Modélisation des systèmes à événements discrets

Partie 6 : Rudiments de Smalltalk

Denis Giacona
ENSISA
Ecole nationale supérieure d'ingénieurs sud Alsace
12, rue des frères Lumière
68 093 MULHOUSE CEDEX
FRANCE

Tél. +33 (0)3 89 33 69 00

MCours.com

1. Introduction 4
1.1. Les environnements de programmation Smalltalk 4
1.2. Les concepts de Smalltalk 5
1.3. Ce qu'il faut savoir de Smalltalk dans PACE 6
1.4. Les éléments syntaxiques de Smalltalk 7
1.5. Les objets de Smalltalk 8
1.5.1. La structure d'un objet 9
1.5.2. Le comportement d'un objet 10
1.6. Les classes 11
1.7. Les métaclasses 12
1.8. L'héritage 13
1.8.1. La stratégie de recherche des méthodes 14
1.9. La surcharge (le masquage) et le polymorphisme 15
1.10. Les mots réservés de Smalltalk 16
1.11. Les variables et l'opérateur d'assignation dans Smalltalk 17
1.12. Les littéraux (expressions constantes) 18
1.13. Les types de messages de Smalltalk 19
1.14. La combinaison de messages 20
1.15. Evaluation des combinaisons de messages 20
1.16. Les outils de développement dans PACE 21
2. Les séquences d'actions 22
2.1. Les types d'expressions 22
2.2. Le séparateur d'expressions « point » 23
2.3. Le séparateur d'expressions « point virgule » 23
2.4. Expressions arithmétiques et relationnelles 24
3. Méthodes de comparaison d'objets (égalité, identité) 25
4. Affectations de variables (l'élément syntaxique :=) 26
4.1. Deux variables désignent le même objet primitif (simple) 26
4.2. Deux variables désignent deux objets structurés distincts 27
4.3. Désignation d'un objet structuré à plusieurs niveaux 28
5. Méthodes de copie d'objets 29

UHA Université de Haute-Alsace Mulhouse-Colmar — ensisa école nationale supérieure d'ingénieurs sud alsace

5.1. Copie superficielle (de premier niveau) : `shallowCopy` et `copy` ... 29
5.2. Copie profonde (à tous les niveaux) : `deepCopy` ... 32
6. Les blocs ... 33
7. Schémas conditionnels ... 35
8. Les itérations ... 38
8.1. Répétitions inconditionnelles ... 38
8.2. L'itérateur `do:` appliqué à une chaîne ou une collection ... 39
8.3. L'itérateur `select:` appliqué à une collection ... 39
8.4. L'itérateur `reject:` appliqué à une collection ... 40
8.5. L'itérateur `collect:` appliqué à une collection ... 40
8.6. L'itérateur `detect:` appliqué à une collection ... 41
9. Les collections de données ... 42
9.1. Classes et sous-classes ... 42
9.2. Création d'instances de collections ... 43
9.3. Accès à une instance de collection ... 44
9.4. Array ... 45
9.4.1. Création de tableaux ... 45
9.4.2. Accès à un tableau ... 46
9.5. OrderedCollection ... 47
9.6. Association ... 49
9.7. Dictionary ... 50
9.7.1. Itérations à partir d'un dictionnaire ... 51
9.7.2. Dictionnaire de fonctions ... 52

1. Introduction

1.1. Les environnements de programmation Smalltalk

- **VisualWorks** (Cincom), **Squeak**, …
- *Open source*
- Orientés objets (« tout est objet »)
- Gratuits pour la plupart
- Capacité à observer ou à modifier leur propre fonctionnement (réflexivité)

PACE est une application développée avec VisualWorks.

- Elle contient des classes et des méthodes dédiées aux réseaux de Petri à attributs
- Un certain nombre de fonctionnalités standard ont été ôtées de l'environnement originel (pas d'accès au explorateurs de classes, donc pas de possibilité de création de nouvelles classes ni de méthodes).

1.2. Les concepts de Smalltalk

- Smalltalk est un système où tout est intégré (environnement de développement, bibliothèque de classes, langage) → pas d'éditeur de texte ni de compilateur externes
- Tout est **objet** (dates, nombres, listes, fenêtres, classes, blocs de code, …) ; tout objet appartient à une **classe**
- Les objets sont typés, mais les variables n'ont pas de type
- Le langage est proche du langage naturel (anglais) ; facile à apprendre car il y a très peu d'éléments syntaxiques
- Une action correspond à l'envoi d'un **message** à un objet
- Il est possible d'inspecter et de manipuler des objets en temps réel
- Le langage n'intègre pas de mots clés pour les structures de contrôle (if-then-else, for, while, …) ; ces structures sont implémentées par l'envoi de messages
- Une application Smalltalk est constituée de deux fichiers principaux :
    - un fichier image .im (bibliothèque standard de classes, environnement de développement, langage), indépendant des plateformes hôtes
    - un exécutable .exe (la machine virtuelle qui exécute le code de l'image), spécifique de la plateforme hôte (Windows, Mac, Linux, …)
- L'état exact de l'environnement de développement, des classes, des fenêtres, du programme peut être sauvegardé (sauvegarde du fichier image sous un nom choisi par l'utilisateur)
- Un ramasse-miettes (récupérateur de mémoire) est intégré et transparent pour le développeur

1.3. Ce qu'il faut savoir de Smalltalk dans PACE

Pour tirer profit de toute la puissance de l'outil PACE, il faut connaître les rudiments du langage Smalltalk, à savoir :

- La définition d'un objet (structure et comportement)
- La façon de séquencer des actions
- La syntaxe d'un message
- Quelques classes et sous-classes usuelles
  - Collection, Boolean, Number, …
- Quelques méthodes usuelles

1.4. Les éléments syntaxiques de Smalltalk

$	annonce d'un littéral caractère
' '	délimitation d'une chaîne de caractères
#	annonce d'un littéral symbole
#( )	création littérale d'un tableau initialisé
| |	zone de déclaration de variables dans un contexte (bloc, méthode, workspace)
:=	affectation d'une variable
=	comparaison de deux valeurs (égalité d'objets)
==	comparaison de deux références (égalité de pointeurs d'objet, identité d'objets)
.	séparation d'expressions (affectations ou envois de messages)
;	mise en cascade de messages à un même objet
[ ]	création d'un bloc
:	annonce d'une variable locale dans un bloc (argument)
|	annonce de la fin des déclarations des arguments dans un bloc
" "	délimitation d'un commentaire
^	renvoi d'un résultat dans une méthode

1.5. Les objets de Smalltalk

- Un objet sert à stocker des informations propres et à y accéder, à envoyer ou répondre à des messages.

- Un objet comporte une structure de données et un comportement

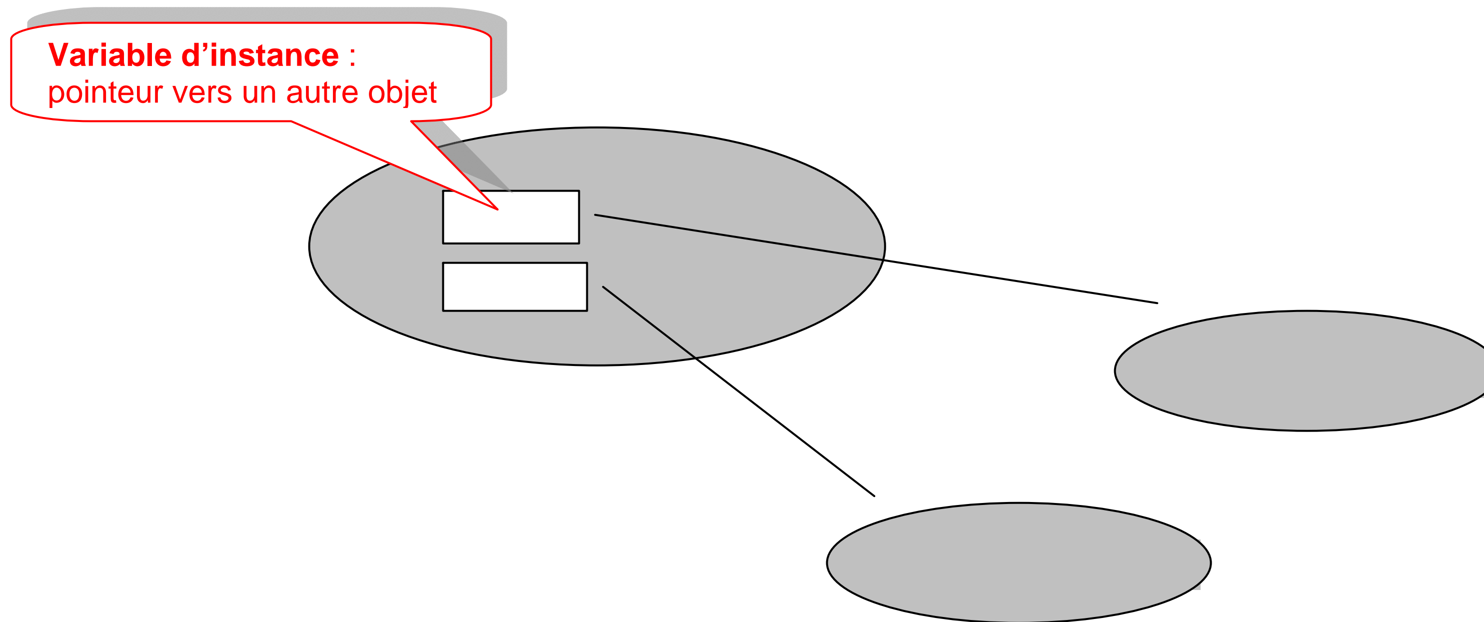
    - les données représentent l'ensemble des valeurs que détient l'objet à un moment donné ; elles sont appelées variables d'instances
        - ce sont des références (pointeurs) vers d'autres objets

    - le comportement est l'ensemble des opérations que l'on peut réaliser sur les données ; les codes de ces opérations sont répartis dans des **méthodes** d'instances
        - les méthodes sont déclenchées par des messages ; le nom d'une méthode est appelé **sélecteur** ; envoyer un message à un objet est le seul moyen d'interagir avec lui
        - les variables d'instances d'un objet sont privées et ne sont accessibles que par les méthodes propres à l'objet (c'est le principe d'encapsulation)

- Un objet n'est pas créé par un procédé individuel mais par l'intermédiaire d'une classe ; tout objet est « instancié par une classe ».

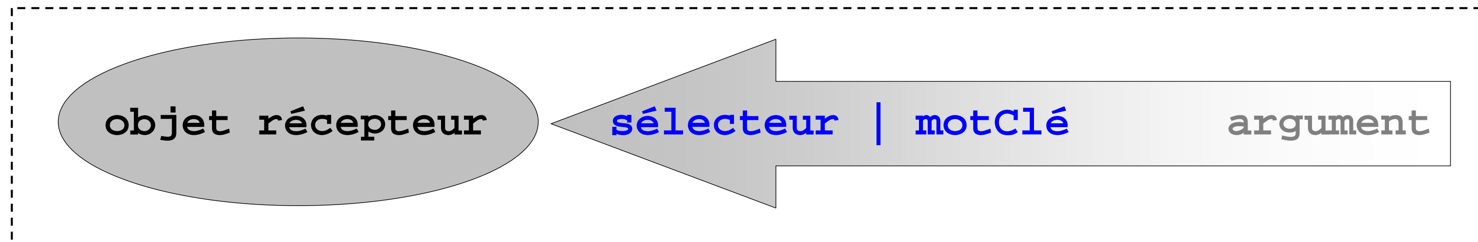
1.5.1. La structure d'un objet

Tout objet contient un ensemble de pointeurs vers d'autres objets



1.5.2. Le comportement d'un objet

❑ **Un message est la demande faite à un objet de déclencher l'un de ses comportements**



❑ **Lorsqu'un objet reçoit un message (d'un autre objet), il peut réagir de différentes manières**

- donner naissance à un nouvel objet
- modifier son état interne
- envoyer des messages à d'autres objets

```
5 squared.
```

Effet : un nouvel objet est créé → 25

```
tableau at: 1 put: 'premier élément'.
```

Effet : le tableau est modifié

```
'Texte' do: [:car | Transcript show: car printString].
```

Effet : demander un affichage de lettres dans une fenêtre

1.6. Les classes

- Une classe a un double rôle :
  - C'est un **modèle** (« moule ») d'instances contenant la définition d'objets ayant une même structure de données et un même comportement
    - Ses données sont réparties en variables de classes (données partagées par toutes les instances) et déclarations des variables d'instance des objets instanciés
    - Son comportement est réparti en méthodes de classe (spécifiques à la classe, non utilisables par les objets instanciés) et méthodes d'instance
      - Les fonctions des méthodes de classe sont par exemple : la création d'instance, l'initialisation de variables de classe (méthode invoquée automatiquement lorsque la classe est chargée en mémoire),…
      - Les variables de classe sont accessibles par les méthodes d'instance ou de classe

  - C'est un **générateur** d'instances (instanciation) ; mais pas toujours
    - Une classe qui génère des instances est dite classe concrète ; une classe qui n'en génère pas est une classe abstraite (classe qui sert uniquement de dépositaire de code commun pour ses sous-classes)
    - Un « objet instance » bénéficie d'une part de ses propres variables d'instance (pour différencier les valeurs entre instances) et d'autre part des variables de classe
    - Les méthodes d'instance utilisées par un « objet instance » sont en fait stockées dans sa classe (car elles sont communes à toutes les instances)

1.7. Les métaclasses

- En Smalltalk une classe est un objet à part entière, et puisque tout objet est issu d'une classe, elle doit donc être elle-même instanciée par une classe ; cette classe s'appelle **métaclasse**

- Lorsqu'une classe est créée, la métaclasse est d'abord créée, qui créera à son tour la classe en question, qui sera son unique instance

- On peut donc distinguer
  - des instances ordinaires pour des objets usuels
  - et des instances classe pour les objets classes

MCours.com

1.8. L'héritage

- L'héritage est un procédé qui permet de construire des classes qui ont un tronc commun mais aussi des parties spécifiques

- Une classe peut avoir une ou plusieurs sous-classes ; elle est alors désignée comme la superclasse des sous-classes ; la sous-classe contient les spécificités de sa superclasse

- Toute classe a une superclasse, sauf la classe **`Object`**, sommet de l'arbre (hiérarchie) des classes

- Une sous-classe **hérite** (données et comportement) de sa superclasse
    - Une instance comprend toutes les méthodes de sa classe et toutes celles de sa superclasse, etc…
    - Une instance contient toutes les variables d'instance de sa classe et toutes celles de sa superclasse, etc…

1.8.1. La stratégie de recherche des méthodes

Lors de la réception d'un message, le receveur transmet le message à sa classe

- si aucune méthode correspondante au message n'est trouvée dans la classe du receveur, elle est recherchée dans sa super classe
- ce procédé se répète jusqu'à ce qu'une méthode correspondant au message soit trouvée dans l'une des super classes
- si la méthode n'est pas trouvée, le système génère une erreur

1.9. La surcharge (le masquage) et le polymorphisme

- Si une classe définit une nouvelle méthode dont le nom est identique à celui d'une méthode qui existe déjà dans une superclasse, la nouvelle méthode remplace celle qui est héritée ; c'est le principe de **surcharge** ; on dit aussi que la méthode de plus bas niveau masque celle de plus haut niveau

- Deux classes différentes peuvent définir deux méthodes différentes qui portent le même nom ; c'est le **polymorphisme**
    - Des objets différents peuvent comprendre le même message (même sélecteur), mais faire des choses différentes en réponse au message

1.10. Les mots réservés de Smalltalk

`true, false` Booléens, instances uniques des classes True et False
(sous-classes de Boolean)

`nil` Objet « nul » ou « indéfini »

`self` Pseudo-variable qui fait référence au receveur d'une méthode (« lui-même ») ;
si une méthode d'un objet veut appeler une autre méthode du même objet,
elle doit envoyer un message à l'objet lui-même
(ce n'est pas une vraie variable car l'utilisateur ne peut pas lui assigner d'objet
par l'opérateur :=)

`super` Pseudo-variable qui fait référence au receveur d'une méthode ;
si une méthode d'un objet envoie une autre méthode à super,
cela veut dire que la définition de cette dernière n'est pas celle de l'objet mais
doit être cherchée dans la super classe

1.11. Les variables et l'opérateur d'assignation dans Smalltalk

- Quand un objet est créé (instancié), le système lui attribue une représentation unique
- Une **variable** Smalltalk permet d'établir un lien avec la représentation d'un objet ; elle peut être assimilée à un **pointeur**, c'est-à-dire l'adresse d'un emplacement mémoire
- Une variable qui n'a pas encore eu d'affectation pointe l'objet **`nil`**
- Plusieurs variables peuvent pointer le même objet

Opérateur d'assignation de variable

```
sousChaine := 'chaîne' copyFrom: 3 to: 4.
```

La variable `sousChaine` pointe un objet évalué par le message, c.-à-d. l'objet **`'ai'`**

1.12. Les littéraux (expressions constantes)

- **Les nombres**

```
14          "nombre décimal"
45.6        "nombre réel"
8r356       "nombre octal"
16rFEAB     "nombre hexadécimal"
```

- **Les caractères**

```
$a
```

- **Les chaînes de caractères**

```
'Ceci est une chaîne'
```

- **Les symboles**

```
#client
```

- **Les tableaux initialisés**

```
#(1 2 3 4)
#(1 #(10 11 12) 2 3)
```

1.13. Les types de messages de Smalltalk

- **Message unaire** `unObjet sélecteur`

```
12 class                → SmallInteger
'abc' asUppercase       → 'ABC'
```

- **Message binaire** `unObjet sélecteur unArgument`

(le sélecteur est une séquence de un ou plusieurs caractères parmi : + - * / , & = > | < _ @)

```
10 + 2                  → 12
'jean' , '-jacques'     "concaténation de 2 chaînes de caractères"
                        → 'jean-jacques'
```

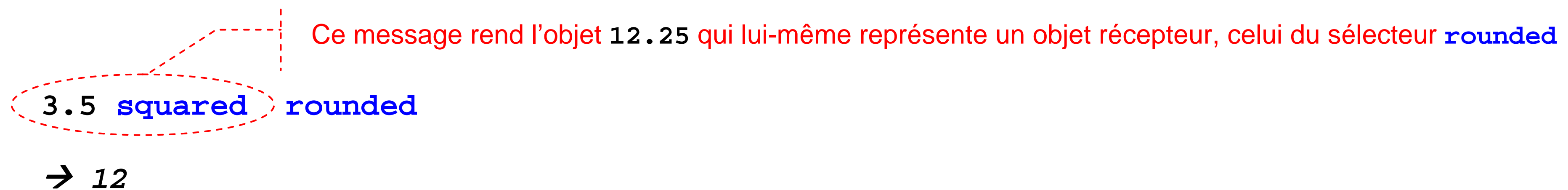
- **Message à mots clés** `unObjet motClé: unArgument`

`unObjet motClé1: unArgument motClé2: unArgument` **etc …**

```
2 raisedTo: 8           → 256
5 between: 1 and: 6     → true
```

1.14. La combinaison de messages

Chaque message retourne un objet ; donc un message peut prendre la place d'un objet dans un autre message, ce qui revient à faire une combinaison de messages.



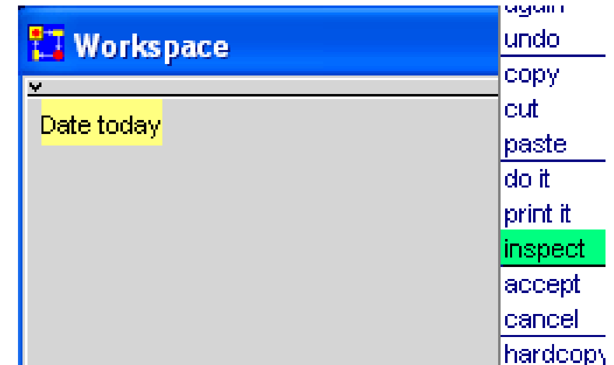
1.15. Evaluation des combinaisons de messages

- **Priorité par type de message :**
    - **d'abord unaire**
    - **ensuite binaire**
    - **enfin à mots clés**
- **Si tous les messages sont du même type, ils sont exécutés de la gauche vers la droite**
- **Un message entre parenthèses est évalué d'abord → les parenthèses permettent d'imposer un ordre d'évaluation**

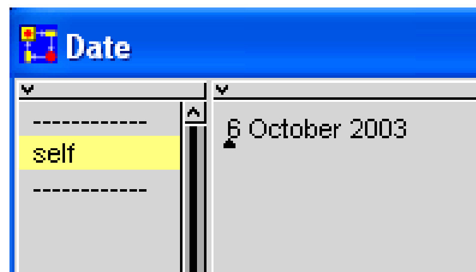
```
3 + 2 raisedTo: 2 squared     "est équivalent à" (3+2) raisedTo: (2 squared)

5 + 3 * 4    "est équivalent à"   (5+3)*4
```

1.16. Les outils de développement dans PACE

- Un espace de travail (*workspace*) pour éditer et évaluer des expressions quelconques



- Des inspecteurs pour afficher ou modifier les valeurs des objets



Remarque : contrairement à un outil de développement classique (du type Visual Works), PACE ne fournit pas de navigateur pour lire ou définir des classes et des méthodes.

2. Les séquences d'actions

Un programme est une séquence d'expressions séparées par des points, des points virgules ou des commentaires.

2.1. Les types d'expressions

- **Affectation de variable**
```
x := 1        "x est un pointeur qui contiendra l'identificateur  de l'objet 1"
```

- **Nom de variable**
```
x
```

- **Littéral**
```
1                     "nombre"
#premier              "symbole"
$A                    "caractère"
'Chaîne de caractères'
#(1 'deux' #trois)    "tableau initialisé"
```

- **Envoi de message**
```
listeNotes  add:16
```

- **Bloc**
```
[x := x + 1]
```

2.2. Le séparateur d'expressions « point »

Le point « . » est le séparateur d'expressions, qu'il s'agisse d'affectations de variables ou de messages.

```
x := 10.
y := 0.
5 timesRepeat: [y := x * 2 + y].
y
→ 100
```

2.3. Le séparateur d'expressions « point virgule »

Une séquence de messages s'appliquant au même récepteur peut être simplifiée par une mise en cascade. Les messages sont alors séparés par des points virgules « ; ».

```
col := OrderedCollection new.
col add: 'a' ; add: 'b' ; add: 'c'.
col.
→ OrderedCollection ('a' 'b' 'c')
```

2.4. Expressions arithmétiques et relationnelles

- **Messages binaires**

Expression	Résultat	
`3 * 4`	→ `12`	
`10 / 3`	→ `(10/3)`	
`10.0 / 3`	→ `3.33333`	
`10 // 3`	→ `3`	**Partie entière de la division**
`10.0 // 3`	→ `3`	
`10 \\ 3`	→ `1`	**Modulo**
`10.0 \\ 3`	→ `1.0`	

- **Expressions booléennes**

`i > j and: [uneExpression. k > l]`
→ **si i > j : le bloc argument de `and:` est évalué et le résultat dépend de k>l**
→ **si i <= j : le résultat est `false` et le bloc argument de `and:` n'est pas évalué**

`i > j or: [uneExpression. k > l]`
→ **si i > j : le résultat est `true` et le bloc argument de `or:` n'est pas évalué**
→ **si i <= j : le bloc argument de `or:` est évalué et le résultat dépend de k>l**

Expression	Résultat	
`10 > 3 & (4 > 6)`	→ `false`	
`10 > 3 | (4 > 6)`	→ `true`	**Evaluation de tous les termes**

3. Méthodes de comparaison d'objets (égalité, identité)

- **Pour les objets primitifs, c.-à-d. uniques (nombres entiers, symboles, caractères), il n'est pas nécessaire de distinguer l'égalité de l'identité**

```
10 = 10        → true     test d'égalité
10 == 10       → true     test d'identité

#a = #a        → true     test d'égalité
#a == #a       → true     test d'identité
```

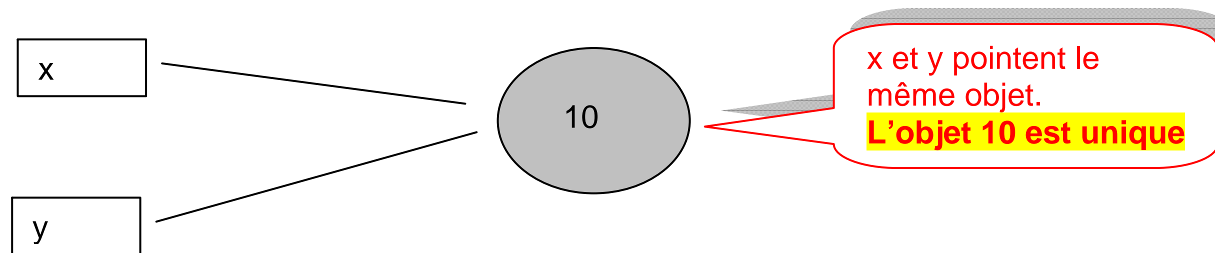
- **Pour les objets non primitifs, nombres réels et objets structurés (Array, Collections, …), il faut distinguer l'égalité de l'identité**

```
#(10) = #(10)     → true     test d'égalité
#(10) == #(10)    → false    test d'identité

'abc' = 'abc'     → true     test d'égalité
'abc' == 'abc'    → false    test d'identité
```

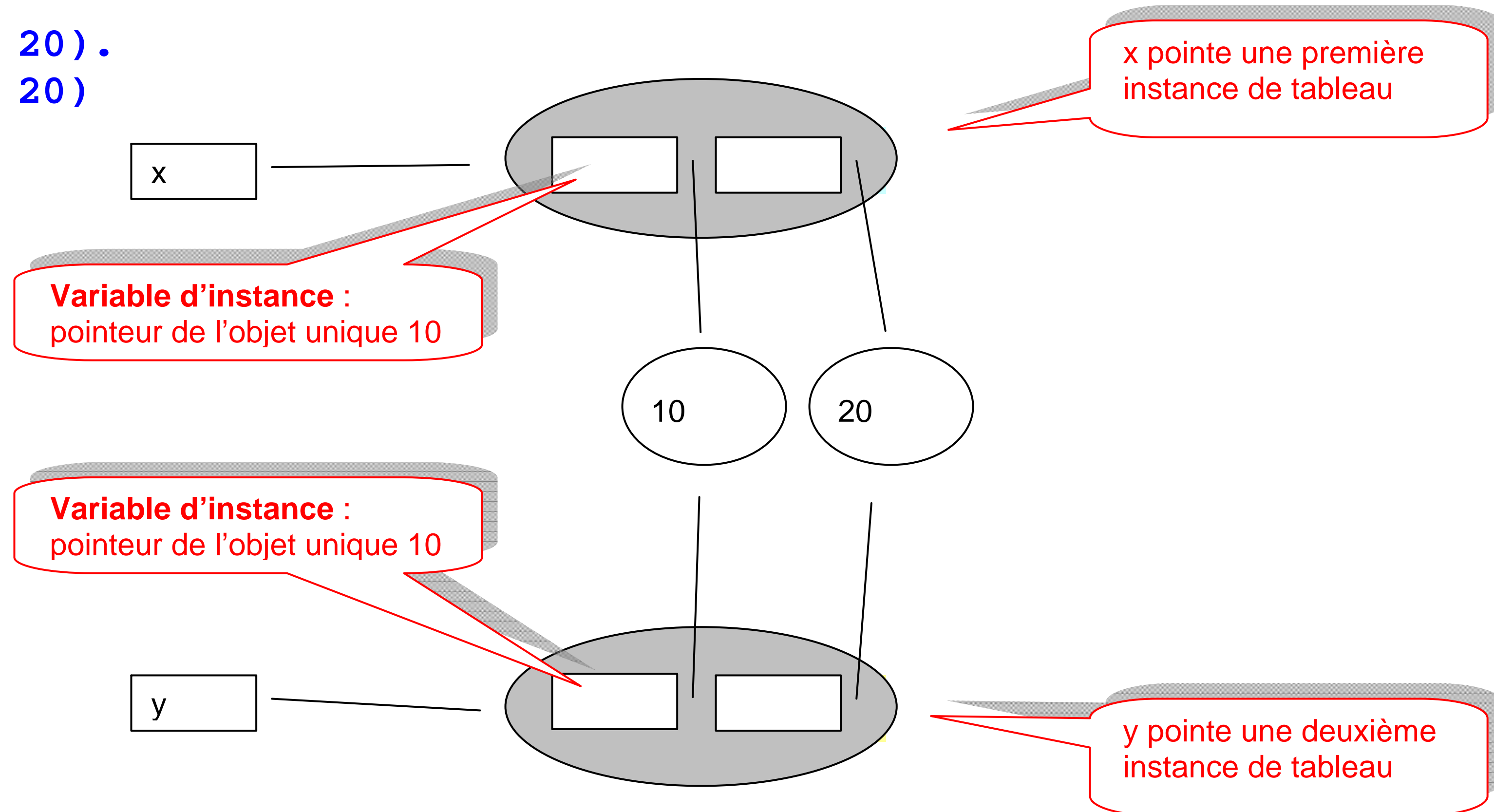
4. Affectations de variables (l'élément syntaxique :=)

4.1. Deux variables désignent le même objet primitif (simple)

```
x := 10.
y := 10.
```



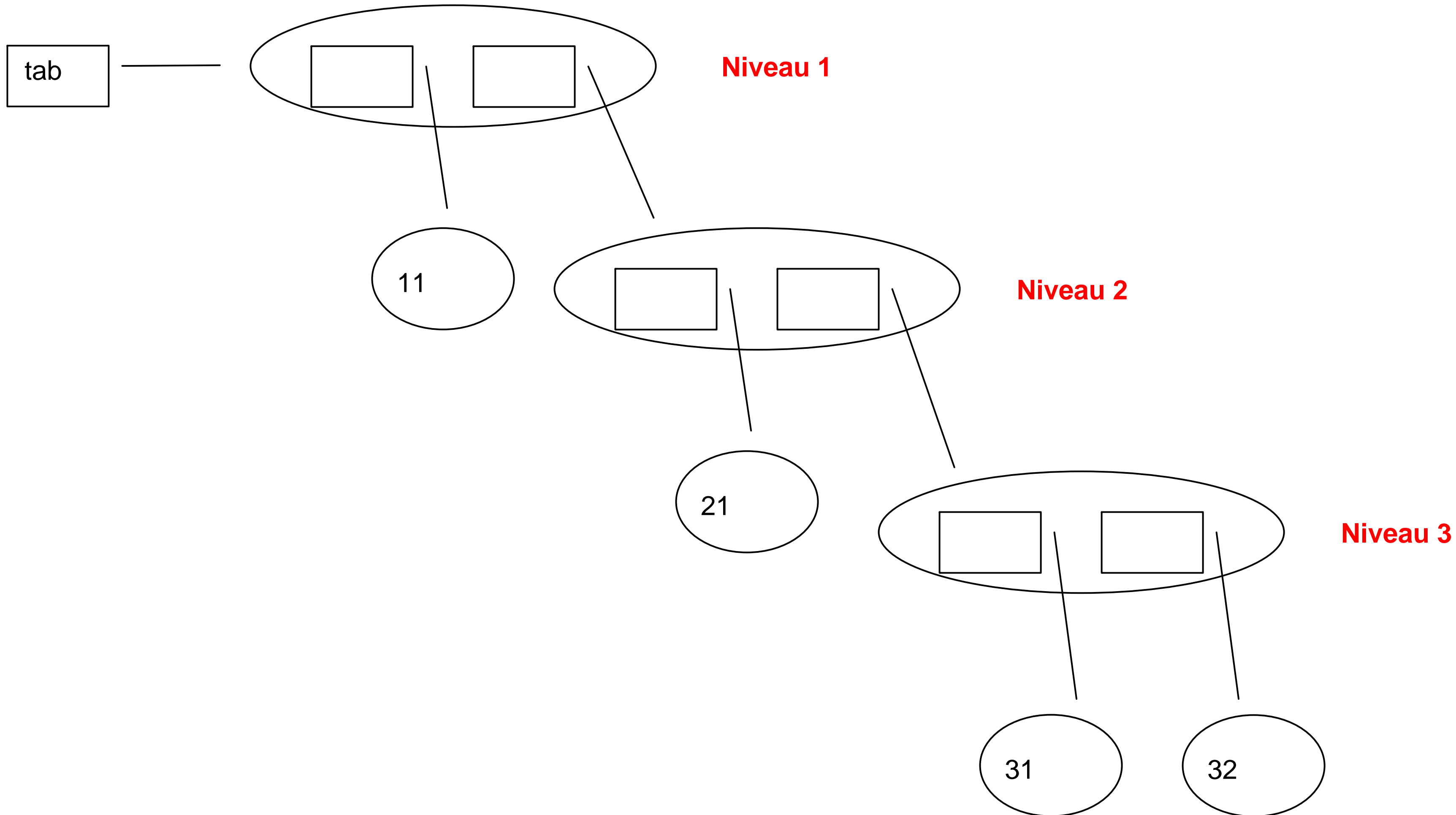
4.2. Deux variables désignent deux objets structurés distincts



Les variables d'instances de x et de y pointent les mêmes objets.

4.3. Désignation d'un objet structuré à plusieurs niveaux

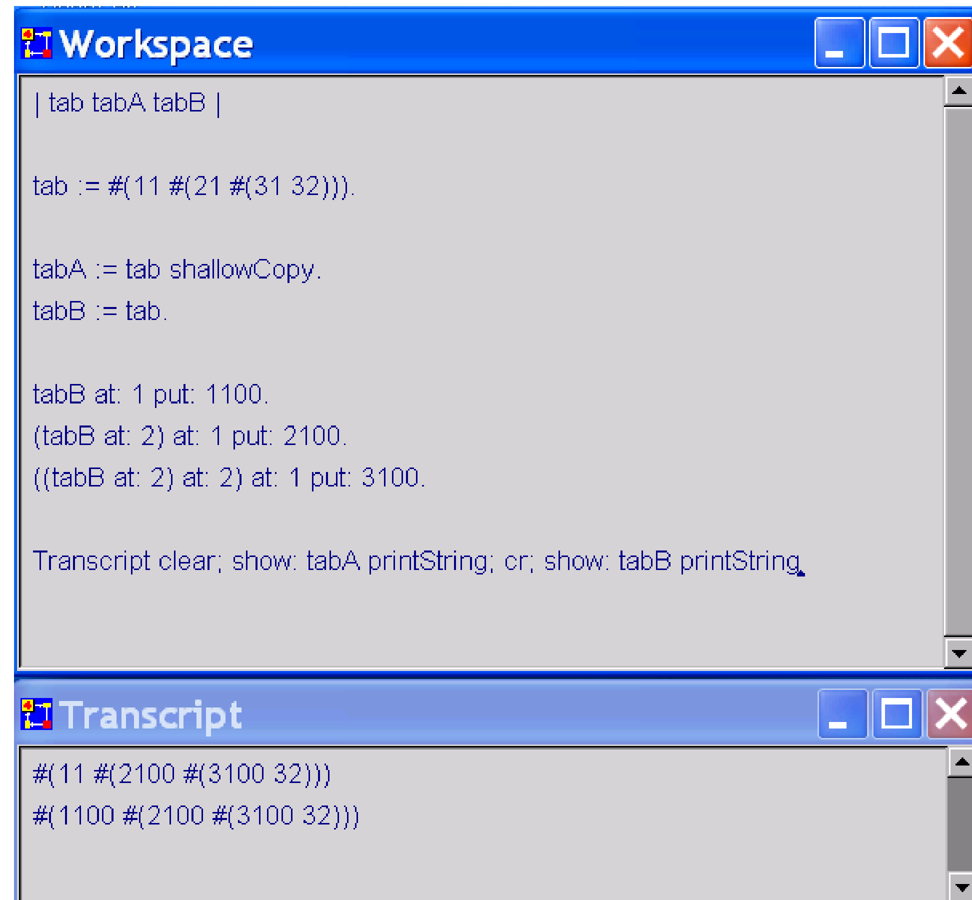
Soit un tableau initialisé : `tab := #(11 #(21 #(31 32)))`



5. Méthodes de copie d'objets

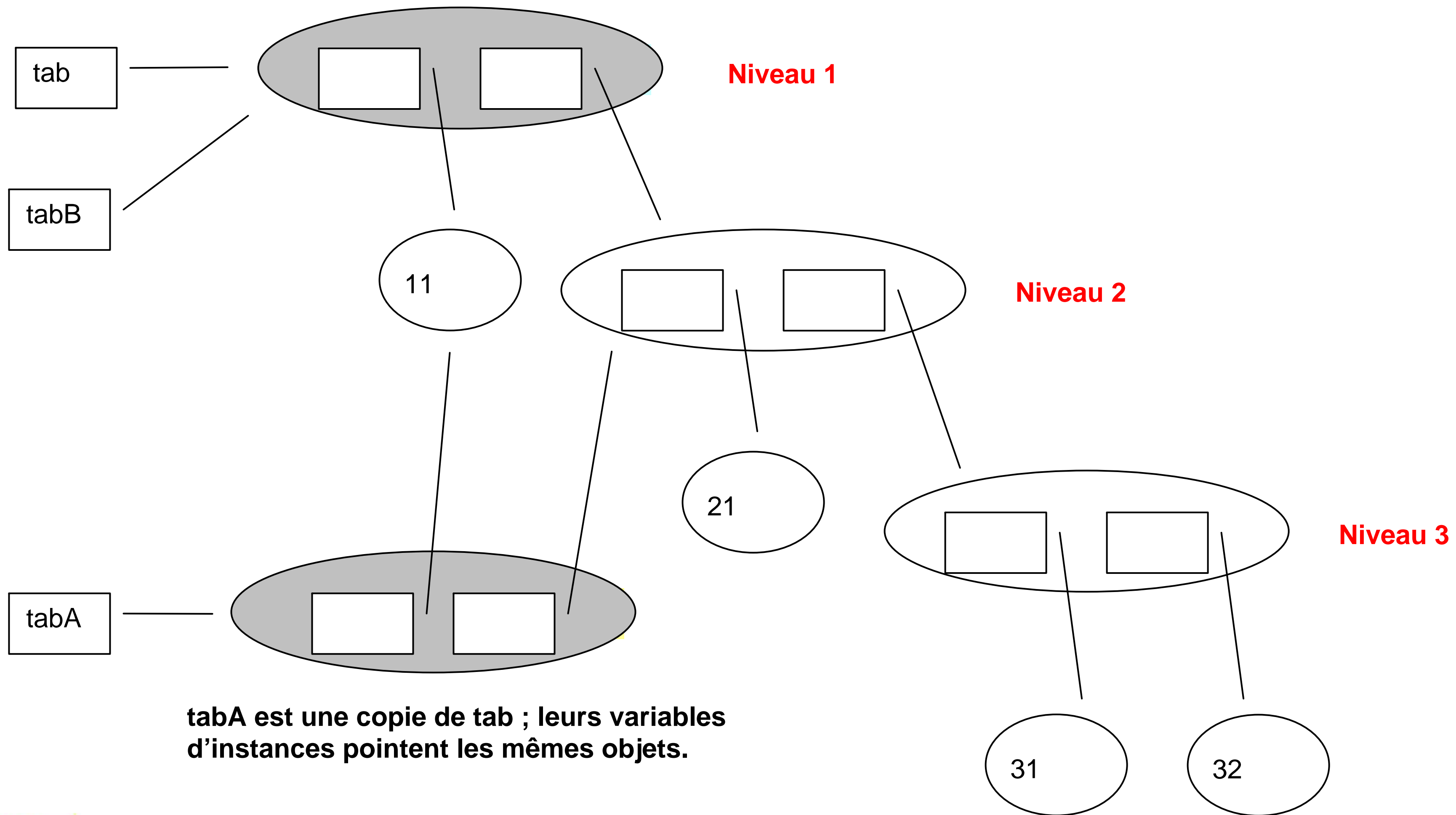
5.1. Copie superficielle (de premier niveau) : `shallowCopy` et `copy`

Les variables d'instance de la copie du receveur pointent les mêmes objets que l'original.



Résultats des messages suivants :

```
tab := #(11 #(21 #(31 32))).
tabA := tab shallowCopy.
tabB := tab
```



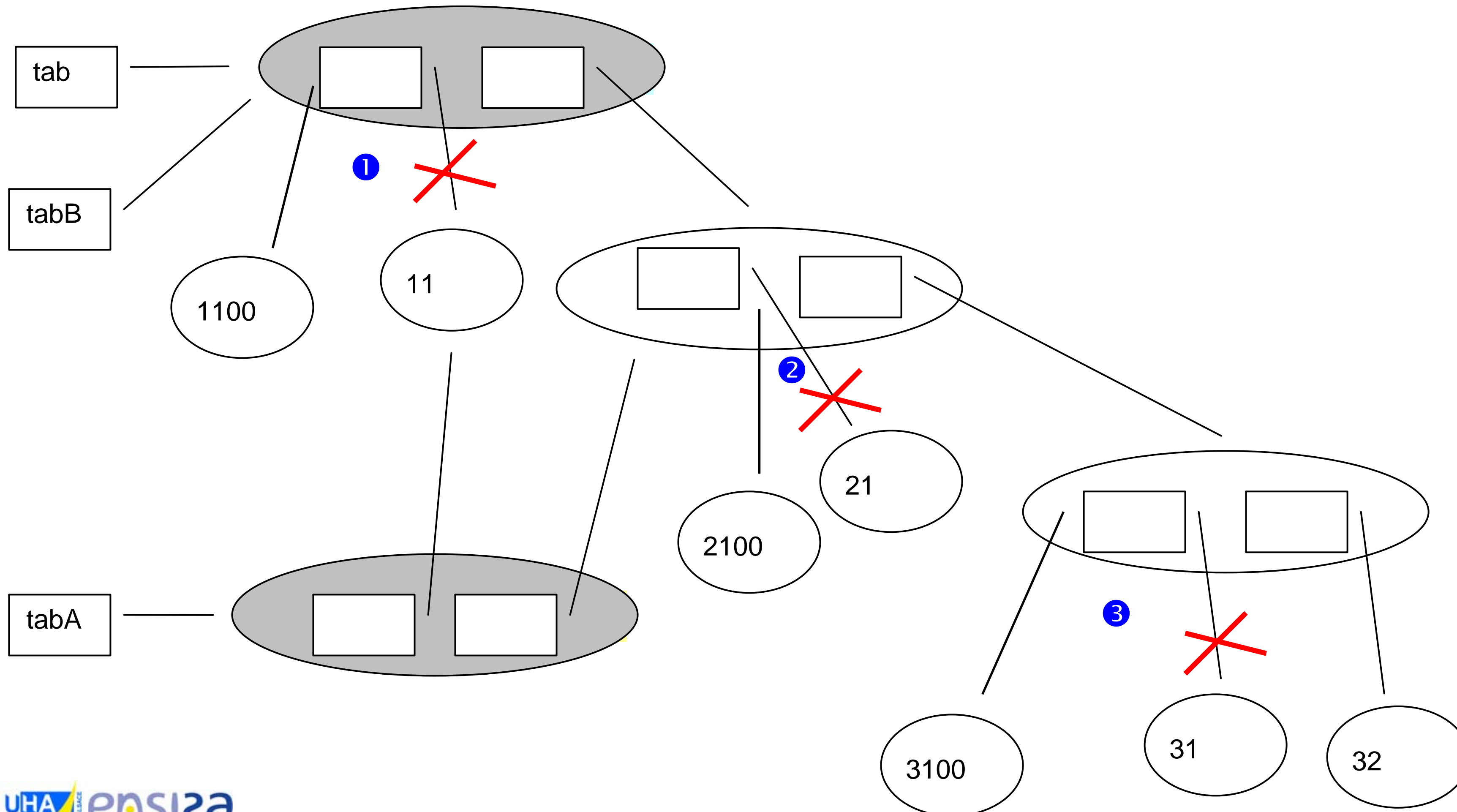
tabA est une copie de tab ; leurs variables d'instances pointent les mêmes objets.

Résultats des messages suivants :

❶ `tabB at : 1 put : 1100.`

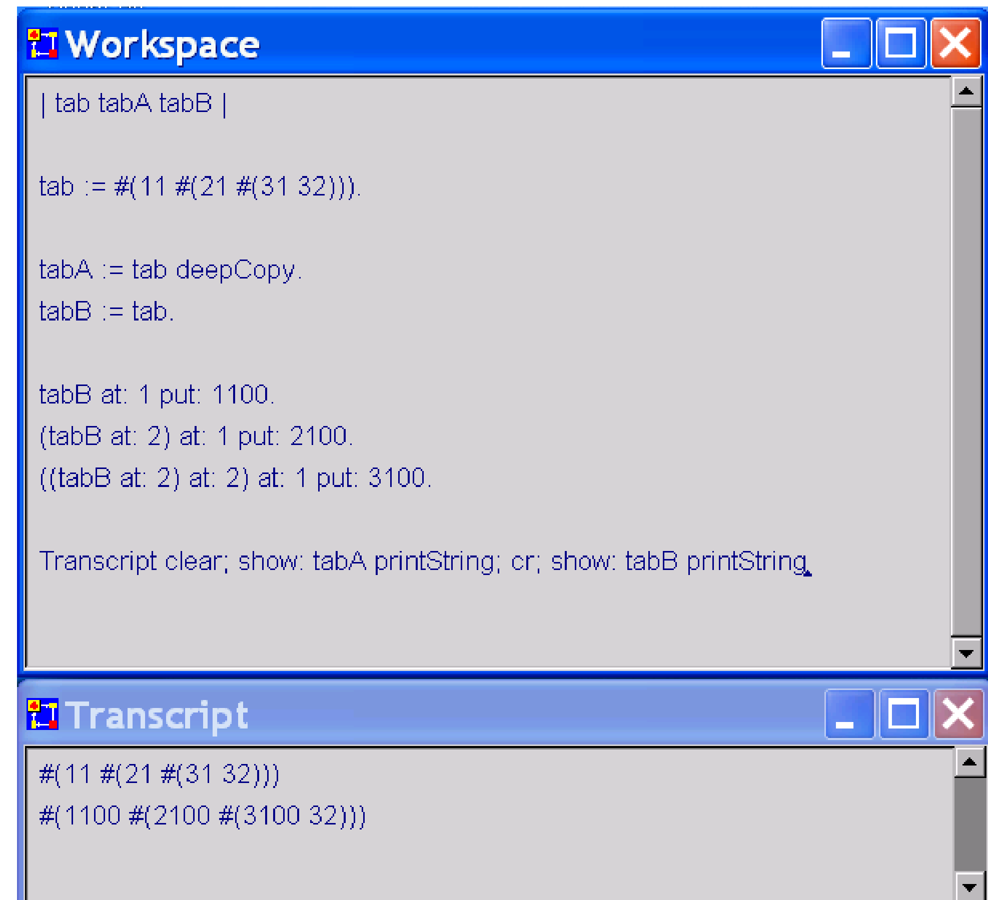
❷ `(tabB at :2) at : 1 put : 2100`

❸ `((tabB at :2) at : 2) at: 1 put : 3100`



5.2. Copie profonde (à tous les niveaux) : `deepCopy`

Les variables d'instance de la copie du receveur pointent des copies des objets sur lesquels pointaient les variables du receveur.



Remarque

- Dans PACE :
  - **`deepCopy`** copie sur tous les niveaux
- Dans SQUEAK :
  - **`deepCopy`** copie sur deux niveaux
  - **`veryDeepCopy`** copie sur tous les niveaux

6. Les blocs

- Les blocs contiennent des séquences d'expressions délimitées par deux crochets [ouvrant et fermant]
- **Un bloc est un objet**
    - la méthode **value** exécute le bloc et rend la valeur de la dernière expression contenue dans le bloc
    - la méthode **value:** exécute le bloc et transmet une valeur au paramètre d'un bloc
    - un bloc peut être référencé par une variable
    - un bloc peut être l'argument d'un message à mots clés

- **Valeur de la dernière expression d'un bloc sans paramètre**

```
[a := 10. b := 2. a + b] value
```

→ *12*

- **Bloc référencé par une variable ; valeur de la dernière expression d'un bloc sans paramètre**

```
n := 0.
v := [n := n + 1].
v value
```

→ *1*

MCours.com

- **Valeur de la dernière expression d'un bloc à 3 paramètres**

```
[:x :y :z | x + y + z] value:1 value:2 value: 3
```

→ *6*

- **Bloc référencé par une variable ; valeur de la dernière expression d'un bloc à 2 paramètres**

```
f := [:a :b | (a + b) factorial].
f value:1 value:2
```

→ *6*

- **Bloc référencé par une variable ; bloc à plus de 3 paramètres**

```
x := [:a :b :c :d :e | a + b + c + d + e].
x valueWithArguments: #(1 2 3 4 5)
```

→ *15*

7. Schémas conditionnels

- **Test d'une expression booléenne**

```
unBooléen   ifTrue: unBloc
            ifFalse: unBloc
```

```
tab := Array with:10 with:12 with:25 with: 40.
(tab at: 1) > 10
     ifTrue: [tab at:1 put: #sup10]
     ifFalse: [tab at: 1 put: #infegal10].
tab.
→ #(#infegal10 12 25 40 )
```

```
i := 5. j := 3.
valeurRetour := i > j
  ifTrue: [i * j]
  ifFalse: [0].
valeurRetour
→ 15
```

```
i := 5. j := 3.
i > j
  ifTrue: [valeurRetour := i * j]          "idem code précédent, en moins élégant"
  ifFalse: [valeurRetour := 0].
valeurRetour
→ 15
```

- **Réexécution d'un bloc tant que son résultat est faux.**

```
unBloc whileFalse
```

- **Réexécution d'un bloc tant que son résultat est vrai.**

```
unBloc whileTrue
```

Workspace

```
"Recherche du premier client dont la commande est terminée"
| index cmd client etat |
      index := 0.
cmd := Table named: 'CommandesClients'.
client := nil.
[
index := index + 1.
etat := cmd cellAt: 2@index.
etat = 'termine' ifTrue: [client := cmd cellAt: 3@index].
(etat = 'termine') | (index =3).
]
whileFalse.
client.

'Schneider'
```

CommandesClients

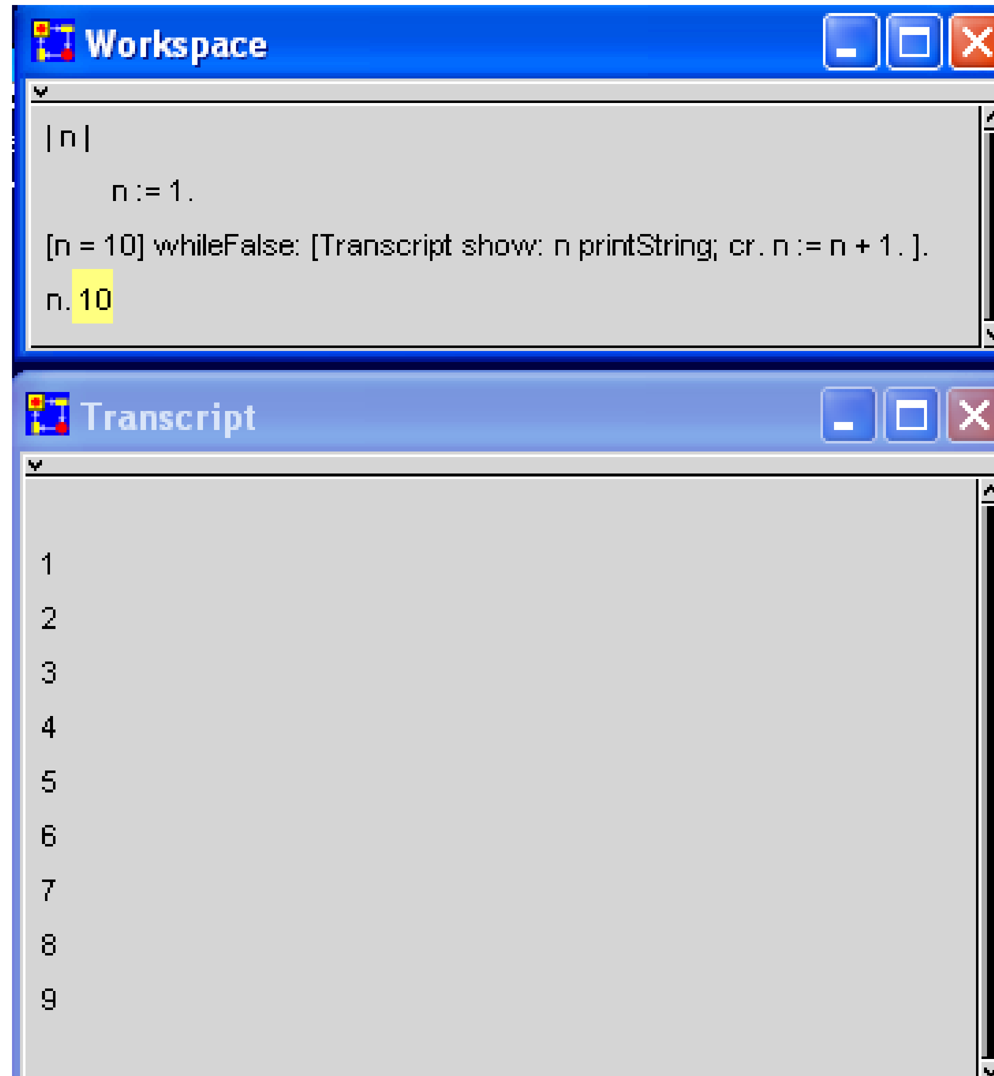
numero	etat	client
03234	attente	Lafarge
03235	termine	Schneider
03236	encours	Siemens

- **Réexécution du bloc argument tant que le résultat du bloc récepteur est faux.**

```
unBloc whileFalse: unBloc
```

- **Réexécution du bloc argument tant que le résultat du bloc récepteur est vrai.**

```
unBloc whileTrue: unBloc
```



8. Les itérations

8.1. Répétitions inconditionnelles

```
unNombre timesRepeat: unBloc
```

```
x := SortedCollection new.
5 timesRepeat: [x add: 1].
x.
→ SortedCollection (1 1 1 1 1 )
```

```
uneValeurInitiale to: uneValeurFinale do: unBlocÀunParamètre
unIntervalle do: unBlocÀunParamètre
```

```
x := 1.
1 to: 4 do: [:i | x := x + i].
x.
→ 11
```

Le paramètre (une variable)

```
uneValeurInitiale to: uneValeurFinale by: unIncrément do: unBlocÀunParamètre
```

```
x := 0.
1 to: 6  by: 2 do: [:i |x := x + i].
x.
→ 9
```

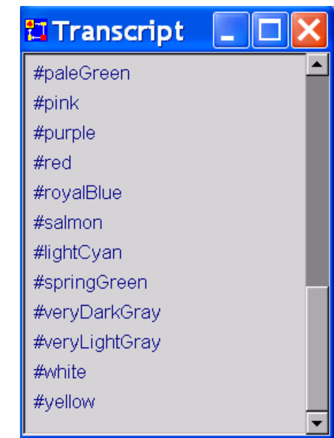
8.2. L'itérateur `do:` appliqué à une chaîne ou une collection

- **Exécution des instructions d'un bloc pour chaque élément d'une chaîne ou de d'une collection; l'élément courant étant pointé par l'unique paramètre du bloc**

```
uneChaîneDeCaractères do: unBlocAunParamètre
uneCollection do: unBlocAunParamètre
```

```
| colors |
colors := ColorValue constantNames.
colors do: [ :colorName | Transcript show: colorName printString; cr ].
```

→ *#(#black #blue #brown #chartreuse #cyan #darkCyan #darkGray #darkGreen #darkMagenta #darkRed #olive #gray #green #lightYellow #lightGray #magenta #navy #orange #orchid #paleGreen #pink #purple #red #royalBlue #salmon #lightCyan #springGreen #veryDarkGray #veryLightGray #white #yellow)*



8.3. L'itérateur `select:` appliqué à une collection

- **Création d'une nouvelle collection qui ne contient que les éléments qui répondent à une condition.**

```
uneCollection select: [ :chaqueElement | condition]
```

```
#(1 2 15 3 4 5 6 7) select: [ :i | i > 4].
```

→ *#(15 5 6 7)*

8.4. L'itérateur `reject:` appliqué à une collection

- **Création à partir d'une collection existante d'une nouvelle collection qui ne contient que les éléments qui ne répondent pas à une condition.**

```
uneCollection reject: [ :chaqueElement | condition]
```

```
#(1 2 15 3 4 5 6 7) reject:  [ :i | i > 4].
→ #(1 2 3 4 )
```

8.5. L'itérateur `collect:` appliqué à une collection

- **Création à partir d'une collection existante d'une nouvelle collection dont les nouveaux éléments sont évalués par un message.**

```
uneCollection collect: [ :chaqueElement | message]
```

```
#(1 2 15 3 4 5 6 7) collect:  [ :i | i > 4].
→ #(false false true false false true true true )

#(1 2 15 3 4 5 6 7) collect:  [ :i | i * i].
→ #(1 4 225 9 16 25 36 49 )
```

8.6. L'itérateur `detect:` appliqué à une collection

- **Obtention du premier élément d'une collection qui satisfait une condition.**

```
uneCollection detect: [ :chaqueElement | condition]
```

```
#(1 2 15 3 4 5 6 7) detect:  [ :i | i > 4].
→ 15
```

9. Les collections de données

9.1. Classes et sous-classes

- `Collection`
    - `SequenceableCollection`
        - `Interval`
        - `ArrayedCollection`
            - `Array`
            - `CharacterArray`
                - `String`
                - `Text`
        - `OrderedCollection`
            - `SortedCollection`
    - `Set`
        - `Dictionary`

9.2. Création d'instances de collections

- **Création d'une collection à taille fixe précisant le nombre d'éléments (initialement des références vers l'objet `nil`)**

```
uneClasseCollection new: unEntier
```

```
Array new: 15.
```

- **Création d'une collection vide** `uneClasseCollection new`

```
OrderedCollection new.
```

- **Initialisation d'une collection de données**

```
uneClasseCollection with: unObjet
uneClasseCollection with: unObjet with: unObjet
uneClasseCollection with: unObjet with: unObjet with: unObjet
uneClasseCollection with: unObjet with: unObjet with: unObjet with: unObjet
```

```
l := OrderedCollection with: 'aaa' with: #deux.
l.
→ OrderedCollection ('aaa' #deux )
```

UHA Université de Haute-Alsace Mulhouse-Colmar — ensisa école nationale supérieure d'ingénieurs sud alsace

9.3. Accès à une instance de collection

- **Rendre la valeur true si la collection est vide :** `uneCollection isEmpty`

```
l := OrderedCollection new.
l isEmpty.
→ true
```

- **Rendre la valeur true si la collection est non vide :** `uneCollection notEmpty`

```
#(10.5) notEmpty.
→ true

#() notEmpty
→ false
```

- **Rendre le nombre d'éléments de la collection :** `uneCollection size`

```
l := OrderedCollection with: 'aaa' with: #deux.
l size.
→ 2
```

9.4. Array

- Les tableaux sont des collections de données de longueur fixe dont les éléments sont accessibles par un index
- Ils sont utilisés lorsque le nombre d'éléments est connu à l'avance et que l'on accède souvent à ces éléments
- Les éléments d'un tableau peuvent être de nature et de longueur différentes

9.4.1. Création de tableaux

```
tab := Array new:10.
tab.

→ #(nil nil nil nil nil nil nil nil nil nil )
```

```
tab := Array with: 1 with: 'trois' with: 'jean' with: 4.3.
tab.

→ #(1 'trois' 'jean' 4.3 )
```

9.4.2. Accès à un tableau

- **Mémorisation d'une valeur à un emplacement précis d'un tableau**

```
unTableau at: unePosition put: unObjet
```

```
tab := Array new:4.
tab  at:1 put: #ok.
tab.
→ #(#ok nil nil nil )
```

- **Lecture d'une valeur à un emplacement précis d'un tableau**

```
unTableau at: unePosition
```

```
t := Array new:2.
t at:1 put: 'aa'; at: 2 put: 'bb'.
t at:1.
→ 'aa'
```

9.5. OrderedCollection

Les collections ordonnées ont des longueurs quelconques. On peut y ajouter des éléments (des objets) à volonté, en supprimer, ou les modifier.

- **Ajout d'un objet dans une collection ordonnée**

```
uneCollectionOrdonnée add: unObjet
```

- **Suppression de l'élément spécifié d'une collection ordonnée**

```
uneCollectionOrdonnée remove: unObjet
```

- **Suppression du premier élément d'une collection ordonnée et restitution de cet élément**

```
uneCollectionOrdonnée removeFirst
```

- **Restitution du premier élément d'une collection ordonnée**

```
uneCollectionOrdonnée first
```

- **Suppression du dernier élément d'une collection ordonnée et restitution de cet élément**

```
uneCollectionOrdonnée removeLast
```

- **Restitution du dernier élément d'une collection ordonnée**

```
uneCollectionOrdonnée last
```

- **Restitution de l'élément désigné par la position spécifiée**

```
uneCollectionOrdonnée at: unePosition
```

- **Remplacement de l'élément désigné par la position par l'objet spécifié**

```
uneCollectionOrdonnée at: unePosition put: unObjet
```

- **Suppression d'un objet d'une collection ordonnée; s'il est absent, évaluation d'un bloc**

```
uneCollectionOrdonnée remove: unObjet ifAbsent: unBloc
```

9.6. Association

Une association est un couple de deux objets, l'un appelé clé, l'autre appelé valeur.
Une association sert typiquement comme entrée dans un dictionnaire.

- **Déclaration d'une association littérale** `unObjetClé -> unObjetValeur`

```
a := #ubiquité -> 'faculté d''être dans plusieurs lieux'.
```

- **Accès à la clé d'une association** `uneAssociation key`

```
a key.
```
→ *#ubiquité*

- **Accès à la valeur d'une association** `uneAssociation value`

```
a value.
```
→ *'faculté d''être dans plusieurs lieux'*

9.7. Dictionary

- **Création d'un nouveau dictionnaire**

```
Dictionary new.
```

- **Insertion ou modification d'une association clé-valeur dans un dictionnaire**

```
unDictionnaire at: unObjetClé put: unObjetValeur.
```

- **Insertion d'une association clé-valeur dans un dictionnaire**

```
unDictionnaire add: unObjetClé -> unObjetValeur.
```

- **Obtention d'une valeur associée à une clé**

```
unDictionnaire at: uneClé.
```

- **Obtention de l'ensemble des clés d'un dictionnaire**

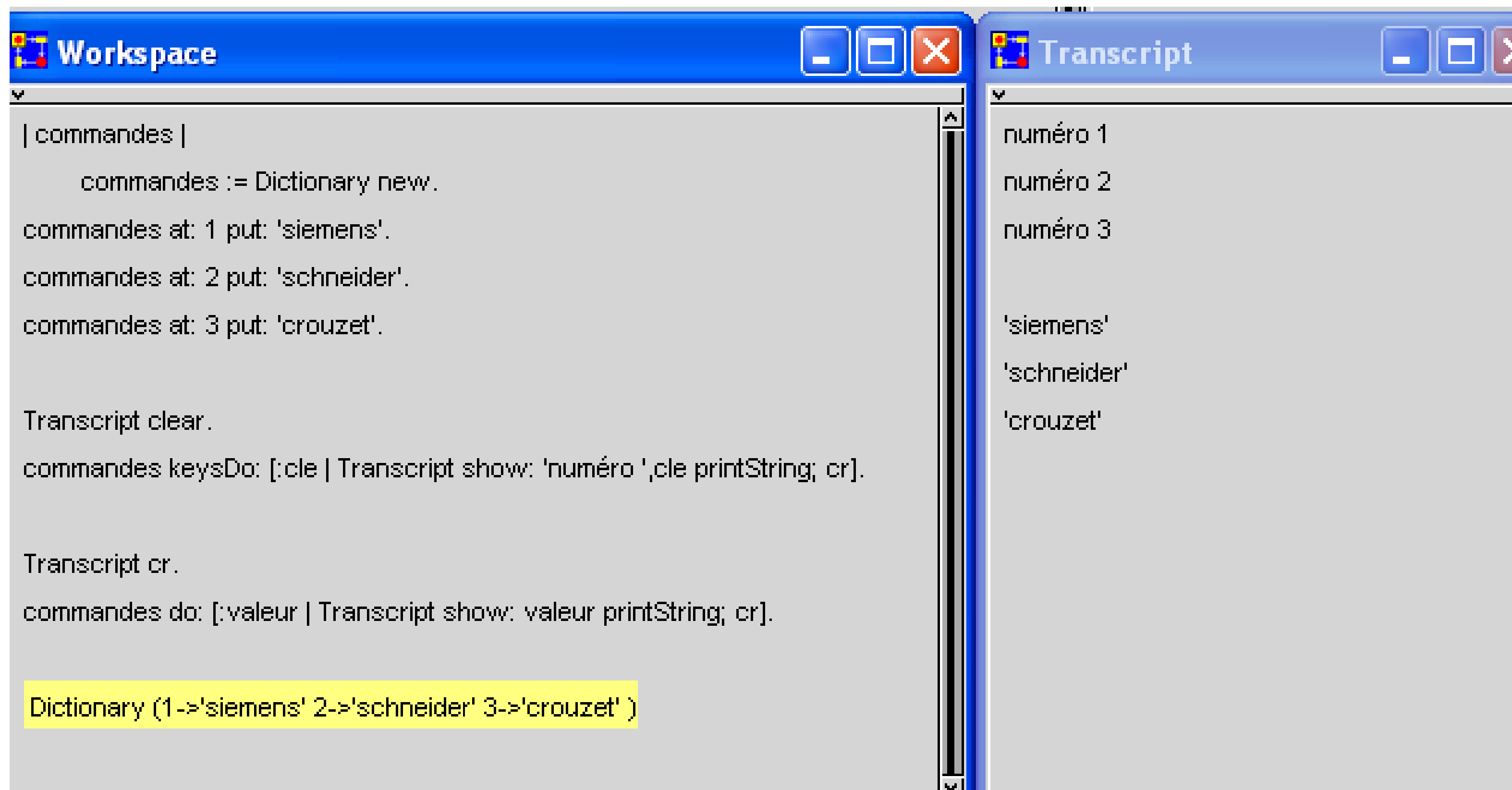
```
unDictionnaire keys.
```

- **Création d'une collection ordonnée des valeurs d'un dictionnaire**

```
unDictionnaire values.
```

9.7.1. Itérations à partir d'un dictionnaire

- **Exécution d'un bloc pour chaque valeur d'un dictionnaire** `unDictionnaire do: unBloc.`
- **Exécution d'un bloc pour chaque clé du dictionnaire.** `unDictionnaire keysDo: unBloc.`



9.7.2. Dictionnaire de fonctions

Initialization Code

```
Fonctions := Dictionary new.

Fonctions
    at: #setProd      put: [:p :d | self addTokenTo: (self placeNamed: p) with: d];
    at: #resetProd    put: [:p | (self placeNamed: p) marking removeAllTokens];
    at: #délaiProd    put: [:p | (Exponential mean: (((Table named: 'Produits') cellAt: 2@p) asNumber)) next].
```

DictionnaireDeFonctions

"Délai" 720
"Action"
(Fonctions at: #setProd) value: 'Prod1' value: 1.
(Fonctions at: #resetProd) value: 'Prod2'

()

"Délai" 720
"Action"
(Fonctions at: #setProd) value: 'Prod2' value: 2.
(Fonctions at: #resetProd) value: 'Prod1'

Prod1

(p)

(p)

"Délai"
(Fonctions at: #délaiProd) value: p

"Action"
date := CurrentTime

(date)

(781.06)
(1113.01)
(1174.74)
(1316.56)

Prod2
(2)

(p)

(p)

"Délai"
(Fonctions at: #délaiProd) value: p

"Action"
date := CurrentTime

(date)

(1698.0)

edit simulat deselect fixed

Produits

Numéro	DélaiMoyenFabrication
1	200
2	300

Time

1698.0