**L3 Informatique - HLIN603**
**Partie III - Expérience du typage dynamique, tout objet et développement Agile avec Smalltalk**
**TP No 1 et 2**

# 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.

# 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 : "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`.

# 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 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 : *World-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''
```

# 4 Classes, instances, méthodes d'instance

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 avec un `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:`, 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'.`.

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.
3. Ecrire une méthode `grow` qui double la capacité d'une pile.

## 5 Composition - méthodes à plusieurs paramètres.

Réaliser une classe distributeur de Bonbons (type foire foraine) à $n$ colonnes utilisant un tableau de $n$ piles.
— 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 élémént de la colonne 1 s'il en reste (donc *aCarambar*, sinon *#YenAPlus*.

## 6 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).

## 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`. De par son nom (`initialize`) cette méthode sera invoquée automatiquement. 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 5.

— Définir une méthode de classe `example` réalisant un exemple de programme utilisant une pile ;

## 8 Héritage - Redéfinitions

1. Typage dynamique ne signifie pas abence de types. Il est possible de programmer des contrôles qui auront lieu à l'exécution. Ecrire une classe `PileTypée`, sous-classe de la classe `Pile` qui permet de d'imposer le type des éléments empilés. En pratique il faut un attribut pour stocker le type des éléments, 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. Identifiez les petits triangles bleus du brower qui permettent d'identifier les méthodes redéfinies et de visualiser facilement les redéfinitions.
3. Naviguez sur la méthode `=` de la classe Objet et visualisez ainsi ses redéfinitions.
4. (Optionel) Evolution de programme. Etendre l'exercice précédent au cas du distributeur en le re-programmant pour en faire une version non contrainte et une version contrainte utilisant des piles typées.

## 9 Utiliser++ les itérateurs

Reprendre l'exercice des comptes bancaires fait en Ocaml. On considère les classes `Account`, `InterestAccount` et `SecureAccount`. écrire la classe `Bank` avec les méthodes suivantes : `add:` ,ajoute un compte bancaire ; `balance`, calcule la somme des soldes des comptes ; `fees`, prélève 5% de frais à tous les comptes.

Je vous donne la méthode `printOn:` :

```
printOn: aStream
    aStream nextPutAll: 'une banque, avec comptes : '.
    accounts printOn: aStream
```

## 10 Listes et Arbres

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 `ArbreBinaireDeRecherche` (ou `ABR`), selon l'énoncé du TP Ocaml, 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).
2. Définissez un itérateur `do:` pour les arbre binaires de recherche.
3. Et si vous définissiez la classe `ABR` comme sous-classe de `Collection` ?
4. **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 UndefinedObject dans le protocole (ou catégorie) **HLIN603-ABR*. Ainsi vous pourrez tout visualisez au même endroit dans le browser.

## 11 S'ouvrir aux 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.
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`.

## 12 Hiérarchie de classes : les expressions arithmétiques

Traitez l'exercice de modélisation des expressions arithmétiques (sur les entiers) du TP OCAML. Ici il sera nécessaire de construire une super-classe (que l'on pourrait appeler Expr) à ces classes pour modéliser les expressions arithmétiques.