionneront, par contre l'environnement (emplacement des menus par exemple) pourra être différent de ce qui est écrit ici.

Une fois l'image téléchargée, elle apparait dans la fenêtre résultante de *pharo-launcher* ; sélectionnez la et cliquez ensuite 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*.

1.1.2 Première chose à faire

Quand vous êtes dans la fenêtre Pharo, vous pouvez commencer à travailler. La première chose à faire est de localiser dans le menu *Pharo* (en haut à gauche), l'item "save" pour sauver ce que vous aurez fait, puis l'item "quit" pour quitter. Après un "save",

Une fois quitté, pour réouvrir le logiciel une seconde fois, relancer l'application *pharo-launcher*, cette fois sélectionnez directement l'image contenant votre travail et recliquez sur la fleche verte (bouton *launch*).

1.1.3 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.
```

```
t at: 1 put: #quelquechose.
t at: 1.

c := OrderedCollection new: 4
1 to: 20 do: [:i | c add: i].
c

"even dit si un nombre est pair''
c count: [:each | each even].

"aller vous ballader sur la classe Collection pour regarder les
itérateurs disponibles''
```

1.1.4 La syntaxe des expressions 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`.
Aussi :
`http://rmod-pharo-mooc.lille.inria.fr/MOOC/PharoMOOC/Week1/C019-W1S05-PharoSyntaxInANutshell.pdf`

1.2 Création d'une classe : Compteur

1.
2. Ouvrez un *System Browser*, dans le menu contextuel de sa fenêtre haut-gauche, créez un package "Add Package", donnez lui le nom `TD-HAI931` (important).
3. Dans le package `TD-HAI931`, définissez la classe *Compteur*.
   Si vous êtes perdus, le code demandé se trouve
   — dans le cours Pharo section 1.5 : `http://www.lirmm.fr/~dony/notesCours/smalltalkOverview.pdf`. (Attention quand vous copiez/collez du pdf, toutes les caractères de ponctuation sont possiblement incorrectement collés à l'arrivée).

   ```
   Object subclass: #Compteur
       instanceVariableNames: 'valeur' "noms des attributs"
       classVariableNames: '' "aucun attributs static"
       category: 'TD-HAI931' "package"
   ```

4. Dans le *playground* essayez `c := Comteur new` "inspectIt".
5. Dans le *playground*, envoyez des messages à c. Essayez "printIt".

1.3 Création d'une classe : Pile

1. Dans le package `TD-HAI931`, définir la classe `Pile` implantée avec un attibut qui est soit un tableau (`Array` - ce sera ainsi dans le corrigé) ou une `OrderedCollection` qui va bien aussi.
   Si vous êtes perdus, le code demandé ici se trouve
   — dans le corrigé du TP : `corrigesTPs/CPile.st`
   Quand les corrigés sont des codes, pour les utiliser :

— solution 1 : ouvrir les fichiers avec un éditeur de texte et faire des copier/coller vers l'éditeur Pharo (dans le *browser)* ;
— solution 2 : sauvegarder le fichier avec son extension ".st", puis en Pharo, ouvrir le fichier avec l'outil *FileBrowser* (menu *System)* puis le charger avec *FileIn* ou *Install*. Le code chargé (installé) sera ensuite visible dans le *browser*.

2. Sélectionnez votre package et pour créer la classe, cliquez dans la seconde fenêtre du brower, renseignez le template (code ci-dessous), puis "accept".

```
Object subclass: #Pile
    instanceVariableNames: 'contenu index capacite' "noms des attributs"
    classVariableNames: 'tailleDefaut' "noms des attributs statics"
    category: 'TD-HAI931' "package"
```

3. Dans le *playground* essayez `Pile new` "inspectIt".
4. Définissez la méthode `initialize:`, équivalent d'un constructeur à 1 paramètre, qui initialise les 3 attributs (dites variables d'instance). Essayez ensuite dans le *playground* : `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)."
```

5. Dans le *playground* essayez `Pile new initialize: 5` "inspectIt".
6. Si vous avez terminé, passez à la séance 2.

2 Séance TD-TP 2

2.1 Classes, instances, méthodes d'instance - bases

Ouvrez un *System Browser*, dans le menu contextuel de sa fenêtre haut-gauche, retrouvez le package `TD-HAI931` et sélectionnez la classe `Pile` que vous avez commencé à écrire.

1. Pour votre classe Pile, écrivez les méthodes : `isEmpty`, `isFull`, `push: unObjet`, `pop`, `top`.
   Puis testez les dans le *playground* en envoyant les messages correspondant à une pile. N'oubliez pas les "." comme séparateurs d'instructions.

```
  c := Pile new initialise: 5.
  c isEmpty.
  c push: 33. "puis ré-excuter la ligne précédente"
  c push: c. "typage dynamique, on met ce que l'on veut dans la pile"
  c push: 'une chaine'.
  c isFull.
  c top.
  c pop.
  c
```

2. 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'`). Envoyer le message `printOn: uneStream` a un objet, ajoute sa représentation à la stream.

```
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: $.
```

3. Signalez les exeptions, en première version, vous écrirez : `self error: 'pile vide'.`.
4. 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. Ecrire une méthode `grow` qui double la capacité d'une pile.

2.2 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 `PileTest`. 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).

2.3 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 2.1.
— 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 ;

3 Seance TD-TP 3 - Utilisation de méta-objets en Pharo

3.1 S'ouvrir aux fermetures

3.1.1 Implantez une mini-classe avec une fermeture (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 |
```

```
3       x := 0.
4       ^ [ x := x + 1 ]
```

3. Appelez deux fois la méthode et stockez les valeurs rendues dans 2 variables.
4. Exécuter plusieurs fois les lambdas (blocks) contenues dans ces deux variables.
5. Inspectez ces deux variables en faisant le lien ave la définition de la classe `BlockCLosure`.

3.1.2 Implantez de nouvelles structures de contrôle

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

3.2 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
'
```

4 Séance TD-TP 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.