La syntaxe du langage Smalltalk

Serge Stinckwich- Serge.Stinckwich@info.unicaen.fr
Stéphane Ducasse - ducasse@iam.unibe.ch

Dans ce troisième article sur Squeak, nous allons découvrir la syntaxe du langage Smalltalk. Nous verrons que la syntaxe est simple et uniforme. Une fois, la syntaxe du langage maîtrisée, il restera à appréhender les classes de base fournies dans l'environnement de Squeak.

Smalltalk : une syntaxe qui tient sur une carte postale !

La syntaxe d'un langage de programmation n'est généralement pas ce qui est le plus intéressant lorsque l'on aborde un nouveau langage. Heureusement, la syntaxe de Smalltalk est simple. On dit généralement qu'elle tient sur une carte postale ! Le bénéfice est immédiat : elle est rapide à apprendre et on se concentre sur l'essentiel, c'est-à-dire écrire du code qui marche !

La syntaxe de Smalltalk comprend principalement :

- 6 mots réservés : **nil**, **true**, **false**, **self**, **super** et **thisContext**,
- les commentaires,
- 3 types d'envois de messages (unaires, binaires et à mots clefs)
- des littéraux pour représenter les objets de base créés à la compilation,
- des clôtures lexicales.

Les mots réservés

Les mots réservés sont également appelés des pseudos-variables parce qu'ils sont en lecture seule. Leur valeur est définie par le système et dépend du contexte d'exécution du programme. L'utilisateur ne peut donc pas les modifier :

- **self et super** font référence au receveur d'un message (on reviendra dessus plus tard),
- **nil** représente une valeur nulle. C'est la valeur par défaut des variables si elles n'ont pas été initialisées. Attention **nil** est un objet, unique instance de la classe **UndefinedObject** (ce qui permet de faire le fameux jeu de mot : nothing is an object),
- **true et false** représentent respectivement la valeur de vérité vrai et faux. **true** est l'unique instance de la classe **True**, **false** l'unique instance de la classe **False**,
- **thisContext** représente la pile d'exécution. Très utile notamment pour le déboggeur de Smalltalk pour accéder au contexte d'exécution.

Les commentaires

Les commentaires sont formés de n'importe quelle séquence de caractères délimitée par des guillemets. Il est possible d'introduire des guillemets dans des commentaires en les entourant de guillemets. Les développeurs Smalltalk font un usage modéré des commentaires, car généralement ils utilisent des noms d'objets ou de méthodes très explicites.

Exemples de commentaires :

```
"ceci est un commentaire"
"ceci est un commentaire
sur plusieurs
lignes"
"ceci est un commentaire avec """un commentaire""" dans un commentaire"
```

On se reportera à l'article précédent pour savoir comment exécuter les expressions Smalltalk données dans

cet article.

Identificateurs

Un identificateur Smalltalk est une suite de caractères commençant par une lettre. Un identificateur permet de nommer un objet ou une méthode Smalltalk. Attention à la casse : **dateDuJour** est différent de **DateDuJour**.

La convention en Smalltalk est d'identifier clairement les noms que l'on donne aux objets et aux méthodes en concaténant les mots les uns aux autres : **ceciEstUnIndentificateur**, **montantFacture**, ... Cela ne devrait pas poser de problèmes à ceux qui sont des adeptes des Wikis (le concepteur du premier Wiki, Ward Cunnigham est d'ailleurs un Smalltakien de renom).

Les identificateurs qui représentent des variables globales, c'est-à-dire connus dans tous le système, commencent forcément par une majuscule. Par exemple : **Transcript**. Les noms de classes sont des variables globales : **OrderedCollection**, **Integer**, ...

On remarquera que les variables ne sont pas typées en Smalltalk. L'opérateur d'affectation (**:=**) permet l'association d'un objet à un identificateur.

x := 2. => affecte à **x** l'objet 2,

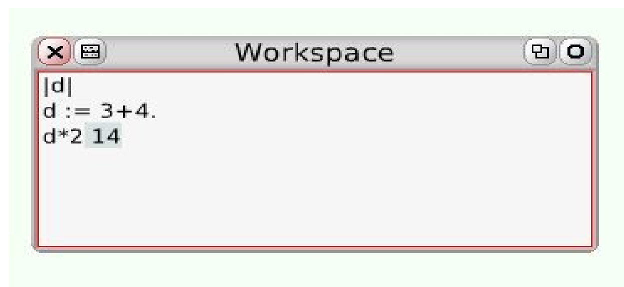
y := 'Hello World'. => affecte à y l'objet 'Hello World',

aujourdhui := Date today. => affecte à **aujourdhui** le résultat de l'expression, c'est-à-dire un objet de la classe **Date**.

Il existe plusieurs catégories de variables suivant l'endroit et la façon dont on les utilise.

Lorsque l'on veut introduire de nouvelles variables locales à un contexte, on les énumère en les entourant de barres verticales : **| |**.

Par exemple, il est possible d'effectuer directement ce code dans un Espace de Travail (**Workspace**) : On déclare la variable d, on affecte le résultat de l'expression 3+4 c'est-à-dire 7, à cette variable. Ensuite on multiplie par deux la valeur contenue par la variable et on obtient 14.



Littéraux

Les littéraux sont les expressions qui représentent des objets couramment employés comme les nombres, les chaînes, le booléens, les tableaux.... Ces objets nous sont pas crées an envoyant le message **new** à une classe comme pour les non-littéraux mais utilise une syntaxe particulière.

Les nombres

12 => nombre entier,
3+4 => somme de deux entiers,
3.141516 => nombre de la classe Float

Précision : les entiers sont en précision aussi importante que l'on souhaite.

Les caractères

Ce sont des instances de la classe Character, ils sont précédés du caractère $:
$a, $Z, $& sont des caractères. Les caractères n'ayant pas de représentation graphique comme espace est obtenu en envoyant un message à la classe correspondante : **`Character space.`**

Chaînes de caractères

Les chaînes de caractères sont formées d'un ensemble de caractères encadrés par des quotes :
`'Ceci est une chaîne de caractères'`. La chaîne vide est bien sur : **`''`**.
Les chaînes de caractères sont des objets de la classe **`String`**, sous-classe de la classe **`Array`**.

Symboles

Un # suivi d'une suite quelconque de caractères forme un symbole (objet de la classe **`Symbol`**, sous-classe de **`String`**). La spécificité d'un symbole est d'être unique dans tout le système.

`#SymboleUnique == #SymboleUnique` retourne donc toujours vrai. == est la méthode qui compare si deux références pointent sur le meme objet contrairement a = qui dit si deux objets sont égaux. Donc ici #SymboleUnique fait référence au même objet en mémoire.

Les tableaux

Un tableau est une instance de la classe **`Array`**, il s'agit d'une collection d'objets de taille fixe qui peuvent être accédés par un indice numérique. Pour créer un tableau, deux possibilités existent :

- **`#()`** définit un tableau créé à la compilation. Les objets sont séparés par des espaces : **`#(1 2 true 1.0 #(1 2 3))`**. On remarquera que les objets dans le tableau sont de nature hétérogène, en effet le typage est dynamique en Smalltalk.

- **`{}`** définit un tableau créé à l'exécution. Les objets sont séparés par des points : **`{ 1. 3+4. true not. Date today}`** crée le tableau : **`#(1 7 false 21 April 2005)`**. Cette forme permet de construire un tableau initialisé dynamiquement à l'exécution avec des expressions Smalltalk, alors que dans la première forme, il n'est possible de mettre dans le tableau que des valeurs littérales.

L'accès aux valeurs d'un tableau s'effectue au moyen des méthodes **`at:`** et **`at:put:`**.**`L'envoi de message at: permet de récupérer la valeur du tableau qui se trouve à l'indice passé en paramètre, at:put: permet de modifier la valeur se trouvant à l'indice indiqué par un nouvel objet. Notez que le premier élément d'un tableau en Smalltalk se trouve à l'indice 1 et non zéro`**

```
t := #(3 1 4 1 5 9).
t at:1.
t at: 1 put: 4.
```

L'évaluation de t retourne : #(4 1 4 1 5 9)

L'envoi de messages : un langage naturel !

L'envoi de message est l'opération de base de Smalltalk. Tous les calculs s'effectuent par envoi de messages à des objets. Une expression d'envoi de message Smalltalk est de la forme : **`<receveur> <message>`**. La résolution d'un message invoque alors la méthode de même nom, méthode qui est définie dans la classe du receveur du message ou une classe ancêtre.

On peut considérer que Smalltalk est une version simplifiée du langage naturel où les objets seraient les noms, les verbes les messages, les compléments les paramètres.

Chaque expression Smalltalk est séparée de la suivante par un point (.), de la même façon qu'une phrase en langage naturel est séparée de la suivante par un point. Le point est un séparateur d'expressions et non pas un terminateur: il n'est pas nécessaire de le mettre à la fin d'une méthode ou d'un bloc.

Voici des exemples d'envois de messages (ne pas tenter de les exécuter car les objets correspondants n'existent pas) :

album play.
album playTrack: 1.
album repeatTracksFrom: 1 to: 10.
album joue. album jouePiste: 1.
album repèteLesPistesDe: 1 a: 10.

Le sens se déduit assez aisément de la syntaxe des expressions Smalltalk.

Il existe 3 formes d'envois de messages :

Messages unaires

Un objet est suivi du verbe (méthode), il n'y a pas d'argument. Par exemple :

`album play.` => la méthode **`play`** est déclenchée sur l'objet album.

`0.7 sin.` => on envoie le message **`sin`** à l'objet 0.7,

`true not.` => on envoie le message **`not`** à l'objet **true**.

`'squeak' reversed.` => retourne la chaîne inversée.

Messages binaires

Un message binaire, principalement utilisé pour les opérations arithmétiques, est formé d'une méthode infixée avec des objets. Le nom d'une méthode binaire est formée par un symbole ou deux symboles dans la liste : **`+ - / \ * ~ < > = @ & ?`**

Voici quelques exemples de messages binaires :

`3 + 4.` => on envoie le message : + 4 à l'objet 3, le résultat est 7. Une banale opération arithmétique est également vu comme un envoi de messages en Smalltalk !

`'Hel', 'lo' .` => on envoie le message : ,'lo' à la chaîne de caractère 'Hel', le résultat est l'objet 'Hello'.

`10@10.` => le message @10 est envoyé à l'entier 10 et à pour conséquence de construire une instance de la classe **`Point`**, qui représente un point du plan cartésien.

Messages à mots-clés

Lorsqu'il y a un ou plusieurs arguments, Smalltalk utilise une syntaxe assez originale, qui peut surprendre dans un premier temps l'utilisateur habitué à celle de C ou Java, mais qui s'avère finalement très simple à lire. Ce sont les messages à mots-clés où les arguments sont insérés entre les éléments du nom de la méthode. Par exemple :

`Color r:10 g:10 b:20 alpha: 0.6.` => envoi d'un message à trois arguments (les trois composantes de couleur rouge, vert, bleu) avec une valeur d'alpha-blending à la classe **`Color`** pour construire l'instance correspondante. L'equivalent en syntaxe Java est **`Color.rgbalpha(10,10,20, 0.6)`**.

`2 raisedTo: 10.` => envoi du message **`raisedTo:`** (c'est-à-dire mettre à la puissance) en utilisant 10 comme paramètre.

L'idée, c'est d'avoir une expression Smalltalk qui peut quasiment se lire comme une phrase de français (ou d'anglais) et communique l'intention du développeur à celui qui lit le code. Un message a mots-clés est un message dont le nom de méthode contient au moins un :. Un : précède donc un argument.

Il est bien sur possible de combiner ces trois formes de base d'envois de messages, si on connaît les règles de priorités d'évaluation suivantes : les envois de messages unaires sont prioritaires sur les binaires qui sont eux-mêmes prioritaires sur les envois à mots-clés. Entre opérateurs de même priorité, on procède par évaluation de gauche à droite. Les parenthèses permettent de modifier naturellement l'évaluation par défaut.

`3 squared reciprocal negated.` => Applique successivement les méthodes unaires : **`squared`**, **`reciprocal`** et **`negated`**.

Speaker woman say: 'Hello, readers of LinuxPratique.'. => On envoie le message unaire **woman** à la classe **Speaker** qui retourne une instance de cette classe initialisée avec une voix féminine, puis on envoie à cette instance un message à mots clés qui déclenche la lecture du texte.

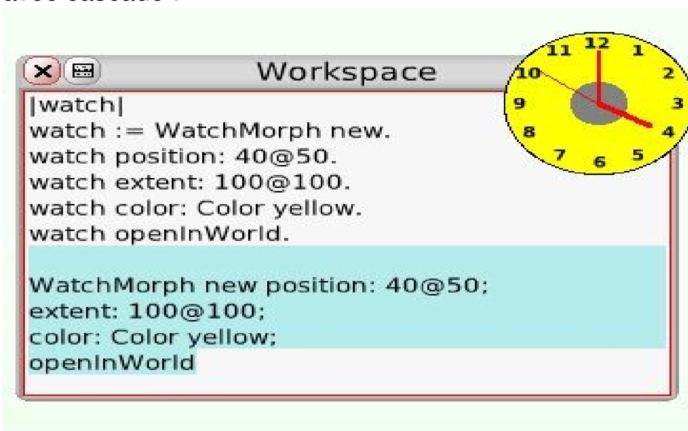
Attention aux erreurs potentielles, l'évaluation de l'expression Smalltalk :

```
2 + 3 * 4 / 2
```

donne 10 et pas 8 comme on pourrait s'y attendre dans un premier temps ... En effet, pour Smalltalk, le + ou * sont des envois de messages comme les autres et on utilise ici la priorité consistant à effectuer les opérations de gauche à droite au lieu d'utiliser les priorités habituelles des mathématiques. Une fois l'habitude prise, le principal avantage est qu'il est alors possible d'introduire ses propres opérateurs sans avoir à gérer de difficiles problèmes de priorités. Ceci permet d'avoir également une grammaire très simple et uniforme pour Smalltalk.

De la même façon que les opérateurs arithmétiques, les opérateurs de comparaisons (=, <, >, ...) ne sont pas des opérateurs prédéfinis en Smalltalk, ce sont en fait des envois de messages comme les autres.

Smalltalk offre une facilité supplémentaire pour exécuter une suite d'envois de messages sur le même objet : la cascade de messages. Par exemple, comparer sur l'exemple suivant qui ouvre à l'écran une montre à fond jaune, la version sans et avec cascade :



Le point-virgule (;) permet de cascader plusieurs envois de messages au même objet, ici une instance de la classe **WatchMorph**.

Blocs et structures de contrôles

Il n'est pas rare d'entendre des gens ne connaissant pas Smalltalk dire que les structures de contrôles n'existent pas en Smalltalk ! C'est bien sur ridicule. En fait, toutes les structures de controle existent, par contre elles sont définies sous forme d'envois de messages à des objets (booléens, fermetures, entiers ou intervalles). De plus il est tout à fait possible de définir ses propres structures de contrôles si cela est nécessaire.

Smalltalk introduit la notion de blocs (ou fermetures lexicales). Il s'agit d'objets qui contiennent du code qui peut être évalué ultérieurement et ainsi transmis comme n'importe quel autre objet. Ces objets disposent d'arguments permettant de paramétrer le code lors de l'évaluation. Un bloc peut etre vu comme une méthode anonyme pouvant etre décrit dans n'importe quelle expression et passé en argument ou retourné comme résultat.

Exemple :

```
x := [3+4].
x value.
```

L'envoi de message **value** évalue le bloc et retourne le résultat de la dernière expression du bloc. Les blocs peuvent être affectés à des identificateurs, passés en paramètre comme on le verra pour les expressions conditionnelles.

Il est possible d'avoir des blocs paramétrés également. Ce sont alors de véritables fonctions :

```
1: |f |
```

```
2: f := [:v | v+1].
3: g := [:x :y | x+y].
4: f value: 1.
5: g value:1 value:2.
```

La ligne 4 retourne comme valeur 2, la ligne 5 retourne 3. Le code suivant génère trois sons différents suivant le paramètre du bloc :

```
block := [ :soundName | (SampledSound soundNamed: soundName) play].
block value: #croak.
block value: #scratch.
block value: #chirp
```

Il est également possible de définir des variables temporaires à l'intérieur d'un bloc en les entourant à l'aide des barres verticales ||.

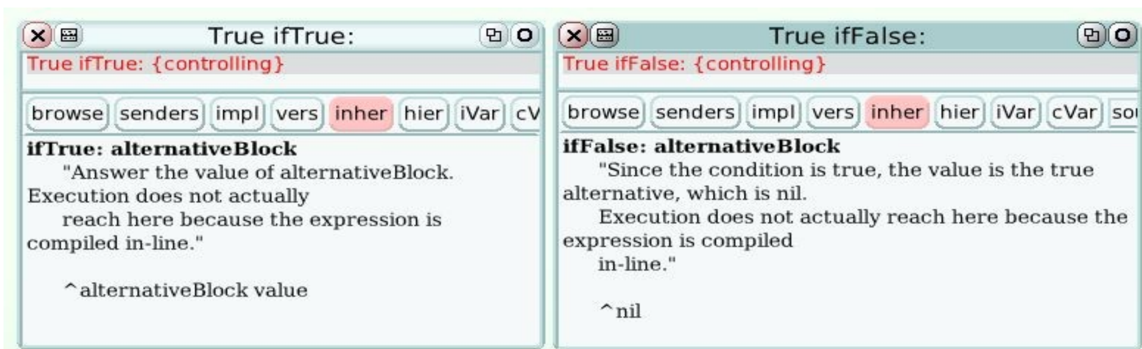
Conditionnelles

Les expressions conditionnelles utilisent les blocs et sont de la forme suivante :

```
1 < 2 ifTrue: [Transcript show: 'vrai'].
1 > 2 IfFalse:[Transcript show: 'faux'].
1 > 2 ifTrue:[Transcript show:'1'] ifFalse:[Transcript show:'2'].
```

On obtient successivement dans la fenêtre de `Transcript` : vrai, faux, et enfin 1.

Avec ce nous avons déjà vu, vous pouvez en déduire que **`ifTrue:`**, **`ifFalse:`**, **`ifTrue:IfFalse:`** sont des envois de messages à mots-clés qui sont envoyés à des instances de la classe **`Boolean`** (ici le résultat de la comparaison de 1 et de 2). Il est également possible de comprendre comment les expressions conditionnelles sont implémentées en Smalltalk en examinant le code des méthodes : **`ifTrue:`**, **`ifFalse:`**, **`ifTrue:ifFalse:`** au niveau des classes **`True`** et **`False`**. Pour cela, ouvrons deux mini-navigateurs de classe sur la classe **`True`** par exemple :



C'est relativement simple. La méthode ifTrue: définie sur la classe True est exécutée quand le receveur du message ifTrue: est true. Dans un tel cas, elle se contente d'évaluer le bloc passé en paramètre. De la même manière ifFalse: définie sur cette même classe ne fait rien car le receveur n'est pas faux mais vrai donc rien ne doit se passer. De la même manière, la méthode ifFalse: définie sur la classe False exécutera son argument et la méthode ifTrue: ne fera rien. À vrai dire, peu de langages permettent de faire cela ! Maintenant pour rassurer les gens pensant que cela est coûteux en terme d'exécution, il faut savoir que le compilateur optimise drastiquement ces envois de messages dans le code généré.

Itérations

De la même façon, les itérations sont des messages envoyés à divers objets (nombres, fermetures) suivants les mêmes règles syntaxiques que celles des messages.

Une simple répétition s'écrit en envoyant le message `timesRepeat:` à un nombre et en passant le code à répéter.

`4 timesRepeat: [ Transcript show: 'hello' ;cr ]` écrit 4 fois 'hello' dans le transcript.

L'itération de type tant-que peut s'utiliser facilement en utilisant deux blocs sans paramètre :

```
| f n |
f := 1.
n := 4.
[ n>1 ]
   whileTrue: [ f := f*n. n := n-1 ].
f.
```

Tant que la condition dans le premier bloc est vérifiée, le deuxième bloc est effectué.

Ici, on calcule la factorielle de 4 : f = 1*2*3*4 = 24.

Voici un autre exemple d'itération, où l'on effectue n fois le bloc paramétré en faisant varier à chaque fois, la valeur de i de 1 à 100 :

```
|somme|
1 to: 100 do: [:i | somme := somme + i*i]
```

On calcule ici la somme des 100 premiers entiers.

De très nombreuses autres formes d'itération existent en Smalltalk, la plupart utilise des parcours de la classe **`Collection`**. Voici quelques exemples sur une collection de nombres.

Le message do:

```
#(19 69 15 10 44 78) do: [:each | Transcript show: each printString ;cr].
```

Affiche tous les éléments de la collection dans le Transcript. En fait, do: applique à chaque élément le bloc passé en argument. Notez que l'on aurait pu utiliser :element ou tout autre nom à la place de :each qui est juste le nom du seul paramètre du bloc comme le montre le bloc suivant [:element | Transcript show: element printString ; cr]

Le message collect:.

```
#(19 69 15 10 44 78) collect: [:each | each odd ].
=>  #(true true true false false false)
```

Envoie le message odd (qui dit si un objet est impair) à chaque élément de la collection et retourne une collection contenant tous les résultats.

```
#(19 69 15 10 44 78) select: [:each | each odd ].
=>  #(19 69 15)
```

Sélectionne tous les éléments de la collection qui sont impairs.

Le message reject:

```
#(19 69 15 10 44 78) reject: [:each | each odd ].
=> #(10 44 78)
```

Retourne tous les éléments de la collection n'étant pas impair.

Comme vous le voyez, parcourir des collections est extrémement simple et élégant en Smalltalk. Sachez d'autant plus que ces méthodes sont définies polymorphiquement et donc s'appliquent aussi bien aux

tableaux, qu'aux listes, ensembles, sac, collections ordonnées et autres structures de données.

Dernier point de syntaxe, dans une méthode, la flèche vers le haut (^) rend le résultat de l'expression qui la suit à l'objet ayant invoqué la méthode. C'est l'équivalent du `return` en Java par exemple.

En conclusion

Dans cet article, nous avons abordé la syntaxe de Smalltalk. Il vous est maintenant possible de comprendre tout le code de squeak. Il ne sera plus nécessaire d'y revenir, puisque tout les aspects syntaxiques ont été abordés dans ces quelques pages. Cette syntaxe légère et uniforme est un des aspects les plus puissants de Smalltalk, elle permet au développeur de se concentrer sur les aspects essentiels de conception de son application. Pour un débutant en programmation objet, c'est du temps de gagné, il ne passe pas plusieurs semaines voire mois à comprendre les subtilités de la syntaxe comme avec d'autres langages.

Dans notre prochain article, nous présenterons le modèle objet de Squeak qui est simple, uniforme et élégant avant d'aborder les jouets électroniques (Etoys) et l'époustouflant framework d'application web dynamique Seaside (http://www.seaside.st/).

D'ici la, vous pouvez parcourir l'ensemble des classes du système de Squeak en jetant un coup d'œil sur l'immense librairie de classes qui vous est offerte. N'essayez pas de tout comprendre, mais il vous est déjà possible d'appréhender certaines choses et éventuellement de les modifier.

Liens

- le site officiel qui est actuellement en plein changement **http://www.squeak.org/**.
- le Wiki de la communauté: **http://minnow.cc.gatech.edu/**
- Le Wiki de la communauté francaise: http://www.iutc3.unicaen.fr/smalltalkfr/pmwiki.php
- Le groupe des utilisateurs européens de Smalltalk (European Smalltalk User Group). L'adhésion est gratuite : **http://www.esug.org/**
- Des livres gratuits en lignes sur Smalltalk et Squeak: **http://www.iam.unibe.ch/~ducasse/FreeBooks.html**
- Un livre sur Squeak en francais: **http://www.iam.unibe.ch/~ducasse/Books.html** : Squeak, X. Briffault et S. Ducasse, Eyrolles, 2002.