L3 Informatique - HLIN603
Partie III - Approche Tout Objet - Développement Agile - Réflexion à l'exécution avec Pharo-Smalltalk.
TDs/TPs

Les notes de cours sont en : http://www.lirmm.fr/~dony/notesCours/smalltalkOverview.s.pdf,

Version imprimable en : http://www.lirmm.fr/~dony/notesCours/smalltalkOverview.pdf.

Le cours, les TDs et TPs utilisent le langage Pharo, une émanation et extension de Smalltalk dont vous trouverez une liste des points marquants ici : https://www.pharo.org/features

1 Téléchargez Pharo Smalltalk

Allez sur le site http://pharo.org/.

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.
Les exercices de ces TPs ont été écrits pour la version "8.0 stable 64 bits" de Pharo, cela ne change rien au code mais par contre l'environnement de programmation a évolué, je vous suggère donc de choisir cette version. Cliquez à gauche sur "8.0 stable 64 bits", puis téléchargez la (clic sur l'étoile orange). Une fois l'image téléchargée, elle apparait dans la fenêtre de droite (voir figure , sélectionnez la et cliquez sur la fleche verte pour télécharger et installer la machine virtuelle correspondante et ouvrir l'environnement *Pharo*. Vous pouvez commencer à travailler.
Pour réouvrir l'environnement (par exemple pour le second TP) relancer le *launcher*, sélectionner l'image contenant votre travail et recliquez sur la fleche verte. Vous pouvez aussi choisir la version 8.0 plus récente dés que vous êtes à l'aise avec l'environnement.

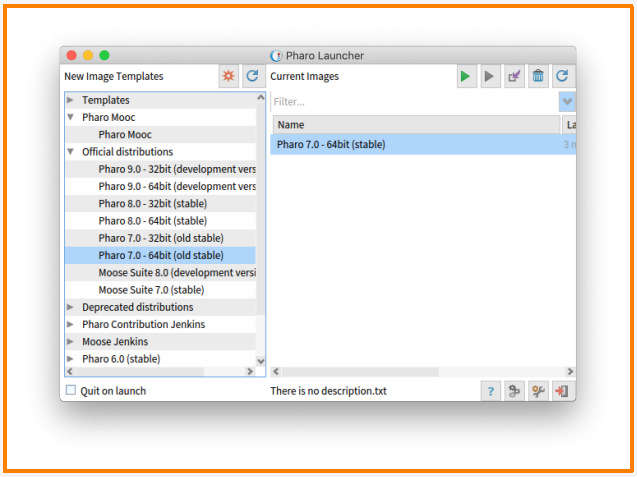

Figure *(1)* – PharoLauncher - image datant de 2018, les versions 8 et 9 n'étaient pas encore sorties

2 La syntaxe et les bases

Voir le cours - notes ci-avant.

Voir les documentations en ligne et les MOOC : https://pharo.org/documentation, il y en a en français.

Faire le tutoriel

L'application Pharo s'ouvre avec une fenêtre ouverte : "Welcome to Pharo xxx". Dans cette fenêtre repérer : "learn Pharo" puis "PharoTutorial go." ou "ProfStef go.". Placez le curseur derrière le point, ouvrez le menu contextuel ("command-Clic" ou "control-clic" ou "clic droit" (selon clavier et 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.

Au cas où vous ne voyez pas cette fenêtre, ouvrez un *playgroud* (menu *Tools¿Playground*), entrez l'expression "ProfStef go." [1] puis *doit* ("Cmd-d").

Le tutoriel 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 aussi un Mooc en ligne : http://mooc.pharo.org.

2.1 Point le syntaxe, les noms et les paramètres des méthodes

Chaque " :" indique qu'un parametre (définition de méthode) ou un argument (appel de méthode) doit suivre.

```
t := Array new: 1.
t at: 1 put: 5 factorial.
t at: 1.
Object compiledMethodAt: #at:put:
```

3 L'Environnement, premières indications

L'environnement [2] , vise à permettre de programmer "in the large" (ce que nous ne faisons pas ici en TD/TP) vite et bien. Voir le menu "tools" pour la globalité des possibilités (notez que Pharo est aussi un environnement de recherche, certains outils sont des prototypes issus de thèses récentes).

- Menu *World* : clic gauche sur fond d'écran. Les items essentiels dans un premier temps sont *Tools-System Browser* (écrire les programmes), *Tools-Playground* (jouer) et *Tools-Transcript* (terminal de réception des traces d'exécution ; essayez par exemple (`Transcript show: 'Hello World'; cr.`).
- Chaque sous-genêtre de chaque outil possède un menu contextuel, "Cmd-clic" ou "Ctrl-clic" ou clic bouton droit souris.
- Sauvegarde de vos travaux : *menu Pharo-save*. Nous ferons plus subtil ultérieurement.
- Ouvrez un `System Browser`. Dans le menu contextuel de sa fenêtre en haut à gauche, choisissez "Find Class" et cherchez la classe *OrderedCollection*. Une fois sélectionnée, regardez la liste de ses méthodes et leur classement en catégories conceptuelles. Les catégories sont un concept de l'environnement mais pas du langage ; elles n'ont pas d'incidence sur l'exécution des programmes.
- Ouvrez un *Playground*, entrez des expressions, puis dans le menu contextuel choisissez "doIt","print It" ou "inspectIt" pour évaluer , évaluer et afficher le résultat ou évaluer et inspecter le résultat.

```
"une collection extensible ordonnée par l'ordre d'insertion des éléments''
c := OrderedCollection new.
"les fonctions anonymes, lambdas du lambda-calcul"
```

1. Notez que c'est un envoi de message qui demande à ProfStef de commencer son cours.
2. Note : forké en Eclipse vers 2000

```
[ :x | x ]. "(lambda (x) x)"
[ :x | x ] value: 33. "((lambda (x) x) 33)"
"ajouter les nombres de 1 à 20 à c''
1 to: 20 do: [:i | c add: i]
"Avec le browser, étudiez la classe Collection, catégorie 'enumerating' pour voir les
itérateurs applicables à une collection. Un iterateur prend
généralement une une lambda en argument."
"compter les nombres pairs dans c"
c count: [:each | each even]
"somme des éléments de c''
c fold: [ :a :b | a + b ].
```

Listing (1) – *Exemple avec une collection*

4 Classes, instances, méthodes d'instance

Pour se familiariser, création d'une classe simple. 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 par composition avec un tableau (`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 (dits "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 lorsque isEmpty sera écrite"
       "self assert: (self isEmpty)."
   ```

   — Ecrivez les méthodes : `isEmpty`, `isFull`, `push:`, `pop`, `top`. Testez les dans le *playground*.

   — Pour rendre les instances de Pile compatibles avec le *printIt* du *Playground*, définissez la méthode `printOn:` suivante sur la classe. C'est l'équivalent du `toString()` de Java. L'opérateur de concaténation des chaînes 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 cette expression, choisissez "debug" dans le menu proposé. Vous voyez une représentation de la pile d'exécution de la machine virtuelle. 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 Intermède - Toute instruction est une expression, tout est objet, toute interaction est un envoi de message

Toute expression a une valeur et nous sommes en style applicatif comme en Scheme (et ocaml ?) donc toute instruction est une expression. De plus tout est objet donc toute exécution de tout code rend en valeur un objet qui peut être inspecté.

Inspectez les objets rendus par les expressions suivantes.

```
"ceci est un commentaire"
5.
5 factorial.
10000 factorial. "Il y a un compilateur JIT dans l'environnement"

5 class.
5 class superclass.
m := 5 class superclass compiledMethodAt: #factorial.
m bytecode.
m isOverridden.
m valueWithReceiver:5 arguments: #().

#fact, #orial.
5 perform: #fact, #orial.

1 = 2.
(1 = 2) ifTrue: [3 factorial] ifFalse: [5 factorial].

Transcript.
100 to: 120 by: 2 do: [:i | Transcript show: i printString; cr].

GTPlayground.
```

```
GTPlayground open.

CircleMorph exampleCircleMorph.

'5 factorial'. "un texte de programme"
OpalCompiler new. "un compilateur"
OpalCompiler new evaluate: '5 factorial'.
(OpalCompiler compiledMethodAt: #evaluate) comment
```

6 Composition de Classes

Programmer une classe `PCalc` réalisant une calculatrice postfixée utilisant une pile pour son implantation (relation de composition UML), que l'on pourra utiliser comme ci-dessous. La dernières ligne est optionnelle ; elle correspond à une version étendue de la calculatrice où un opérateur est toute fonction unaire ou binaire applicable aux éléments présents la pile.

```
c := PCalc new initialize.
c enterOperand: 4.
c enterOperand: 10.
c enterOperator: #*. "rend 40"
c enterOperand: 8.
c enterOperator: #/. "rend 5"
c enterOperator: #factorial. "rend 120"
```

7 Méthodes de classes (méthodes d'instance de méta-classes)

Les méthodes de classe sont définies sur des (méta-)classes (dont les instances sont des classes). Elles s'exécutent en envoyant des messages aux classes (ainsi considérées comme des objets). Exemple d'invocation d'une méthode de classe : `Date today`.

Pour observer ou définir les méthodes de classes, il faut faut utiliser le bouton "class" du browser.

— Avant de passer côté *class*, ajouter une variable de classe `tailleDefaut` à la classe PIle (cela se fait côté "instance" - demandons nous pourquoi ?).

— Définir une **méthode de classe** `initialize` qui fixe à 5 la taille par défaut des piles, valeur à stocker dans la variable de classe `tailleDefaut`.
Vous devez exécuter cette méthode pour que la variable de classe soit initialisée. De par son nom spécifique (`initialize`) cette méthode est reconnue par le *browser* (voir flêche verte en face du nom). Si vous décidez de la nommer `truc`, vous aurez exécuter `Pile truc` quelque part.

— Définir sur `Pile` une méthode de classe `new: n` qui permet de créer une pile de taille `n` ; elle doit appeller la méthode **d'instance** `initialize:` définie en section 4.

— Définir une méthode de classe `example` pour la classe `PCalc` incluant le code du listing 6.

8 Héritage - Redéfinitions

Pour expérimenter l'héritage, reprenons l'exercice des comptes bancaires fait en C++ et Ocaml.

On traite dans ces cection des classes `Account`, `InterestAccount` et `SecureAccount`.

On traitera de la classe *Bank* dont les instances possèdent une collection de comptes en section 9.

1. Définir `Account` puis ses méthodes `initialize`, `deposit:` , `withdraw:`, et `get`.
   Je vous donne la méthode `printOn:`.

```
printOn: aStream
    aStream nextPutAll: 'un ' , self class name , ' de solde : ' .
    balance printOn: aStream.
```

2. **Redéfinition 1**.
   Ouvrez un second browser, visualisez la méthode `=` de la classe `Object`. Repérez le petit triangle bleu à la gauche du nom qui permet d'identifier puis de visualiser toutes ses redéfinitions.
   Redéfinissez la méthode `=` sur `Account` afin que deux comptes soient égaux s'ils ont la même balance.
3. **Redéfinition 2**.
   Redéfinissez `withdraw:` sur `SecureAccount` afin qu'il soit impossible de retirer d'un *secureAccount* si le solde est insuffisant.
4. **Redéfinition 3**.
   Redéfinissez `deposit:` sur `InterestAccount` avec le code ci-dessous, et définissez `depositInterest:` qui ajoute un bonus de 5% à la somme déposée `n`.

```
"méthode d'instance de la classe InterestAccount"
deposit: n
    super deposit: n.
    self depositInterest: n.
```

9 Polymorphisme paramétrique : collection de comptes

Le typage dynamique fait qu'il n'y a aucun problème à ce que le type d'un paramètre (ou d'un attribut) varie d'une utilisation d'une méthode donnée (ou instance donnée) à une autre. Il n'y a pas de problème pour réaliser une collection d'entiers d'un côté et une collection de comptes bancaires d'un autre.
Mais il n'y a aucun contrôle du compilateur pour empêcher une utilisation hétérogène non voulue d'une collection donnée.
On peut éventuellement réaliser de tels contrôles à l'exécution en utilisant le fait que les types sont définis par les classes qui sont des objets. Toute fonction peut avoir un paramètre "type".

```
x := 2.
type := Number.
x isKindOf: type. "-> true"
```

L'usage courant est néanmoins que les éléments qui sont stockés dans une collection doivent être capables de répondre aux messages qui leur sont envoyés lors d'une itération. Chaque itérateur définit en quelque sorte le type ouvert que doivent posséder les éléments présents dans la collection itérée. Par exemple si un itérateur

envoie les message `x` et `y` aux éléments de la collection itérée, ceux-ci doivent comprendre ces messages. Par exemple, tous les objets comprenant le message `clone`, le type ouvert correspondant est `Object`.

Questions.

1. Définissez la classe `Bank`. Une banque possède un ensemble de comptes stockés dans un attribut `accounts` initialisé avec une *OrderedCollection* vide.
2. Définissez les méthodes.
   - `add:`, ajoute un compte au receveur ;
   - `balance`, calcule la somme des soldes des comptes du receveur ;
   - `deposit: n`, depose la somme n sur chacun des comptes du receveur.
3. Une bonne habitude à prendre : écrivez la méthode `printOn:`.

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

4. Définissez et exécutez la méthode de classe `example` ci-dessous.

```
example
    "une méthode de classe de la classe Bank"
    "Bank example" "Selectionnez le commentaire precedent puis printIt"
    | b |
    b := self new initialize.
    b add: (SecureAccount new: 200). "pourquoi les parenthèses ?"
    b add: InterestAccount new.
    b add: (SecureAccount new: 150).
    b deposit: 100.
    ^b
```

5. En l'exécutant, vérifiez que la liaison dynamique fonctionne correctement, donc que la bonne méthode `deposit:` est bien invoquée pour chaque sorte de compte présent dans la collection.
   Vous comparerez les 3 réalisations de cet exercice dans les 3 langages étudiés.
6. Pour voir plus d'itérateurs, ajoutez à la classe `Bank` les méthodes : `fees`, enlève 5% de frais à tous les comptes ; `min`, rendant le compte ayant le plus faible solde.
   Pour cela revoir la collection des itérateurs (protocole *enumetating)*) sur la classe `Collection`.

10 Polymorphisme d'inclusion et Traits

1. Préalable : Définissez sur `Account` les méthodes `=` et `<` permettant de comparer 2 comptes sur la base de la valeur de leur `balance` respectives.
2. Relire éventuellement la section du cours relative aux traits.
   On souhaite pouvoir trier la collection de comptes d'une banque par la balance de ses comptes, du plus bas au plus élevé. On définit la méthode `sort` sur Banque avec comme code : `accounts sort`.
   Sachant que :
   - l'attribut *accounts* est une `OrderedCollection`,
   - `OrderedCollection` hérite de `SequenceableCollection`,

- `SequenceableCollection` utilise le trait `TSortable` dont une partie du code est donné dans le listing ci-dessous,
- il existe un trait `TComparable` définissant la méthode `<=` en fonction de la méthode `<` (Regardez le avec le *Browser) pour voir comment c'est écrit ...*

... déduisez ce qu'il faut faire sur la classe `Account` pour obtenir le résultat demandé, donc pour que `uneBanque sort` fonctionne, avec interdiction d'écrire la méthode `<=` sur `Compte`.

```
1  Trait named: #TSortable
2      uses: {}
3      category: 'Collections-Abstract'
4
5  !TSortable methodsFor: 'sorting' !
6  sort
7      "Sort this collection into ascending order using the '<=' method."
8      self sort: [:a :b | a <= b]
```

11 Tests

Typage dynamique et développement Agile vont de pair avec les tests systématique. La définition des tests peut même être un préalables à l'écriture du code ; les tests faisant partie des spécifications (voir).

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

1. Appliquez le tutoriel ci-dessous au cas de la pile en créant une classe `TestPile`. Il faut pour cela créer, dans le même package que `Pile` 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).

12 Tout est objet : `nil` comme la liste ou l'arbre vide

`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`.

Par ailleurs, cet exercice est l'occasion de bien noter que l'envoi de message réaliste un test de type implicite, à comparer à un test de type explicite réalisé dans un *case-switch*).

1. En utilisant cette information, programmez la classe `ArbreBinaireDeRecherche` (ou `ABR`), selon l'énoncé du TD2 Ocaml - question 4, 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).
   NB : **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 la classe *UndefinedObject* dans le protocole (ou catégorie) de méthodes nommé **HLIN603-ABR*. Ainsi vous pourrez tout visualisez au même endroit (le package HLIN603) dans le browser.

2. Définissez un itérateur `do:` pour les arbre binaires de recherche.

13 Méta-programmation et IDM

La méta-programmation est la programmation des entités de niveau méta[3] opérée avec le langage standard. Elle suppose que les entités de niveau méta soient accessibles aux programmes standards, donc que le langage soit réflexif. On appelle méta-objet un objet représentant une entité du niveau méta.

13.1 Inspections

Inspectez la classe `Pile`, puis son dictionnaire des méthodes, puis sa méthode `push:`.

Inspectez le résultat de l'expression : `Pile compiledMethodAt: #push:`.

Trouvez la classe sur laquelle est définie la méthode `compiledMethodAt:`.

13.2 Un programme qui fabrique un programme

But de l'exercice : créer par programme la classe Cpt représentant les compteurs, puis l'instancier puis envoyer un message à une instance. Les compteurs sont des objets détenant une valeur qu'ils peuvent incrémenter ou décrémenter.

Pour cela, créez une classe *CreateCpt*. Sur cette classe créez la méthode de classe `do` suivante, testez la et améliorez la.

```
do
    "fabriquer une classe et ses méthodes par programme"
    "detruisez la classe Cpt si elle existe avant de relancer l'exécution"
    "CreateCpt do"

    | newClass unCpt initializeMethod |

    "creer la classe Cpt"
    Object
        subclass: #Cpt
        instanceVariableNames: 'val'
        classVariableNames: ''
        package: 'HLIN603'.

    "rérérencer la classe Cpt dans une variable"
    newClass := Smalltalk classNamed: #Cpt.

    "creer une instance"
    unCpt := newClass new.

    "fabriquer le code et compiler la méthode initialize de Cpt"
```

3. par exemple, une classe, le dictionnaire des méthodes d'une classe, l'arbre de syntaxe d'une méthode, la pile d'exécution de la machine virtuelle, le parseur, le compilateur, le browser, ... , toutes les entités qui permettent la fabrication et l'exécution des programmes.

```
22    initializeMethod := OpalCompiler new
23                source: 'initialize\ ^val := 0' withCRs;
24                class: newClass;
25                compile.

27    "ajouter la méthode à la classe"
28    newClass addSelector: #initialize withMethod: initializeMethod.

30    "executer la méthode et verifier la valeur rendue"
31    self assert: (unCpt initialize == 0).

33    "à vous de continuer"
34    "commencer par programmer en début de méthode : détruire la classe Cpt si elle existe"
```

13.3 Accéder à la pile d'Exécution de la machine virtuelle

Implantez sur la classe Symbol, les méthodes `catch` et `returnToCatchwith:` suivantes qui accèdent à la pile d'exécution via la pseudo-variable *thisContext*.

```
!Symbol methodsFor: 'catch-throw'!

catch: aBlock
    "execute aBlock with a throw possibility"
    aBlock value.

returnToCatchWith: aValue
        | catchMethod currentContext |
        currentContext := thisContext.
        catchMethod := Symbol compiledMethodAt: #catch:.
        [currentContext method == catchMethod and: [currentContext receiver == self]]
                whileFalse: [currentContext := currentContext sender].
        currentContext return: aValue.
        ^aValue
```

Listing (2) *– version Pharo-6*

Exemple d'utilisation dans le *playground* :

```
^#Essai catch: [
        Transcript show: 'a';cr.
        Transcript show: 'b';cr.
        Transcript show: 'c';cr.
        #Essai returnToCatchWith: 22.
        Transcript show: 'd';cr.
        33]
```

Listing (3) *– should display a b c (not d) in the Transcript and return 22 (not 33)*

Vous pouvez ensuite lire la méthode `signal` (équivalent de `throw`) de la classe `Exception`.

13.4 Création d'une nouvelle méta-classe

Modifiez la classe `Pile` pour qu'elle mémorise la liste de ses instances dans une collection stockée dans une variable d'instance de la métaclasse, et dotée d'un accesseur sur cette collection.

14 Tester les 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. Exécuter plusieurs fois les blocks contenus dans ces deux variables.

```
c1 := Counter create.
c1 value.
c1 value.
c2 := Counter create.
c2 value.
```

4. Inspectez ces deux variables en faisant le lien avec la définition de la classe `BlockCLosure`.

15 Une petite application complète

Pour mettre ensemble tout ce que vous avez appris sur un exemple un peu plus important, vous pouvez essayer de programmer un jeu de TicTacToe.

Ou bien étudiez la solution qui vous est proposée en http://nerdysermons.blogspot.fr/2012/03/tictactoe-game-in-
html et dont j'ai mis une version à jour dans le répertoire des corrigés. Pour télécharger le corrigé, ouvrez un *FileBrowser*, et faites *FileIn* sur le fichier `TTTGame.st`.

Exercice, modifier le programme pour que l'on puisse choisir le nombre de lignes de la grille (actuellement 3).