

**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 TP 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 apparaît 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.
```

```

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 c

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

12 "aller vous ballader sur la classe Collection pour regarder les
13 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).

---

```

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

```

---

4. Dans le *playground* essayez `c := Compteur 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 attribut 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*.
- Sélectionnez votre package et pour créer la classe, cliquez dans la seconde fenêtre du browser, renseignez le template (code ci-dessous), puis "accept".

---

```

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

```

---

- Dans le *playground* essayez `Pile new` "inspectIt".
- 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`.

---

```

1 initialize: taille
2   "la pile est vide quand index = 0"
3   index := 0.
4   "la pile est pleine quand index = capacite"
5   capacite := taille.
6   "le contenu est stocké dans un tableau"
7   contenu := Array new: capacite.
8   "pour les tests, enlever le commentaire quand isEmpty est écrite"
9   "self assert: (self isEmpty)."

```

---

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

## 2 Séance TD-TP 2 et 3 - Un programme jouet de base

### 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.

- 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.

---

```

1 c := Pile new initialise: 5.
2 c isEmpty.
3 c push: 33. "puis ré-exécuter la ligne précédente"
4 c push: c. "typage dynamique, on met ce que l'on veut dans la pile"
5 c push: 'une chaine'.
6 c isFull.
7 c top.
8 c pop.
9 c

```

---

- 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: aStream` à un objet, ajoute sa représentation à la stream.

---

```

1 printOn: aStream
2   aStream nextPutAll: 'une Pile, de taille: '.
3   capacite printOn: aStream.
4   aStream nextPutAll: ' contenant: '.

```

```

5     index printOn: aStream.
6     aStream nextPutAll: ' objets : ('.
7     contenu do: [ :each | each printOn: aStream. aStream space ].
8     aStream nextPut: $).
9     aStream nextPut: $..

```

---

3. Signalez les exceptions, en première version, vous écrirez : `self error: 'pile vide'..`
4. Apprenez à utiliser 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 débogger 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 : <https://jiaxianhua.github.io/smalltalk/2015/04/28/pharo-by-example-7>.
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 cliquer sur le bouton "class-side" du browser.

- 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 flèche 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`: (une de même nom existe déjà quelque part), à 1 paramètre, pour qu'elle appelle la méthode d'instance `initialize`: en lui transmettant l'argument qu'elle a reçu.
- Redéfinir sur `Pile` la méthode **de classe** `new` pour qu'elle appelle la méthode **d'instance** `initialize`: définie en section 2.1 avec comme argument `tailleDefaut`.
- Définir une méthode de classe `exemple` réalisant un programme utilisant une pile;

## 3 Seance TD-TP 4 et 5 - Utilisation de méta-objets en Pharo

### 3.1 S'ouvrir aux fermetures

Les lambda-expressions avec capture de variable (ou fermetures lexicales ou blocks) sont des objets universels de la programmation.

#### 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` :

---

```

1 create
2   | x |
3   x := 0.
4   ^ [ x := x + 1 ]

```

---

3. Appelez deux fois la méthode et stockez les valeurs rendues par ces appels dans 2 variables distinctes.
4. Avez vous remarqué qu'une fermeture est en essence exactement ce qu'est un objet au sens de la programmation par objets (un attribut encapsulé, une fonction pour le manipuler) ?
5. Exécutez plusieurs fois les lambdas (blocks) contenues dans ces deux variables.
6. Les fermetures sont des objets. Inspectez ces deux variables en faisant le lien avec la définition de la classe `BlockClosure`.
7. Les lambdas *Java* sont-elles des fermetures (pouvez vous faire le même exercice en Java) ? Même question avec C++, Python, Javascript.

### 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:` qui permet de récupérer un objet représentant une méthode compilée définie sur une classe.  
Trouvez la classe sur laquelle est définie la méthode `compiledMethodAt:`.
4. En utilisant ces protocoles d'introspection, écrivez sur la classe `Object`, une méthode `exoInspect` qui réalise un **un inspecteur d'objet** de base (non graphique), capable de rendre une chaîne de caractères, contenant les noms et valeurs des attributs du receveur.

Exemple<sup>1</sup> :

---

```

1 Pile new initialize: 4; push: 33; exoInspect. "doit rendre"
2 'contenu: #(33 nil nil nil)
3 index: 1
4 capacite: 4
5 '

```

---

## 4 Seance TD-TP 5 et 6 - Utilisation de méta-objets en Pharo - Suite

(donc en synthèse il y a 3 séances pour les TD-TP 4, 5 et 6 mais vous allez à votre rythme)

### 4.1 Les méta-Objets pour l'accès aux classes, au compilateur et aux méthodes compilées

#### 4.1.1 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 :
  - crée une sous-classe de `Pile` nommée `PileGrossissante`,

---

1. note : le ";" est la cascade de messages, envoyer plusieurs messages successifs au même receveur

- enlève la méthode `grow` de la classe `Pile` et la met sur la classe `PileGrossissante` (voir la méthode `addSelector:withMethod:` de la classe `Behavior`),
- crée 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 nommée `saveGrow` qui contiendra une copie de la méthode `grow` qui doit être définie sur votre classe `Pile` avant de commencer l'exercice.

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

---

```

1 initialize
2   saveGrow
3   ifNil: [ saveGrow :=
4           [ Pile compiledMethodAt: #grow ]
5           on: KeyNotFound
6           do: [ self error: 'Définir grow sur Pile avant de commencer svp' ] ]

```

---

## 4.2 Listes et Arbres - Implantation avec le méta-objet “UndefinedObject” - optionnel, si vous êtes en avance

1. `nil` (null en *Java*) est la valeur par défaut contenue dans tout mot mémoire géré par la machine virtuelle. 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 l'objet `nil` est parfaitement défini mais c'est un point de vue personnel). On peut donc envoyer des messages à `nil` qui appelleront méthodes définies sur `UndefinedObject` (ou sur l'une de ses sur-classes).

En utilisant cette information, programmez une nouvelle version de la classe `LinkedList` (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.** Il est possible de visualiser les méthodes définies sur la classe `UndefinedObject` en consultant le package `HAI931`. Si `HAI931` est bien le nom de votre package de travail usuel pour ce TP, définissez votre 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.

## 4.3 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 particulier la classe `Exception` et sa méthode `signal` (équivalent du `throw` de *Java*), mais bien sûr ici `signal` est une vraie méthode que l'on invoque par un envoi de message avec liaison dynamique alors que le `throw` de *Java* est une forme spéciale du compilateur.

## 5 Les méta-classes implicites en Pharo

Au delà de l'utilisation de base des méta-classes en *Pharo* (cf. paragraphe 2.3), 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. Réalisez `MemoPile`, une sous-classe de la classe `Pile` qui soit conceptuellement une *MemoClass*, donc une classe qui mémorise la liste de ses instances. Ceci revient à définir sur la métaclasse `MemoPile` `class` (automatiquement créée) le comportement adéquat pour que son instance (`MemoPile`) se comporte comme une *MemoClass*. Ainsi il sera possible de demander à `MemoPile` la liste de ses instances (`MemoPile 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 mais sera possible en *CLOS*.

- 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).
- Définissez les classes **Chien** et **Chat** comme sous-classes de **Carnivore**, elle même sous-classe de **Animal**. Faites de **Animal** une classe abstraite.
- Après avoir fait de **Animal** une classe abstraite, vérifiez que **Carnivore** est aussi abstraite. Puis essayez de créez une instance de **Chat**. Résolvez le problème. Comparez avec une solution à méta-classes explicites.
- Relisez le cours et observez la figure montrant la métaclasses Smalltalk (chapitre 5, section "Solution 'a méeta-classes implicites (Smalltalk, ...)"). On y voit que chaque classe est instance d'une métaclasse automatiquement créée.  
Mettez un point d'arrêt (`self halt.`) au début de la méthode `subclass: t instanceVariableNames: f classVariableNames: d poolDictionaries: s package: cat` de la classe **Class**.  
Puis créez une nouvelle classe et observez en pas à pas le processus de création d'une classe. Essayez de trouver où et comment la métaclasse automatiquement associée à cette nouvelle classe est créée.
- Inspectez une méta-classe, quelle différence avec une classe ?

## 6 Séance TD-TP 8 et 9 : Les méta-classes explicites et la méta-programmation en CLOS

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

- Reprenez les exemples de base présentés dans le cours.
- Optionnel : 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.
- 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.
- 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.
- 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.
- 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.