# Smalltalk Exercises

Alexandre Bergel, University of Berne
Noury Bouraqadi, Ecole des Mines de Douai
Marcus Denker, University of Berne
Catherine Dezan, Université de Brest
Stéphane Ducasse, Université de Savoie
Bernard Pottier, Université de Brest
Roel Wuyts, Université Libre de Bruxelles
And many others (please contact stef to update the list) Main Editor: S. Ducasse

November 6, 2005

# Contents

# Part I

# Robots and Fun

# 1 Fun with Bots Inc

Main Author(s): Stéphane Ducasse

The goal of this chapter is to get you started with some programming by steering graphical robots. We use the contents of the book Squeak: Learn programming with Robots by Stphane Ducasse, Apress Publishers, 2005 (http://smallwiki.unibe.ch/botsinc/).

More information on http://www.apress.com/book/bookDisplay.html?bID=444

## Getting the environment

Go to http://smallwiki.unibe.ch/botsinc/ and download the distribution for your machine. Follow the instructions described page http://smallwiki.unibe.ch/botsinc/installation/ (Unzip it and drap the file name ready.image on the exe file).

Watch the videos you can find at http://www.iam.unibe.ch/ ducasse/Web/BotsInc/Videos/ and reproduce them during the following exercises.

## 1.1 Create and talking

Create a robot and talk to it (video 1). You can get the list of action bringing the menu and selecting vocabulary.

- Draw a square of 200 pixels
- Draw a rectangle of 100 and 200 pixels

## 1.2 Writing Scripts

Direct interaction does not scale. So we propose you to write scripts as shown in the video 3.

- Write a script that draws a square of 200 pixels.
- Write a script that draws a rectangle of 100 and 200 pixels.

## 1.3 Looping

Read chapter 7 on loop http://smallwiki.unibe.ch/botsinc/chaptersamples/

- Using a loop write a script that draws a square of 100.
- Using a loop write a script that draws a rectangle of 100 and 200 pixels.
- Write a script that draws a stair case (experience 7.7)
- Try to reproduce Figure 7-11.

## 1.4 Variables

Watch video 5 that introduces the use of variables. There the variable l is incremented each step of the loop. Redo the experiment shown by the video. Using the same technique, draw a staircase whose step increase regularly. Do not watch at video 6.

## 1.5 Methods: Naming Scripts

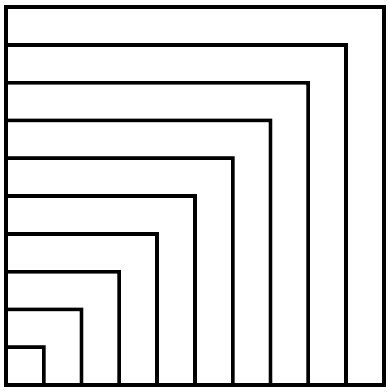
Scripts are powerful but difficult to reuse. First get a code browser (an editor to define method) (video 7, 8 and 9), then read the extra chapter available at http://www.iam.unibe.ch/∼ducasse/Teaching/CoursAnnecy/0506-MDSI/4916_Ch12_FINAL.pdf.

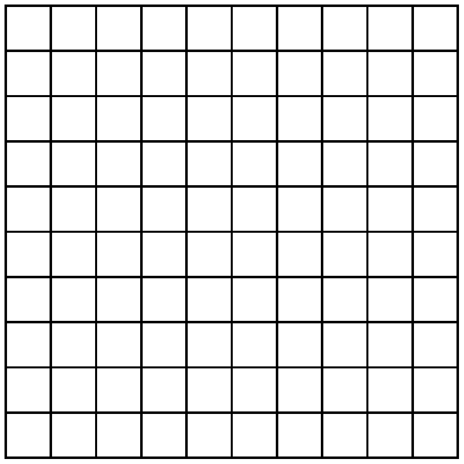
Follow the chapter and define the methods asked there and to the exercises.

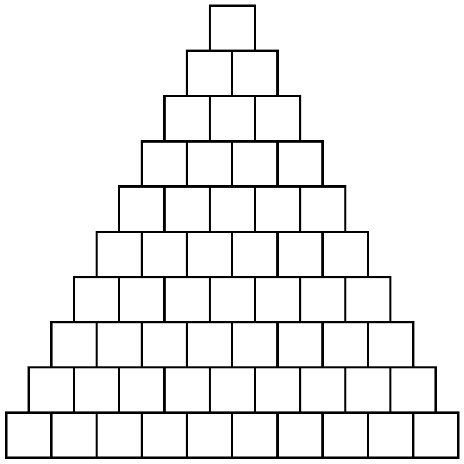
## 1.6 Composing methods

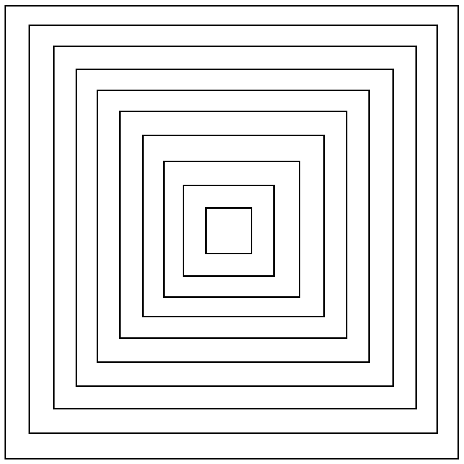
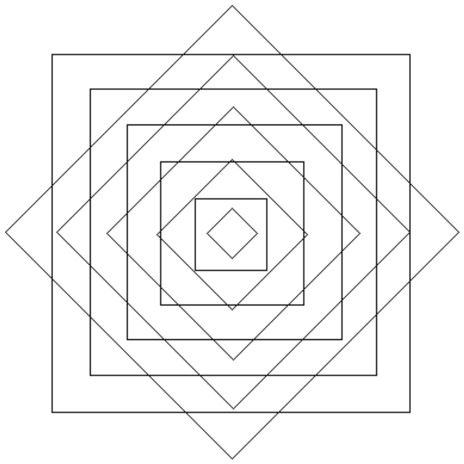
Read chapter 13 available http://smallwiki.unibe.ch/botsinc/chaptersamples/ and do the exercises.

Now using the method square define the following Figures:







Hints: it may be appropriate to define another method square that draws a square centered around a point.

## Part II

# Modeling on Paper

# 2

# Modeling, modeling...

Main Author(s): Stéphane Ducasse

## 2.1 Classes or Instances

**Classes/Instances.** What are instances and what are classes in the following lists:

- Les bijoux de la castafiore, Comix, Asterix le gaulois, Book, Novel, Fahrenheit 451, Da Vinci Code.
- Tokyo, Lisbon, Paris, Capital, Country, State, Madrid, France, Spain, Portugal
- Video, Book, ”le fabuleux destin d’amelie Poulain”, Client, ”la Bible”, ”Squeak”, StarWars4, StarWars5, Dupont, Dupond. GhostInTheShell, LeVoyageDeChihiro.

## 2.2 Classification

Classify the following abstractions: Opera, Film, Book, Novel, Comics. What could be a possible common superclass? To check if your hierarchy makes sense here is a guideline: It should be possible to substitute any instance of a superclasse by any instance of a subclass without breaking a program of the superclass.

## 2.3 Capitals and countries

- How do you relate the following entities: Lisbon, Paris, Capital, Country, State, Madrid, France, Spain, Portugal?
- What are the possible relationships from countries to capitals and vice versa? Write some examples.
- How would you characterize countries and states in the context of international income?

## 2.4 A Basic LAN Application

Identify the classes and their possible responsibilities in the following example. Note that responsibilities are not state or properties of the objects but their behavior.

Imagine a local network simulator **Local Area Network (LAN)**. Workstations, nodes, printers are linked together and form a ring. Each node (workstations, nodes, printers) is pointing to a next node forming a ring. When a packet is received by a node it checks if this is for him, if this is the case, it performs appropriate actions (printing the contents of the packet for a printer), displaying it on a screen if this is a workstation). Only workstation can emit packet to reach other workstations or printers. When a workstation or a printer received a packet that is sent to it it does not forward it to the next node of the network. A packet identifies who sent it, ist addressee and its contents.

## 2.5 Mail

A mail has a title, one text, one sender, multiple recipients, and multiple attached files. What are the possible classes and their relationships.

## 2.6 Hotel Reservation

Monsieur Formulain, directeur d'une chaîne d'hôtels, vous demande de concevoir une application de gestion pour ses hôtels. Voici ce que vous devez modéliser :

Un hôtel Formulain est constitué d'un certain nombre de chambres. Un responsable de l'hôtel gère la location des chambres. Chaque chambre se loue à un prix donné (suivant ses prestations).

L'accès aux salles de bain est compris dans le prix de la location d'une chambre. Certaines chambres comportent une salle de bain, mais pas toutes. Les hôtes de chambres sans salle de bain peuvent utiliser une salle de bain sur le palier. Ces dernières peuvent être utilisées par plusieurs hôtes.

Les pièces de l'hôtel qui ne sont ni des chambres, ni des salles de bain (hall d'accueil, cuisine...) ne font pas partie de l'étude (hors sujet).

Des personnes peuvent louer une ou plusieurs chambres de l'hôtel, afin d'y résider. En d'autre termes : l'hôtel héberge un certain nombre de personnes, ses hôtes (il s'agit des personnes qui louent au moins une chambre de l'hôtel...).

## 2.7 Classes/Instances.

Describe with the best accuracy the information that you would like to have to describe a video for an online catalog. You can get inspired by http://www.imdb.com/ or http://wrapper.rottentomatoes.com/.

Now from the list StarWars4, StarWars5, DVDStarWarsBoite35, DVDStarWarsBoite35, Video, K7, LeVoyageDeChihiro, BoiteNoireLeVoyageDeChihiro145, BoiteNoireLeVoyageDeChihiro146, how would you identify instances and classes. For this item we suggest you to read http://students.engr.scu.edu/ gvenkata/typeobjectpattern.htm.

## 2.8 Redo the Exercises using UML notation

# Part III

# First Contact

# 3 Smalltalk Environment Basics

Main Author(s): Ducasse and Wuyts

## 3.1 Videos

You can find videos showing some important points of Smalltalk manipulation at http://www.iam.unibe.ch/ ducasse/Videos/SqueakO

## 3.2 Starting up

Similarly to Java, Smalltalk the source code is translated to byte-codes, which are then interpreted and executed by the Smalltalk Virtual Machine. (Note that this is an approximation because Smalltalk dialects were also the first languages to develop Just in Time compilation, i.e., a method is compiled into byte-codes but also into native code that is directly called instead of executing the byte codes.)

There are three important files:

**Squeak.image** (Binary): contains byte code of all the object of the system, the libraries and the modifications you made.

**Squeak.changes** (ASCII): contains all the modifications made in the image-file since this was created. It contains also all the code of the actions you will perform.

**SqueakV3.sources** (ASCII): contains the textual code of the initial classes of the system.

To open an image with drag-drop (on Macintosh, Windows):

- Drag the file 'visual.im' on the virtual machine to start the image.
- If you want to start your own image, just double click on it or drag it over the virtual machine.

On Unixes (Linux, Solaris, HPUX, AIX, ...): you should invoke the virtual machine passing it an image as parameter. For the first opening, execute the first script that per default uses the original image the script is installation dependent but should look like path/bin/visualworks path/image/visual.im Then after you can specify your own image.

**Creating a fresh image.** We are going to create an image for this lesson.

- select Save As... in the menu
- when the system prompts you for the name for the new image, you type *lesson*, followed by your username.
- the image is saved in the image directory.
- Have a look at the Transcript, and note what it says.

**About the mouse.** Smalltalk was the first application to use multiple overlapping windows and a mouse. It extensively uses three mouse buttons, that are context sensitive and can be used everywhere throughout Smalltalk:

- the left mouse button is the select button
- the middle button is the operate button
- the right button is the window button

On a Macintosh, where only one button is available, you have to use some keyboard keys together with pressing your mouse button:

- the select button is the one button itself
- for the operate button, press the button while holding the alt-key pressed
- for the window button, press the button while holding the apple-key pressed

## 3.3 Adding Goodies and setting Preferences

Out of the box, there is already quite some code in a Smalltalk image (about 1000 classes containing the basic system: the complete compiler, parser classes, GUI framework, development environment, debugger, collection libraries, etc.) [1]. But for developing, it is convenient to load some extra tools.

Using the open... Squeak Map Package Loader entry, you can load the following tools: Shout, eCompletion, Refactoring Browser. Normally we already loaded them in the image you are using.

## 3.4 Selecting text, and doing basic text manipulations

One of the basic manipulations you do when programming is working with text. Therefore, this section introduces you to the different ways you can select text, and manipulate these selections.

The basic way of selecting text is by clicking in front of the first character you want to select, and dragging your mouse to the last character you want in the selection while keeping the button pressed down. Selected text will be highlighted.

**Exercise 0** Select some parts of text in the Transcript. You can also select a single word by double clicking on it. When the text is delimited by ” (single quotes), ”” (double quotes), () (parentheses), [] (brackets), or  (braces), you can select anything in between by double clicking just after the first delimiter.

**Exercise 1** Try these new selection techniques.

Now have a look at the text operations. Select a piece of text in the Transcript, and bring on the operate menu. Note that you have to keep your mouse button pressed to keep seeing the window.

**Exercise 2** Copy this piece of text, and paste it after your selection. Afterwards cut the newly inserted piece of text.

**Exercise 3** See if there is an occurrence of the word visual in the Transcript. Note that to find things in a text window, there is no need to select text. Just bring up the operate menu .

**Exercise 4** Replace the word visual with C++ using the replace operation (if it does not contain Smalltalk, add this word or replace something else). Take your time and explore the different options of the replace operation.

**Exercise 5** Bring up the operate menu, but don’t select anything yet. Press and hold the shift button, and select paste in the operation menu. What happens ?

[1] To answer a common question: yes, there are ways to strip this so that you can deploy smaller images to clients that do not contain all of these tools.

## 3.5 Opening a WorkSpace Window

We will now open a workspace window, a text window much like the Transcript, you use to type text and expressions and evaluate them.

To open a workspace:

- open the left flap or bring the world menu and choose open...
- You will see a framing rectangle (with your mouse in the upper left corner), that indicates the position where the Workspace will open. Before you click, you can move your mouse around to change this position. Click one time once you have found a good spot for your Workspace.
- Now your mouse is in the bottom right corner, and you can adjust the size. If you click once more, once you have given it the size you like, the Workspace window appears.

This is the basic way of opening many kinds of applications. Experiment with it until you feel comfortable with it.

## 3.6 Evaluating Expressions

In the Workspace, type : 3. Select it, and bring up the operate menu. In the operate you will see the next three different options for evaluating text and getting the result:

**do it:** do it evaluates the current selection, and does not show any result of the execution result.

**print it:** prints the result of the execution after your selection. The result is automatically highlighted, so you can easily delete it if you want to.

**inspect it:** opens an inspector on the result of the execution.

The distinction before these three operations is essential, so check that you REALLY understand their differences **Exercise 6** Select 3, bring up the operate menu, and select print it.

**Exercise 7** Print the result of 3+4

**Exercise 8** Type Date today and print it. Afterwards, select it again and inspect it.

After exercise 9, you will have an inspector on the result of the evaluation of the expression Date today (this tells Smalltalk to create an object containing the current date). This Inspector Window consists of two parts: the left one is a list view containing self (a pseudo variable containing the object you are inspecting) and the instance variables of the object. Right is a text field.

**Exercise 9** Click on self in the inspector. What do you get? Does it resemble the result shown by printstring?

**Exercise 10** Select day. What do you get? Now change this value, bring up the operate menu, and select accept it. Click again on self. Any difference?

**Exercise 11** In the inspector edit field, type the following: self weekday, select it and print it. This causes the message weekday to be sent to self (i.e., the date object), and the result is printed. Experiment with other expressions like:

```
self daysInMonth
self monthName
```

Close the inspector when you are finished.

**Exercise 12** Type in the Workspace the following expression: Time now, and inspect it. Have a look at self and the instance variables.

**Exercise 13** Type in the Workspace the following expression: Time dateAndTimeNow. This tells the system to create an object representing both today's date and the current time, and open an inspector on it. Select the item self in the inspector. [Note that self is an object called an Array. It holds on to two other objects (elements 1 and 2). You can inspect each element to get either the time or the date object.

**Using the System Transcript.** We have already seen that the Transcript is a text window where the system informs you important information. You can also use the Transcript yourself as a very cheap user interface.

If you have a Workspace open, place it so that it does not cover the System Transcript. Otherwise, open one and take care of where you put it. Now, in the Workspace, type:

```
Transcript cr.
Transcript show: 'This is a test'.
Trancript cr.
```

Select these 3 lines and evaluate (do It) them with do it. This will cause the string This is a test to be printed in the Transcript, preceeded and followed by a carriage return. Note that the argument of the show: message was a literal string (you see this because it is contained in single quotes). It is important to know, because the argument of the show: method always has to be a string. This means that if you want any non-string object to be printed (like a Number for example), you first have to convert it to a string by sending the message printString to it. For example, type in the workspace the following expression and evaluate it:

Transcript show: 42 printString, 'is the answer to the Universe'. Note here that the comma is used to concatenate the two strings that are passed to the show: message 42 printString and 'is the answer to the Universe'.

**Exercise 14** Experiment on your own with different expressions. Transcript cr ; show: This is a test ; cr Explain why this expression gives the same result that before. What is the semantics of ; ?

# 4

# Objects and expressions

This lesson is about reading and understanding Smalltalk expressions, and differentiating between different types of messages and receivers. Note that in the expressions you will be asked to read and evaluate, you can assume that the implementation of methods generally corresponds to what their message names imply (i.e., `2 + 2 = 4`).

**Exercise 15** For each of the Smalltalk expressions below, fill in the answers:

```
3 + 4
```

- What is the receiver object?
- What is the message selector?
- What is/are the argument (s)?
- What is the message?
- What is the result returned by evaluating this expression?

```
Date today
```

- What is the receiver object?
- What is the message selector?
- What is/are the argument (s)?
- What is the message?
- What is the result returned by evaluating this expression?

```
anArray at: 1 put: 'hello'
```

- What is the receiver object?
- What is the message selector?
- What is/are the argument (s)?
- What is the message?
- What is the result returned by evaluating this expression?

**Exercise 16** What kind of object does the literal expression `'Hello, Dave'` describe?

**Exercise 17** What kind of object does the literal expression `#Node1` describe?

**Exercise 18** What kind of object does the literal expression `#(1 2 3)` describe?

**Exercise 19** What can one assume about a variable named `Transcript`?

**Exercise 20** What can one assume about a variable named `rectangle`?

**Exercise 21** Examine the following expression:

```
| anArray |
anArray := #('first' 'second' 'third' 'fourth').
anArray at: 2
```

What is the resulting value when it is evaluated (ˆ means return)? What happens if you remove the ˆ. Explain

**Exercise 22** Which sets of parentheses are redundant with regard to evaluation of the following expressions:

```
((3 + 4) + (2 * 2) + (2 * 3))

(x isZero)
  ifTrue: [....]
(x includes: y)
  ifTrue: [....]
```

**Exercise 23** Guess what are the results of the following expressions

```
6 + 4 / 2
1 + 3 negated
1 + (3 negated)
2 raisedTo: 3 + 2
2 negated raisedTo: 3 + 2
```

**Exercise 24** Examine the following expression:

```
25@50
```

- What is the receiver object?
- What is the message selector?
- What is/are the argument (s)?
- What is the message?
- What is the result returned by evaluating this expression?

**Exercise 25** Examine the following expression and write down the sequence of steps that the Smalltalk system would take to execute the following expression:

```
Date today daysInMonth
```

**Exercise 26** Examine the following expression and write down the sequence of steps that the Smalltalk system would take to execute the following expression:

```
Transcript show: (45 + 9) printString
```

**Exercise 27** Examine the following expression and write down the sequence of steps that the Smalltalk system would take to execute the following expression:

```
5@5 extent: 6.0 truncated @ 7
```

**Exercise 28** During lecture, we saw how to write strings to the Transcript, and how the message printString could be sent to any non-string object to obtain a string representation. Now write a Smalltalk expression to print the result of 34 + 89 on the Transcript. Test your code !

**Exercise 29** Examine the block expression:

```
| anArray sum |
sum := 0.
anArray := #(21 23 53 66 87).
anArray do: [:item | sum := sum + item].
sum
```

What is the final result of sum ? How could this piece of code be rewritten to use explicit array indexing (with the method at: ) to access the array elements[1]? Test your version. Rewrite this code using inject:into:

[1]Note this is how you would proceed with Java or C++

# 5
# Les objets de Smalltalk-80

Main Author(s): to be fixed: B. Pottier, Université de Brest, Bernard.Pottier@univ-brest.fr

## 5.1 Observation des objets et règles de priorité

**Vocabulaire.** Nous utilisons le terme evaluer et inspecter pour deux actions distinctes. Evaluer : selectionner une zone et faire “print it”. Inspecter : selectionner une zone et faire “inspect”

### 5.1.1 Inspecter les expressions suivantes

```
1
2.0
$a
’une chaine’
1@2
1.0@2.0
7/2
```

Parmi les messages, on distingue

**les messages unaires** comme new, sin, sqrt, size, first, last, negated)

**les messages binaires** `+ - * / ** // \\ < <= > >= = ˜= == ˜˜ & | @ ,`

**les message à mot clé** comme at: put:, x: y:, bitOr:, bitAnd:

Dans une expression, on évalue en priorité en respectant le parenthésage, les messages unaires puis binaires puis à mots clés. Si l’expression ne comporte que des messages de même priorité, l’évaluation se fait classiquement de la gauche vers la droite.

### 5.1.2 Evaluer et inspecter les expressions suivantes.

```
7.0/2.0
1 + 1
(1 + 1) printString
(1/2) class
```

Expliquer pourquoi le parenthésage est obligatoire dans les expressions précédentes.

Attention, il n’y a pas de priorité entre opérateurs, la méthode + est juste traitée comme n’importe quelle autre méthode. l’évaluation suit l’ordre des messages.

Evaluer :

```
2 + 3*4
2 + (3*4)
2  + 1/2
2 + (1/2)
```

### 5.1.3 Uniformité des messages.

Un même message peut être adressé à des objets de types différents. Evaluer et inspecter :

```
2 sqrt
2.0 sqrt
(3/2) sqrt
(3/5) + (6/7)
```

### 5.1.4 Arithmétique exacte et conversion de type.

Evaluer :

```
(1111111111111111111/11111111111111111112) + (1/11111111111111111112)
2r1000
16rFF
256 printStringRadix:2

(8 bitOr: 1) printStringRadix:2
(8 bitAnd: 9)
2e10 asInteger
```

Faire le ou bit à bit des nombres binaires 1010 et 0011 et donner le résultat en binaire.

### 5.1.5 Les variables.

Evaluer :

```
| x |
x := 2.

| x |
x := 2 + 1/2.

| x |
x := 2 + 1/2.
x := 2.
```

Quelle est la valeur de x après évaluation de la dernière portion de code?

### 5.1.6 Les tableaux sont aussi des objets.

Evaluer :

```
| table |
table := #(1 2.0 'trois' 4444444444444444444444444444444444).
table := #(1 3 6 9).
table first.
table last.
table reverse.

| string |
string := '#(1 3 6 9)'.
string first.
string last.
string reverse.
```

```
#(10 20 30 40) at: 2

| table |
table := #(1 3 6 9).
table at:1 put: (3/4)

| string |
string := '#(1 3 6 9)'.
string at: 4 put: $r
```

### 5.1.7 Liste des objets et des messages.

Donner la liste des objets et des messages définis dans les expressions ci-dessous. Expliquer le résultat de l'évaluation.

```
| aPoint |
aPoint:= Point x:2 y:1.
aPoint x: aPoint x * 2

| x |
x:=1.5.
x negated rounded.
Fraction  numerator: x*2 denominator: 3 + x negated rounded.
```

## 5.2 Tableaux

1. Multiplier par 2 le 2 ème élément d'un tableau,
2. Remplacer la valeur du 2 ème élément d'un tableau par son opposé.
3. Remplacer la valeur du 3 ème élément par la valeur du 2 ème élément.
4. Remplacer la valeur du 3 ème élément par la somme des 2 ème et 3ème (ancienne valeur) éléments.
5. Le 2 ème du tableau étant une fraction, remplacer cette fraction par la fraction inverse dans le tableau.

## 5.3 Nombres

### 5.3.1 Maximum

La méthode `max: unAutreNombre` appliqué à un nombre renvoie le plus grand des deux nombres.
Exemple : `2 max: 6` renvoie `6`.

1. Calculer le maximum de 3 variables `a b c` contenant des valeurs quelconques.
2. Calculer le maximum de 3 variables `a b c` contenant des valeurs quelconques **sans utiliser de variables intermédiaires**

### 5.3.2 Conversion Celsius-Fahrenheit

La formule de conversion Celsius-Fahrenheit est : C = (5/9) (F - 32).
Convertir une variable contenant un nombre (en degrés Fahrenheit), en degrés Celsius.

# 6
# Les blocs et les enumerateurs

Main Author(s): to be fixed: B. Pottier, Université de Brest, Bernard.Pottier@univ-brest.fr

## 6.1 Commençons

On affiche une chaîne dans le Transcript en lui envoyant le message show: uneChaine. On passe à la ligne dans le Transcript en lui envoyant le message cr
Exemple : Transcript show: ’Salut’. Transcript cr.
en utilisant la cascade (plusieurs messages envoyés au même objet)
Transcript show: ’Salut’; cr.

### 6.1.1 Les blocs

L’évaluation d’un bloc sans parametre s’obtient en envoyant le message value au bloc. Les instructions du bloc sont exécutées et le résultat de la dernière instruction est retourné.

Un bloc avec une variable s’évalue en envoyant le message value: uneValeur.
Un bloc avec deux variables s’évalue en envoyant le message
value: unePremiereValeur value: uneDeuxiemeValeur (ceci jusqu’à 4 variables).

1. Ecrire un bloc (sans variables) qui calcule la racine carrée de 1, puis de 2, puis de 3. Evaluer ce bloc.
2. Ecrire un bloc avec une variable, qui renvoie le carré de cette variable. Evaluer ce bloc.
3. Ecrire un bloc avec deux variables, qui renvoie la plus grande de ces deux variables (utiliser la méthode max:). Evaluer ce bloc.

### 6.1.2 Les méthodes d’intervalle

Les nombres comprennent les messages suivants : to: uneValeurArret do: unBloc
et : to: uneValeurArret by: unPas do: unBloc

Exemple qui affiche les nombres de 1 à 10 dans le Transcript
1 to: 10 do: [ :i |Transcript show: i printString]

Exemple qui affiche, par pas de 2, les nombres de 1 à 10 dans le Transcript
1 to: 10 by: 2 do: [ :i |Transcript show: i printString]

1. Convertir le nombre 65 (représentant un code ASCII) en Character (asCharacter), puis en Symbol (asSymbol), puis en String (asString).
2. Afficher dans le Transcript les caractères compris entre les codes ASCII 65 et 122 (bornes incluses).
3. Afficher les nombres impairs de 1 à 33 dans le Transcript

### 6.1.3 Les énumérateurs

Toutes les sous-classes de Collection comprennent ces messages, appelés *énumérateurs* :

- do: unBloc évalue unBloc sur chaque élément de la collection,
- collect: unBloc comme do: mais renvoie une collection des résultats,
- select: unBloc évalue unBloc sur chaque élément et renvoie ceux pour qui l'évaluation renvoie true,
- reject: unBloc évalue unBloc sur chaque élément et renvoie ceux pour qui l'évaluation renvoie false,
- detect: unBloc renvoie le premier élément pour qui l'évaluation renvoie true,
- detect: unBloc ifNone: unAutreBloc a le même comportement que detect: mais permet d'exécuter le deuxième bloc (unAutreBloc) s'il n'y a pas d'élément pour qui l'évaluation de unBloc renvoie true.
- inject: uneValeur into: unBlocBinaire injecte le résultat de l'exécution précédente de unBlocBinaire (un bloc à deux paramètres) dans la suivante.
  uneCollection inject: valeur into: [arg1 arg2 |...]
  arg1 est initialisé avec valeur,
  arg2 prend successivement la valeur de chaque élément et évalue le bloc avec cette valeur (comme un do:),
  à l'issue de l'évaluation courante, le résultat de l'évaluation est affectée dans arg1.

**L'énumérateur do:**

1. Faire la somme des éléments d'un tableau.
2. La méthode constantNames envoyée à la classe ColorValue renvoie un tableau **constant** de symboles, chaque symbole ayant le nom d'une couleur.

   Afficher, dans le Transcript, les couleurs **constantes** de la classe ColorValue

**L'énumérateur collect:**

1. Construire un premier tableau, **et avec la méthode collect:**, construire un deuxième tableau identique au premier.
2. A partir d'un premier tableau, construire un deuxième tableau dont la valeur de chaque élément est le double de l'élément correspondant dans le premier tableau.

**Autres énumerateurs**

1. A partir du tableau constant de symboles ayant le nom d'une couleur #(noir bleu rouge rose blanc vert), détecter la première couleur commençant par un r
   (on obtient le premier caractère d'un symbole avec first).
2. A partir du tableau constant de symboles ayant le nom d'une couleur, construire un tableau des noms de couleur commençant par un r
3. A partir du tableau constant de symboles ayant le nom d'une couleur, construire un tableau des noms de couleur ne commençant pas par un r
   Afficher (avec un do:) ce tableau dans le Transcript

### 6.1.4 Les structures alternatives

Smalltalk définit sur la classe True et sur la classe False quatre méthodes d'instance
ifTrue:, ifFalse:, ifTrue:ifFalse:, ifFalse:ifTrue: [1].

| | Classe True | Classe False |
|---|---|---|
| Méthode:<br>Action: | ifTrue: unBloc<br>renvoyer l'évaluation de unBloc | ifTrue: unBloc<br>renvoyer nil |
| Méthode:<br>Action: | ifFalse: unBloc<br>renvoyer nil | ifFalse: unBloc<br>renvoyer l'évaluation de unBloc |
| Méthode:<br>Action: | ifTrue: unBloc ifFalse: unAutreBloc<br>renvoyer l'évaluation de unBloc | ifTrue: unBloc ifFalse: unAutreBloc<br>renvoyer l'évaluation de unAutreBloc |
| Méthode:<br>Action: | ifFalse: unBloc ifTrue: unAutreBloc<br>renvoyer l'évaluation de unAutreBloc | ifFalse: unBloc ifTrue: unAutreBloc<br>renvoyer l'évaluation de unBloc |

Par ailleurs, les opérateurs logiques & (conjonction, le ET), Eqv (équivalence), not (négation), xor (ou exclusif), | (disjonction, le OU) sont définies sur les classes True et False.

1. Tester si un nombre est impair (en lui envoyant le message odd) et sonner la cloche (Smalltalk beep ) si c'est vrai.
2. Sans utiliser max:, écrire un bloc avec deux paramètres qui renvoie le maximum des deux paramètres

### 6.1.5 Enumérateurs et alternatives

1. Faire la somme des éléments positifs d'un tableau
2. Créer un tableau de 0 ou 1 à partir d'un tableau existant, un nombre du tableau existant est remplacé par un 0 s'il est supérieur ou égal à 10, par un 1 s'il est inférieur à 10.

### 6.1.6 Itération de blocs

Utilisation des messages timesRepeat:, repeat, whileTrue:

1. Ecrire 10 fois la chaîne 'coucou' dans le Transcript (en passant à la ligne après chaque 'coucou').
2. Ecrire un bloc avec une variable n qui écrit n fois la chaîne 'coucou' dans le Transcript (en passant à la ligne après chaque 'coucou'). Evaluer ce bloc.
3. Itérer avec un repeat: un bloc qui incrémente un compteur de 1. On s'arrête quand le compteur est supérieur à 10.
4. Créer un objet de la classe Time à 3 secondes du temps courant.
   Time now addTime: (Time fromSeconds: 3)
   Boucler jusqu'à ce que le temps courant dépasse cet objet, en passant à la ligne dans le Transcript

## 6.2 Exercices

### 6.2.1 Les blocs

1. Ecrire un bloc avec une variable, qui renvoie la conversion en degrés Celsius de cette variable (supposée être en Fahrenheit). Evaluer ce bloc (C = (5/9) (F - 32)).
2. Ecrire un bloc avec deux variables, qui convertit ces variables en String, les concatène en les séparant avec un blanc et renvoie le résultat de la concaténation.

[1] Un truc : Si on tape $< Control > t$ (ou $< Control > f$) dans une fenêtre de code, le système insère ifTrue: (ifFalse:)

### 6.2.2 Les énumérateurs

#### L'énumérateur do:

1. Compter le nombre d'éléments d'un tableau. Vérifier avec size
2. Faire la moyenne des éléments d'un tableau.

#### L'énumérateur collect:

1. Une chaine instance de la classe cString est un tableau, donc on peut avoir son premier élément (first), son i-ème élément (at: i), affecter uneValeur dans son i-ème élément at: i put: uneValeur, etc

   A partir du tableau **constant** de symboles ayant le nom d'une couleur, construire un tableau de String où le nom de la couleur commence par une majuscule (asUppercase).
2. Construire un tableau dont la valeur des éléments est la somme de l'élément précédent avec l'élément courant.

### 6.2.3 Les structures alternatives

1. Sans utiliser max:, écrire un bloc avec trois paramètres qui renvoie le maximum des trois paramètres
2. Ecrire un bloc avec un paramètre, qui teste si le paramètre est une minuscule (isLowercase), une majuscule (isUppercase), un chiffre (isDigit) ou autre.
   Le bloc renvoie 'minuscule' ou 'majuscule' ou 'chiffre' ou 'autre'.

### 6.2.4 Les intervalles

#### Méthodes d'intervalles

1. Afficher les 10 premiers carrés.
2. Afficher les multiples de 10.

#### Créer et utiliser des intervalles

Envoyé à un nombre, la méthode to: borneSuperieure renvoie un Interval allant du nombre à la borneSuperieure par pas de 1.

Envoyé à un nombre, la méthode to: borneSuperieure by: lePas renvoie un Interval allant du nombre à la borneSuperieure par pas de lePas.

Les Interval étant des collections, on peut utiliser les énumérateurs do:, collect:, etc.

Exemple : tab := (1 to: 100) crée un interval des 100 premiers entiers.

1. Calculer la somme des 100 premier nombres entiers à l'aide d'un interval. Vérifier avec la formule n*(n+1)/2
2. Créer un interval avec les nombres de $0$ à $360$ de $30$ en $30$. Utiliser cet interval pour afficher la table des sinus de $30$ degrés en $30$ degrés.

### 6.2.5 Enumérateurs et alternatives

1. A partir d'un premier tableau, construire un tableau en inversant tous les éléments qui sont des fractions.

### 6.2.6 Autres énumerateurs

1. A partir d'un premier tableau, construire un deuxième tableau en supprimant les éléments négatifs.
2. Définir la somme des éléments d'un tableau avec inject:into:.
3. Définir la somme des éléments positifs d'un tableau avec select: puis inject:into:.
4. Définir la somme des éléments négatifs d'un tableau avec reject: puis inject:into:.

### 6.2.7 Itération de blocs

1. Afficher 10 * dans le Transcript, puis passe à la ligne.
2. Afficher 10 lignes de 10 * dans le Transcript (en passant à la ligne après chaque ligne de 10 *.
3. Ecrire un bloc à deux paramètres ctl et c, qui affiche l lignes de c colonnes de * dans le Transcript (en passant à la ligne après chaque ligne).

# 7 Counter Example

Main Author(s): Bergel, Ducasse, Wuyts

## 7.1 A Simple Counter

We want you to implement a simple counter that follows the small example given below. Please note that we will ask you to define a test for this example.

```
| counter |
counter := SimpleCounter new.
counter increment; increment.
counter decrement.
counter value = 1
```

## 7.2 Creating your own class

In this part you will create your first class. In traditional Smalltalk environments a class is associated with a category (a folder containing the classes of your project).

The steps we will do are the same ones every time you create a class, so memorize them well. We are going to create a class SimpleCounter in a category called DemoCounter. Figure 7.1 shows the result of creating such a category.

### 7.2.1 Creating a Class category

In the System Browser, click on the left pane and select *add*. The system will ask you a name. You should write DemoCounter. This new category will be created and added to the list.

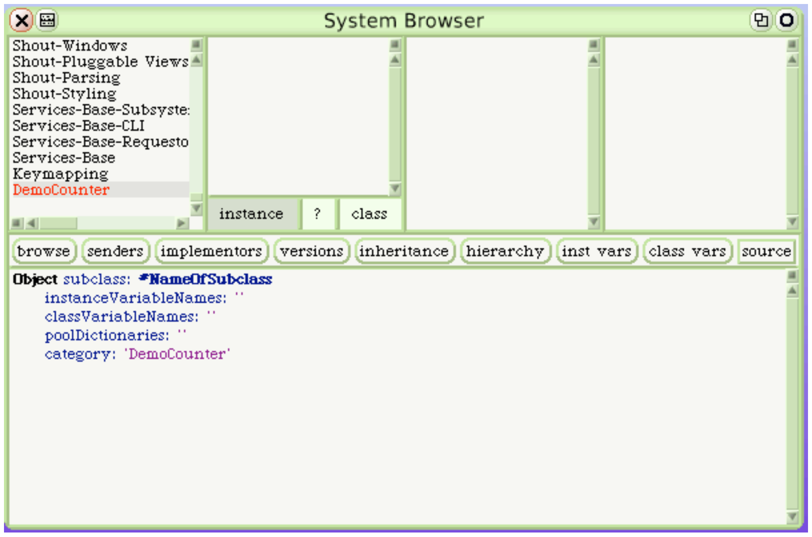
### 7.2.2 Creating a Class

Creating a class requires five steps. They consist basically of editing the class definition template to specify the class you want to create.

1. **Superclass Specification**. First, you should replace the word NameOfSuperclass with the word Object. Thus, you specify the superclass of the class you are creating. Note that this is not always the case that Object is the superclass, since you may to inherit behavior from a class specializing already Object.

2. **Class Name**. Next, you should fill in the name of your class by replacing the word NameOfClass with the word SimpleCounter. Take care that the name of the class starts with a capital letter and that you do not remove the # sign in front of NameOfClass.

3. **Instance Variable Specification**. Then, you should fill in the names of the instance variables of this class. We need one instance variable called value. You add it by replacing the words *instVarName1* and *instVarName2* with the word value. Take care that you leave the string quotes!



```
Object subclass: #NameOfSubclass
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'DemoCounter'
```

Figure 7.1: Your category is created.

4. **Class Variable Specification**. As we do not need any class variable make sure that the argument for the class instance variables is an empty string (classInstanceVariableNames: ”).

5. **Compilation**. That’s it! We now have a filled-in class definition for the class SimpleCounter. To define it, we still have to **compile** it. Therefore, select the **accept** option from the operate menu (right-click button of the mouse). The class SimpleCounter is now compiled and immediately added to the system.

As we are disciplined developers, we provide a comment to SimpleCounter class by clicking **Comment** button of the class definition . You can write the following comment:

---

```
SimpleCounter is a concrete class which supports incrementing
and decrementing a counter.

Instance Variables:

value           <Integer>
```

---

Select **accept** to store this class comment in the class.

## 7.3 Defining protocols and methods

In this part you will use the System Browser to learn how to add protocols and methods.

### 7.3.1 Creating and Testing Methods

The class we have defined has one instance variable value. You should remember that in Smalltalk, everything is an object, that instance variables are private to the object and that the only way to interact with an object is by sending messages to it.

Therefore, there is no other mechanism to access the instance variables from outside an object than sending a message to the object. What you can do is to define messages that return the value of the instance variable of a class. Such methods are called **accessors**, and it is a common practice to always define and use them. We start to create an accessor method for our instance variable value.

Remember that every method belongs to a protocol. These protocols are just a group of methods without any language semantics, but convey important navigation information for the reader of your class. Although protocols can have any name, Smalltalk programmers follow certain conventions for naming these protocols. If you define a method and are not sure what protocol it should be in, first go through existing code and try to find a fitting name.

**An important remark:** *Accessors* can be defined in protocols accessing or private. Use the accessing protocol when a client object (like an interface) really needs to access your data. Use private to clearly state that no client should use the accessor. This is purely a convention. There is no way in Smalltalk to enforce access rights like *private* in C++ or Java. To emphasize that objects are not just data structure but provide services that are more elaborated than just accessing data, put your accessors in a private protocol. As a good practice, if you are not sure then define your accessors in a private protocol and once some clients really need access, create a protocol accessing and move your methods there. Note that this discussion does not seem to be very important in the context of this specific simple example. However, this question is central to the notion of object and encapsulation of the data. An important side effect of this discussion is that you should always ask yourself when you, as a client of an object, are using an accessor if the object is really well defined and if it does not need extra functionality.

**Exercise 30** Decide in which protocol you are going to put the accessor for value. We now create the accessor method for the instance variable value. Start by selecting the class DemoCounter in a browser, and make sure the **Instance** button is selected . Create a new protocol clicking the right-button of the mouse on the pane of methods categories, and choosing New, and give a name. Select the newly created protocol. Then in the bottom pane, the edit field displays a method template laying out the default structure of a method. Replace the template with the following method definition:

```
value
   ”return the current value of the value instance variable”

   ˆvalue
```

This defines a method called value, taking no arguments, having a method comment and returning the instance variable value. Then choose **accept** in the operate menu (right button of the mouse) to compile the method. You can now test your new method by typing and evaluating the next expression in a Workspace, in the Transcript, or any text editor SimpleCounter new value.

This expression first creates a new instance of SimpleCounter, and then sends the message value to it and retrieves the current value of value. This should return nil (the default value for noninitialised instance variables; afterwards we will create instances where value has a reasonable default initialisation value).

**Exercise 31** Another method that is normally used besides the *accessor* method is a so-called *mutator* method. Such a method is used to *change* the value of an instance variable from a client. For example, the next expression first creates a new SimpleCounter instance and then sets the value of value to 7:

```
SimpleCounter new value: 7
```

This mutator method does not currently exist, so as an exercise write the method value: such that, when invoked on an instance of SimpleCouter, the value instance variable is set to the argument given to the message. Test your method by typing and evaluating the expression above.

**Exercise 32** Implement the following methods in the protocol operations.

```
increment
  self value: self value + 1
decrement
  self value: self value - 1
```

**Exercise 33** Implement the following methods in the protocol printing

```
printOn: aStream
   super printOn: aStream.
   aStream nextPutAll: ’ with value: ’,
   self  value printString.
   aStream cr.
```

Now test the methods increment and decrement but pay attention that the counter value is not initialized. Try:

```
SimpleCounter new value: 0; increment ; value.
```

Note that the method printOn: is used when you print an object or click on self in an inspector.

### 7.3.2 Adding an instance initialization method

Now we have to write an initialization method that sets a default value to the value instance variable. However, as we mentioned the initialize message is sent to the newly created instance. This means that the initialize method should be defined at the instance side as any method that is sent to an instance of SimpleCounter like increment and decrement. The initialize method is responsible to set up the instance variable default values.

Therefore at the instance side, you should create a protocol initialize-release, and create the following method (the body of this method is left blank. Fill it in!).

```
initialize
  ”set the initial value of the value to 0”
```

Now create a new instance of class SimpleCounter. Is it initialized by default? The following code should now work without problem:

```
SimpleCounter new increment
```

### 7.3.3 Another instance creation method

If you want to be sure that you have really understood the distinction between instance and class methods, you should now define a different instance creation method named withValue:. This method receives an integer as argument and returns an instance of SimpleCounter with the specified value. The following expression should return 20.

```
(SimpleCounter withValue: 19) increment ; value
```

**A Difficult Point** Let us just think a bit! To create a new instance we said that we should send messages (like new and basicNew) to a class. For example to create an instance of SimpleCounter we sent new to SimpleCounter. As the classes are also objects in Smalltalk, they are instances of other classes that define the structure and the behavior of classes. One of the classes that represents classes as objects is Behavior. Browse the class Behavior. In particular, Behavior defines the methods new and basicNew that are responsible of creating new instances. If you did not redefine the new message locally to the class of SimpleCounter, when you send the message new to the class SimpleCounter, the new method executed is the one defined in Behavior. Try to understand why the methods *new* and *basicNew* are on the instance side on class Behavior while they are on the class side of your class.

## 7.4 SUnit

For the advanced ones, we suggest you to look at the videos and download the tutorial SUnit explained from http://www.iam.unibe.ch/∼ducasse/Books.html. Then define a TestCase with several tests for the SimpleCounter class. To open the test runner execute

```
TestRunner open
```

## 7.5 Saving your Work

Several ways to save your work exist: You can

- Save the class by clicking on it and selecting the fileout menu item.
- Use the Monticello browser to save a package

8

# A Simple Application: A LAN simulation

Main Author(s): Ducasse, Wuyts

## Basic LAN Application

The purpose of this exercise is to create a basis for writing future OO programs. We work on an application that simulates a simple **Local Area Network (LAN)**. We will create several classes: Packet, Node, Workstation, and PrintServer. We start with the simplest version of a LAN, then we will add new requirements and modify the proposed implementation to take them into account.

### Creating the Class Node

The class Node will be the root of all the entities that form a LAN. This class contains the common behavior for all nodes. As a network is defined as a linked list of nodes, a Node should always know its next node. A node should be uniquely identifiable with a name. We represent the name of a node using a symbol (because symbols are unique in Smalltalk) and the next node using a node object. It is the node responsibility to send and receive packets of information.

---

Node inherits from Object
Collaborators: Node and Packet
Responsibility:
name (aSymbol) - returns the name of the node.
hasNextNode - tells if a node has a next node.
accept: aPacket - receives a packet and process it.
By default it is sent to the next node.
send: aPacket - sends a packet to the next node.

---

**Exercise 34** Create a new category LAN, and create a subclass of Object called Node, with two instance variables: name and nextNode.

**Exercise 35** Create accessors and mutators for the two instance variables. Document the mutators to inform users that the argument passed to name: should be a Symbol, and the arguments passed to nextNode should be a Node. Define them in a private protocol. Note that a node is identifiable via its name. Its name is part of its public interface, so you should move the method name from the private protocol to the accessing protocol (by drag'n'drop).

**Exercise 36** Define a method called hasNextNode that returns whether the node has a next node or not.

**Exercise 37** Create an instance method printOn: that puts the class name and name variable on the argument aStream. Include my next node's name ONLY if there is a next node (Hint: look at the method printOn: from previous exercises or other classes in the system, and consider that the instance variable name is a symbol and nextNode is a node). The expected printOn: method behavior is described by the following code:

```
(Node new
  name: #Node1 ;
  nextNode: (Node new name: #PC1)) printString

Node named: Node1 connected to: PC1
```

**Exercise 38** Create a **class** method new and an **instance** method initialize. Make sure that a new instance of Node created with the new method uses initialize (see previous exercise). Leave initialize empty for now (it is difficult to give meaningful default values for the name and nextNode of Node. However, subclasses may want to override this method to do something meaningful).

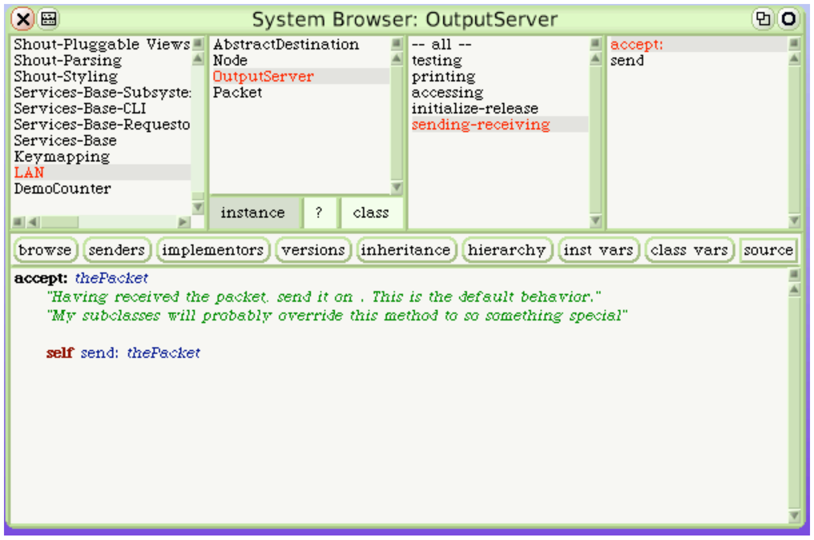
**Exercise 39** A node has two basic messages to send and receive packets. When a packet is sent to a node, the node has to accept the packet, and send it on. Note that with this simple behavior the packet can loop infinitely in the LAN. We will propose some solutions to this issue later. To implement this behavior, you should add a protocol send-receive, and implement the following two methods -in this case, we provide some partial code that you should complete in your implementation:

```
accept: thePacket
 ”Having received the packet, send it on. This is the default
behavior My subclasses will probably override me to do
something special”

   . . .

send: aPacket
    ”Precondition: self have a nextNode”

    ”Display debug information in the Transcript, then
    send a packet to my following node”

    Transcript show:
            self name printString,
            ’ sends a packet to ’,
            self nextNode name printString; cr.
. . .
```

## Creating the Class Packet

A packet is an object that represents a piece of information that is sent from node to node. So the responsibilities of this object are to allow us to define the originator of the sending, the address of the receiver and the contents.

```
Packet inherits from Object
Collaborators: Node
Responsibility:
addressee returns the addressee of the node to which
the packet is sent.
contents - describes the contents of the message sent.
originator - references the node that sent the packet.
```



```
accept: thePacket
    "Having received the packet, send it on . This is the default behavior."
    "My subclasses will probably override this method to so something special"

    self send: thePacket
```

Figure 8.1: Definition of accept: method

**Exercise 40** In the LAN, create a subclass of Object called Packet, with three instance variables: contents, addressee, and originator. Create accessors and mutators for each of them in the accessing protocol (in that particular case the accessors represents the public interface of the object). The addressee is represented as a symbol, the contents as a string and the originator has a reference to a node.

**Exercise 41** Define the method printOn: aStream that puts a textual representation of a packet on its argument aStream.

## Creating the Class Workstation

A workstation is the entry point for new packets onto the LAN network. It can originate packet to other workstations, printers or file servers. Since it is kind of network node, but provides additional behavior, we will make it a subclass of Node. Thus, it inherits the instance variables and methods defined in Node. Moreover, a workstation has to process packets that are addressed to it.

---

Workstation inherits from Node
Collaborators: Node, Workstation
and Packet
Responsibility: (the ones of node)
originate: aPacket - sends a packet.
accept: aPacket - perform an action on packets sent to the
workstation (printing in the transcript). For the other
packets just send them to the following nodes.

---

**Exercise 42** In the category LAN create a subclass of Node called Workstation without instance variables.

**Exercise 43** Define the method accept: aPacket so that if the workstation is the destination of the packet, the following message is written into the Transcript. Note that if the packets are not addressed to the workstation they are sent to the next node of the current one.

---

```
(Workstation new
    name: #Mac ;
    nextNode: (Printer new name: #PC1))
          accept: (Packet new addressee: #Mac)
```

A packet is accepted by the Workstation Mac

**Hints:** To implement the acceptance of a packet not addressed to the workstation, you could copy and paste the code of the Node class. However this is a bad practice, decreasing the reuse of code and the “Say it only once” rules. It is better to invoke the default code that is currently overriden by using super.

**Exercise 44** Write the body for the method originate: that is responsible for inserting packets in the network in the method protocol send-receive. In particular a packet should be marked with its originator and then sent.

```
originate: aPacket
 ”This is how packets get inserted into the network.
  This is a likely method to be rewritten to permit
  packets to be entered in various ways. Currently,
  I assume that someone else creates the packet and
  passes it to me as an argument.”
 . . .
```

## Creating the class LANPrinter

**Exercise 45** With nodes and workstations, we provide only limited functionality of a real LAN. Of course, we would like to do something with the packets that are travelling around the LAN. Therefore, you will now create a class LanPrinter, a special node that receives packets addressed to it and prints them (on the Transcript). Note that we use the name LanPrinter to avoid confusion with the existing class Printer in the namespace Smalltalk.Graphics (so you could use the name Printer in your namespace or the Smalltalk namespace if you really wanted to). Implement the class LanPrinter.

LanPrinter inherits from Node
Collaborators: Node and Packet
Responsibility:
accept: aPacket - if the packet is addressed to the printer, prints the packet contents else sends the packet to the following node.
print: aPacket - prints the contents of the packet (into the Transcript for example).

## Simulating the LAN

Implement the following two methods on the class side of the class Node, in a protocol called examples. But take care: the code presented below has **some bugs** that you should find and fix!.

```
simpleLan
  ”Create a simple lan”
  ”self simpleLan”

  — mac pc node1 node2 igPrinter —

  ”create the nodes, workstations, printers and fileserver”
  mac := Workstation new name: #mac.
  pc := Workstation new name: #pc.
  node1 := Node new name: #node1.
  node2 := Node new name: #node2.
  node3 := Node new name: #node3.
  igPrinter := Printer new name: #IGPrinter.

  ”connect the different nodes.”
  mac nextNode: node1.
  node1 nextNode: node2.
  node2 nextNode: igPrinter.
  igPrinter nextNode: node3.
  node3 nextNode: pc.
  pc nextNode: mac.

  ”create a packet and start simulation”
  packet := Packet new
          addressee: #IGPrinter;
          contents: ’This packet travelled around
to the printer IGPrinter.

  mac originate: packet.
```

---

```
anotherSimpleLan
   ”create the nodes, workstations and printers”

   |mac pc node1 node2 igPrinter node3 packet |
   mac:= Workstation new name: #mac.
   pc := Workstation new name:#pc.
   node1 := Node new name: #node1.
   node2 := Node new name: #node2.
   node3 := Node new name: #node3.
   igPrinter := LanPrinter new name: #IGPrinter.

   ”connect the different nodes.”
   mac nextNode: node1.
   node1 nextNode: node2.
   node2 nextNode:igPrinter.
   igPrinter nextNode: node3.
   node3 nextNode: pc.
   pc nextNode: mac.

   ”create a packet and start simulation’’
   packet := Packet new
           addressee: #anotherPrinter;
           contents: ’This packet travels around
           to the printer IGPrinter’.
   pc originate: packet.
```

As you will notice the system does not handle loops, so we will propose a solution to this problem in the future. To break the loop, use either **Ctrl-Y** or **Ctrl-C**, depending on your VisualWorks version.

## Creating the Class FileServer

Create the class FileServer, which is a special node that saves packets that are addressed to it (You should just display a message on the Transcript).

FileServer inherits from Node
Collaborators: Node and Packet
Responsibility:
accept: aPacket - if the packet is addressed to the file server save it (Transcript trace) else send the packet to the following node.
save: aPacket - save a packet.

# 9 Fundamentals on the Semantics of Self and Super

Main Author(s): Ducasse, Wuyts

This lesson wants you to give a better understanding of self and super.

## 9.1 self

When the following message is evaluated:

```
aWorkstation originate: aPacket
```

The system starts to look up the method originate: starts in the class of the message receiver: Workstation. Since this class defines a method originate:, the method lookup stops and this method is executed.

Following is the code for this method:

```
Workstation>>originate: aPacket

  aPacket originator: self.
  self send: aPacket
```

1. It first sends the message originator: to an instance of class Packet with as argument self which is a pseudo-variable that represents the receiver of originate: method. The same process occurs. The method originator: is looked up into the class Packet. As Packet defines a method named originator:, the method lookup stops and the method is executed. As shown below the body of this method is to assign the value of the first argument (aNode) to the instance variable originator. Assignment is one of the few constructs of Smalltalk. It is not realized by a message sent but handle by the compiler. So no more message sends are performed for this part of originator:.

```
Packet>>originator: aNode

    originator := aNode
```

2. In the second line of the method originate:, the message send: thePacket is sent to self. self represents the instance that receives the originate: message. **The semantics of self specifies that the method lookup should start in the class of the message receiver.** Here Workstation. Since there is no method send: defined on the class Workstation, the method lookup continues in the superclass of Workstation: Node. Node implements send:, so the method lookup stops and send: is invoked

```
Node>>send: thePacket

   self nextNode accept: thePacket
```

The same process occurs for the expressions contained into the body of the method send:.

## 9.2 super

Now we present the difference between the use of self and super. self and super are both pseudo-variables that are managed by the system (compiler). They both represents the receiver of the message being executed. However, there is no use to pass super as method argument, self is enough for this.

The main difference between self and super is their semantics regarding method lookup.

- The semantics of self is to start the method lookup **into the class of the message receiver and to continue in its superclasses.**

- The semantics of super is to start the method look into **the superclass of class in which the method being executed was defined and to continue in its superclasses.**. Take care the semantics is **NOT** to start the method lookup into the superclass of the receiver class, the system would loop with such a definition (see exercise 1 to be convinced). Using super to invoke a method allows one to invoke overridden method.

Let us illustrate with the following expression: the message accept: is sent to an instance of Workstation.

```
aWorkstation accept: (Packet new addressee: #Mac)
```

As explained before the method is looked up into the class of the receiver, here Workstation. The method being defined into this class, the method lookup stops and the method is executed.

```
Workstation>>accept: aPacket

   (aPacket addressee = self name)
      ifTrue: [ Transcript show: 'Packet accepted', self name asString ]
      ifFalse: [ super accept: aPacket ]
```

Imagine that the test evaluates to false. The following expression is then evaluated.

```
super accept: aPacket
```

The method accept: is looked up in the superclass of the class in which the containing method accept: is defined. Here the containing method is defined into Workstation so the lookup starts in the superclass of Workstation: Node. The following code is executed following the rule explained before.

```
Node>>accept: aPacket

   self hasNextNode
      ifTrue: [ self send: aPacket ]
```

**Remark.** The previous example does not show well the vicious point in the super semantics: the method look into **the superclass of class in whichor the method being executed was defined and not in the superclass of the receiver class.**

You have to do the following exercise to prove yourself that you understand well the nuance.

**Exercise 46** Imagine now that we define a subclass of Workstation called AnotherWorkstation and that this class does NOT defined a method accept:. Evaluate the following expression with both semantics:

```
anAnotherWorkstation accept: (Packet new addressee: #Mac)
```

You should be convinced that the semantics of super change the lookup of the method so that the lookup (for the method via super) does NOT start in the superclass of the receiver class but in the superclass of the class in which the method containing the super. With the wrong semantics the system should loop.

# 10 Object Responsibility and Better Encapsulation

## 10.1 Reducing the coupling between classes

To be a good citizen you as an object should follow as much as possible the following rules:

- Be private. Never let somebody else play with your data.
- Be lazy. Let do other objects your job.
- Be focused. Do only one main task.

While these guidelines are not really formal, one of the main consequences is that this is the responsibility of an object to provide a well defined interface protecting itself from its clients. The other consequence is that by delegating to other objects an object concentrates on a single task and responsibility. We now look how such guidelines can help us to provide better objects in our example.

### 10.1.1 Current situation

The interface of the packet class is really weak. It just provides free access to its data. The main impact of this weakness is the fact that the clients of the class Packet like Workstation relies on the internal coding of the Packet as shown in the first line of the following method.

```
Workstation>>accept: aPacket

  aPacket addressee = self name
    ifTrue: [  Transcript show: 'A packet is accepted by the Workstation ', self name asString ]
    ifFalse:  [ super accept: aPacket ]
```

As a consequence, if the structure of the class Packet would change, the code of its clients would have to change too. Generalizing such a bad practice would lead to system that are badly coupled and being really difficult to change to meet new requirements.

### 10.1.2 Solution.

This is the responsibility of a packet to say if the packet is addressed to a particular node or if it was sent by a particular node.

- Define a method named isAddressedTo: aNode in 'testing' protocol that answers if a given packet is addressed to the specified node.
- Define a method named isOriginatedFrom: aNode in 'testing' protocol that answers if a given packet is originated from the specified node.

Once these methods are defined, change the code of all the clients of the class Packet to call them.

## 10.2 A Question of Creation Responsibility

One of the problem with the previous approach for creating the nodes and the packets is the following: it is the responsibility of the client of the objects to create them well-formed. For example, it is possible to create a node without specifying a name! This is a disaster for our LAN system (create an example method 3, and try it out). The same problem occurs with the packet: it is possible to create a packet without address nor contents.

We will find a solution to these problems.

**Exercise 47** Define a class method named withName: in the class Node (protocol 'instance creation') that creates a new node and assign its name.

```
withName: aSymbol
....
```

Define a class method named withName:nextNode: in the class Node (protocol 'instance creation') that creates a new node and assign its name and the next node in the LAN

```
withName: aSymbol nextNode: aNode
....
```

Note that the first method can simply invoke the second one.

Define a class method named send:to: in the class Packet (protocol 'instance creation') that creates a new Packet with a contents and an address.

```
send: aString to: aSymbol
....
```

Now the problem is that we want to forbid the creation of non-well formed instances of these classes. To do so, we will simply redefine the creation method new so that it will raise an error.

**Exercise 48** Rewrite the new method of the class Node and Packet as the following:

```
new

   self error: 'you should invoke the method... to create a...'
```

However, you have just introduced a problem: the instance creation methods you just wrote in exercise 11 will not work anymore, because they call *new*, and that calling results in an error ! The solution is to rewrite them such as

```
Node class>>withName: aSymbol nextNode: aNode
     ˆ self basicNew initialize name: aSymbol ; nextNode: aNode
```

Do the same for the instance creation methods in class Packet.

**Exercise 49** Update and rerun your tests to make sure that your changes were correct.

Note that the previous code may break if a subclass specialize the nextNode: method does not return the instance. To protect ourslef from possible unexpected extension we add yourself that returns the receiver a the first cascaded message (here name:), here the newly created instance.

```
Node class>>withName: aSymbol nextNode: aNode
     ˆ self basicNew initialize name: aSymbol ; nextNode: aNode ; yourself
```

## 10.3 Reducing the coupling between classes

To be a good citizen you as an object should follow as much as possible the following rules:

- Be private. Never let somebody else play with your private data.
- Be lazy. Let do other objects your job.
- Be focused. Do only one main task.

While these guidelines are not really formal, one of the main consequences is that this is the responsibility of an object to provide a well defined interface protecting itself from its clients. The other consequence is that by delegating to other objects an object concentrates on a single task and responsibility. We now look how such guidelines can help us to provide better objects in our example.

### 10.3.1 Current situation

The interface of the Packet class is really weak. It just provides free access to its data. The main impact of this weakness is the fact that the clients of the class Packet like Workstation relies on the internal coding of the Packet as shown in the first line of the following method.

```
Workstation>>accept: aPacket

   aPacket addressee = self name
     ifTrue: [  Transcript show: ’A packet is accepted by the Workstation ’, self name asString ]
     ifFalse:  [ super accept: aPacket ]
```

As a consequence, if the structure of the class Packet would change, the code of its clients would have to change too. Generalizing such a bad practice would lead to system that are badly coupled and being really difficult to change to meet new requirements.

### 10.3.2 Solution.

This is the responsibility of a packet to say if the packet is addressed to a particular node or if it was sent by a particular node.

- Define a method named isAddressedTo: aNode in ‘testing’ protocol that answers if a given packet is addressed to the specified node.
- Define a method named isOriginatedFrom: aNode in ‘testing’ protocol that answers if a given packet is originated from the specified node.

Once these methods are defined, change the code of all the clients of the class Packet to call them. You should note that a better interface encapsulates better the private data and the way they are represented. This allows one to locate the change in case of evolution.

## 10.4 A Question of Creation Responsibility

One of the problems with the first approach for creating the nodes and the packets is the following: it is the responsibility of the client of the objects to create them well-formed. For example, it is possible to create a node without specifying a name! This is a disaster for our LAN system, the node would never reachable, and worse the system would breaks because the assumptions that the name of a node is specified would not hold anymore (insert an anonymous node in Lan and try it out). The same problem occurs with the packet: it is possible to create a packet without address nor contents.

The solution to these problems is to give the responsibility to the objects to create well-formed instances. Several variations are possible:

- When possible, providing default values for instance variable is a good way to provide well-defined instances.
- It is also a good solution to propose a consistent and well-defined creation interface. For example one can only provide an instance creation method that requires the mandatory value for the instance and forbid the creation of other instances.

**The class Packet.** We investigate the two solutions for the Packet class. For the first solution, the principle is that the creation method (new) should invoke an initialize method. Implement this solution. Just remember that new is sent to classes (a class method) and that initialize is sent to instances (instance method). Implement the method new in a 'instance creation' protocol and initialize in a 'initialize-release' protocol.

```
Packet class>>new

...

Packet>>initialize
  ...
```

The only default value that can have a default value is contents, choose

```
contents = 'no contents'
```

Ideally if each LAN would contain a default trash node, the default address and originator would point to it. We will implement this functionality in a future lesson. Implement first your own solution.

**Remarks and Analysis.** Note that with this solution it would be convenient to know if a packet contents is the default one or not. For this purpose you could provide the method hasDefaultContents that tests that. You can implement it in a clever way as shown below:

Instead of writing:

```
Packet>>hasDefaultContents

 ^ contents = 'no contents'

Packet>>initialize
 ...

contents := 'no contents'
...
```

You should apply the rule: 'Say only once' and define a new method that returns the default content and use it as shown below:

```
Packet>>defaultContents

   ^ 'no contents'

Packet>>initialize
   ...

   contents := self defaultContent
   ...

Packet>>hasDefaultContent
   ^contents = self defaultContents
```

With this solution, we limit the knowledge to the internal coding of the default contents value to only one method. This way changing it does not affect the clients nor the other part of the class.

## 10.5 Proposing a creational interface

**Packet.** We now apply the second approach by providing a better interface for creating packet. For this purpose we define a new creation method that requires a contents and an address.

Define a **class** methods named `send:to:` and `to:` in the class `Packet` (protocol 'instance creation') that creates a new `Packet` with a contents and an address.

```
Packet class>>send: aString to: aSymbol

....

Packet class>>to: aSymbol

....
```

**The class Node.** Now apply the same techniques to the class `Node`. Note that you already implemented a similar schema that the default value in the previous lessons. Indeed by default instance variable value is `nil` and you already implemented the method `hasNextNode` that to provide a good interface.
Define a **class** method named `withName:` in the class `Node` (protocol 'instance creation') that creates a new node and assign its name.

```
Node class>>withName: aSymbol

....
```

Define a **class** method named `withName:connectedTo:` in the class `Node` (protocol 'instance creation') that creates a new node and assign its name and the next node in the LAN.

```
Node class>>withName: aSymbol connectedTo: aNode

....
```

Note that if to avoid to duplicate information, the first method can simply invoke the second one.

## 10.6 Forbidding the Basic Instance Creation

One the last question that should be discussed is the following one: should we or not let a client create an instance without using the constrained interface? There is no general answer, it really depends on what we want to express. Sometimes it could be convenient to create an uncompleted instance for debugging or user interface interaction purpose.

Let us imagine that we want to ensure that no instance can be created without calling the methods we specified. We simply redefine the creation method new so that it will raise an error. Rewrite the `new` method of the class `Node` and `Packet` as the following:

```
Node class>>new

  self error: 'you should invoke the method... to create a...'
```

However, you have just introduced a problem: the instance creation methods you just wrote in the previous exercise will not work anymore, because they call `new`, and that calling results in an error! Propose a solution to this problem.

### 10.6.1 Remarks and Analysis.

A first solution could be the following code:

```
Node class>>withName: aSymbol connectedTo: aNode

    ˆ super new initialize name: aSymbol ; nextNode: aNode
```

However, even if the semantics permits such a call using super with a different method selector than the containing method one, it is a bad practice. In fact it implies an implicit dependency between two different methods in different classes, whereas the super normal use links two methods with the same name in two different classes. It is always a good practice to invoke the own methods of an object by using self. This conceptually avoids to link the class and its superclass and we can continue to consider the class as self contained.

The solution is to rewrite the method such as:

```
Node class>>withName: aSymbol connectedTo: aNode

ˆ  self basicNew initialize name: aSymbol ; nextNode: aNode
```

In Smalltalk there is a convention that all the methods starting with 'basic' should not be overridden. basicNew is the method responsible for always providing an newly created instance. You can for example browse all the methods starting with 'basic*' and limit yourself to Object and Behavior.

You can do the same for the instance creation methods in class Packet.

## 10.7 Protecting yourself from your children

The following code is a possible way to define an instance creation method for the class Node.

```
Node class>>withName: aSymbol

 ˆ self new name: aSymbol
```

We create a new instance by invoking new, we assign the name of the node and then we return it. One possible problem with such a code is that a subclass of the class Node may redefine the method name: (for example to have a persistent object) and return another value than the receiver (here the newly created instance). In such a case invoking the method withName: on such a class would not return the new instance. One way to solve this problem is the following:

```
Node class>>withName: aSymbol

 | newInstance |
 newInstance := self new.
 NewInstance name: aSymbol.
 ˆ newInstance
```

This is a good solution but it is a bit too much verbose. It introduces extra complexity by the the extra temporary variable definition and assignment. A good Smalltalk solution for this problem is illustrated by the following code and relies on the use of the yourself message.

```
Node class>>withName: aSymbol

  ˆ self new name: aSymbol ; yourself
```

yourself specifies that the receiver of the first message involved into the cascade (name: here and not new) is return. Guess what is the code of the yourself method is and check by looking in the library if your guess is right.

# 11 Hook and Template Methods

Main Author(s): Ducasse and Wuyts

In this chapter you will learn how to introduce hooks and template methods to favor extensibility. First we look at the current situation and introduce changes step by steps.

## 11.1 Providing Hook Methods

**Current situation.** The solution proposed for printing a Node displays the following string Node named: Node1 connected to: PC1 obtained by executing the following expression:

```
(Node withName: #Node1 connectedTo: (Node new name: #PC1)) printString
```

A straightforward way to implement the printOn: method on the class Node is the following code:

```
Node>>printOn: aStream

   aStream nextPutAll: 'Node named: ', self name asString.
   self hasNextNode
      ifTrue: [ aStream nextPutAll: ' connected to: ', self nextNode name ]
```

However, with such an implementation the printing of all kinds of nodes is the same.

**New Requirements.** To help in the understanding of the LAN we would like that depending on the specific class of node we obtain a specific printing like the following ones:

```
 (Workstation withName: #Mac connectedTo: (LanPrinter withName:
#PC1) printString

    Workstation Mac connected to Printer PC1

(LanPrinter withName: #Pr1 connectedTo: (Node withName: #N1)
printString

    Printer Pr1 connected to Node N1
```

Define the method *typeName* that returns a string representing the name of the type of node in the 'printing' protocol. This method should be defined in Node and all its subclasses.

```
(LanPrinter withName: #PC1) typeName

    'Printer'

(Node withName: #N1) typeName
    'Node'
```

Define the method simplePrintString on the class Node to provide more information about a node as show below:

```
(Workstation withName: #Mac connectedTo: (LanPrinter withName:
#PC1)) simplePrintString

    'Workstation Mac'

(LanPrinter withName: #PC1) simplePrintString

    'Printer PC1'
```

Then modify the printOn: method of the class Node to produce the following output:

```
(self withName: #Mac connectedTo: (LanPrinter new name:
#PC1))

'Node Mac connected to Printer PC1'
```

**Remark:** The method typeName is called a *hook* method. This reflects the fact that it allows the subclasses to specialize the behavior of the superclass, here the printing of a all the different kinds of nodes. The method simplePrintString, even if in our case is rather simple, is called a template method. This name reflects the fact that the method specifies the context in which hook methods will be called and how they will fit into the template method to produce the expected result.

Note that for abstract classes hook methods can be abstract too, one other case the hook method can propose a default behavior.

The Smalltalk class library contains a lot of such hooks that allows an easy customization of the proposed behavior. The proposed requirement already exists in the system.

**Exercise 50** Study the method printOn: on the class Object. Check its implementors and senders.

**Exercise 51** Study the method copy on the class Object. Check its implementors and senders. What do you think about the method postCopy check its senders and implementors.

# 12 SUnit Testing

Main Author(s): Stéphane Ducasse

For each class described in the Chapter on collection ”Set, Dictionary and Bag”, define unit tests for each of the described behavior.

## 12.1 Set

**Exercise 52** create a Unit test for each of the following methods

- aSet add: anObject and aSet addAll: aCollection,
- aSet includes: anObject
- aSet remove: un Objet and aSet remove: anObjet ifAbsent: aBlocException,
- aSet size and aSet capacity.

## 12.2 Dictionary

- aDictionary at: aKey put: anObject,
- aDictionary add: uneAssociation,
- aDictionary removeKey: aKey, aDictionary removeKey: aKey ifAbsent: aBloc,
- aDictionary includesKey: aKey,
- aDictionary includes: anObject,
- aDictionary occurencesOf: anObject,
- aDictionary keyAtValue: anObject,
- aDictionary do: aBloc,
- aDictionary keysDo: aBloc,
- aDictionary keysAndValuesDo: aBloc.

## 12.3 Bag

- aBag add: unObjet
- aBag remove: unObjet and aBag remove: unObjet ifAbsent: aBlock
- aBag addAll: aCollection and aBag removeAll: aCollection
- aBag add: anObject withOccurrences: n and aBag removeAllOccurrencesOf: anObject ifAbsent: aBlock

# Part IV

# Seaside

# 13 Web dynamique avec Seaside

Main Author(s): N. Bouraqadi, Université Libre de Bruxelles, bouraqadi@ensm-douai.fr

## 13.1 Compléments sur Seaside

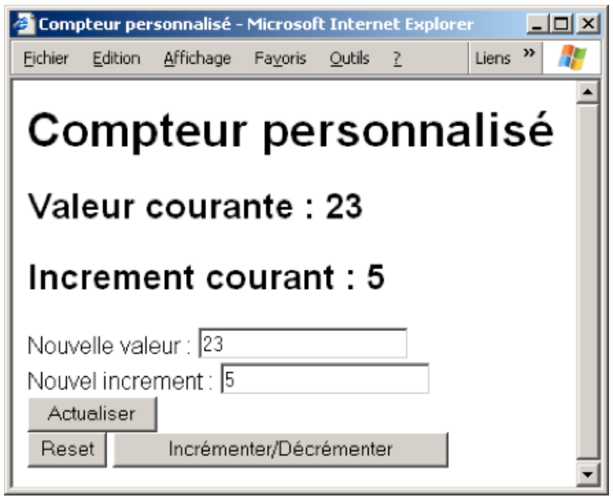
Quelques messages pour générer du html. Le destinataire de ces messages est l'objet passé en paramètre de la méthode renderOn: (instance de WAHtmlRender).

- text: 'chaine de caracteres' affiche simplement la chane de caractères.
- heading: 'texte du titre' level: niveau affiche un titre. Le deuxième paramètre est un entier qui correspond au niveau hiérarchique du titre (1 correspond au le plus grand)
- break introduit un retour à la ligne
- horizontalRule introduit une ligne horizontale
- form: [''definition de boutons, zones de saisies, ''] définit un formulaire au sens Html. Nécessaire pour avoir des boutons et autres zones de saisies dans une page Html. Reoit en paramètre un bloc qui contient les messages de création des boutons, zones de saisie,
- textInputWithValue: valeurInitiale callback: [ :valeur |''traitements''] crée une zone de saisie simple (sans barre de défilement). La valeur initiale est celle qui est affichée au démarrage (nil pour ne rien afficher). Le dernier argument est un bloc qui reçoit comme paramètre la valeur saisie (valeur) dans le champ. Cette valeur peut être utilisée dans le traitement défini par le bloc. Ce bloc est exécuté quand la touche ''Entrée'' est pressée ou quand on clic sur un bouton du formulaire dans lequel se trouve la zone de saisie.
- submitButtonWitAction: [''traitements''] text: 'titre du bouton' ajoute un bouton qui a pour titre la chane de caractères passée comme deuxième argument. Un clic sur le bouton provoque l'exécution des traitements définis dans le bloc passé comme premier paramètre.

## 13.2 Encore des compteurs !

Il s'agit de réaliser encore un compteur, mais cette fois, il devra être accessible via le web (utilisation de Seaside). De plus, il devra être personnalisable dans la mesure où l'utilisateur doit pouvoir modifier directement la valeur du compteur et modifier l'incrément. Concrètement, vous devez définir une classe CompteurPersonnalise sous-classe de WAComponent qui représente une application Seaside. CompteurPersonnalise sera munie de :

- deux champs (value et increment),
- une méthode d'initialisation (initialize),
- ainsi que la méthode de génération du code html (renderOn:).



Compteur personnalisé

Valeur courante : 23

Increment courant : 5

Nouvelle valeur : 23

Nouvel increment : 5

Figure 13.1: L'interface du compteur personnalisé

L'interface utilisateur doit être analogue à celle de la figure 13.1. Deux champs de saisie permettent de modifier la valeur du compteur et son incrément après clic sur le bouton "Actualiser". Le bouton "Reset" réinitialise le compteur (value mise à 0 et increment mis à 1). Enfin, le bouton "Incrémenter/Décrémenter" permet d'ajouter l'incrément au compteur et donc de l'incrémenter si l'incrément est positif ou de le décrémenter dans le cas contraire.

## 13.3 Séparer l'interface du code métier

La structure suggérée pour l'exercice précédent n'est pas très propre. En effet, un même objet prend en charge à la fois le traitement (code métier : incrémenter/décrémenter, modification de l'incrément, ) et l'interface utilisateur. Ce choix de conception rend difficile les éventuelles évolutions ou réutilisation. En particulier, si l'on souhaite changer d'interface utilisateur, voire de modèle de communication distante.

Dans cet exercice, on se propose de faire la séparation entre code métier et code d'interface et en illustrer l'utilité à l'aide d'un exemple simple. Cet exemple tourne autour d'une calculatrice arithmétique. Vous définirez tout d'abord la classe Calculatrice qui dispose de deux champs qui représentent respectivement l'opérande gauche et l'opérande droite. Munissez la classe d'accesseurs en lecture écriture à ces deux champs, ainsi que de 4 méthodes pour réaliser les 4 opérations arithmétiques. Bien entendu, ces quatre méthodes :

- ne prennent pas de paramètres,
- effectuent le calcul en utilisant les champs représentant les deux opérandes,
- et retournent le résultat du calcul

Définissez ensuite la classe CalculatriceWeb sous-classe de WAComponent qui représente une application Seaside. CalculatriceWeb permet l'utilisation à travers le web des opérations fournies par Calculatrice. Son interface s'apparente à celle donnée par la figure 13.2.

Vous allez maintenant exploiter la séparation entre code métier et code d'interface utilisateur. En effet, vous allez réutiliser la classe Calculatrice pour faire un nouveau compteur accessible via le web. L'interface devra être identique à celle du compteur de l'exercice précédent.

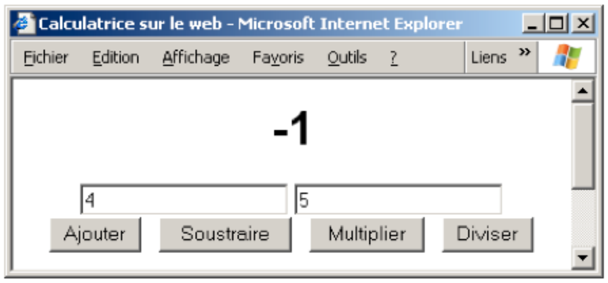


Figure 13.2: L'interface de la calculatrice.

## 13.4 Une application un peu plus sophistiquée

Il s'agit ici de définir un outil qui permet de gérer des tableaux blancs partagés via le web. Un tableau blanc est une zone de texte que plusieurs utilisateurs peuvent modifier. Chaque tableau est caractérisé par un nom et dispose d'une liste identifiants les utilisateurs qui ont le droit d'y accéder.

Chaque utilisateur dispose d'un identifiant et d'un mot de passe qu'il fournit pour se connecter. Une fois connecté il a le choix entre créer un nouveau tableau ou modifier tableau existant. Les utilisateurs qui ont accès à un tableau peuvent en modifier le contenu ainsi que la liste des utilisateurs qui ont accès au tableau.

# 14
# A Simple Application for Registering to a Conference

Main Author(s): A. Bergel, Universitaet Bern, bergel@iam.unibe.ch

The goal of this tutorial is to give you a feeling on creating a web application using Seaside. RegConf is a tool intended to help people to register to a conference.

## 14.1 RegConf: An Application for Registering to a Conference

Four steps are necessary to complete a registration:

1. A participant has to enter some personal data such as firstname, name, the institute where she is attached, and her email address.

2. Then some information about the hotel are required. For instance a room can be single or double in an hotel ranked between 1 and 4 stars. A price has then to be computed.

3. Finally informations regarding the payment are required. Once the credit card number, the issue date, and the type are entered,

4. A confirmation screen shows a summary of what was entered.

The flow of the application is described in the following figure.

The dashed rectangle designate the part of the application which is *isolated*. This means that once the flow of the running application leaves this box, there is no way to come back in it, specially using the back button.

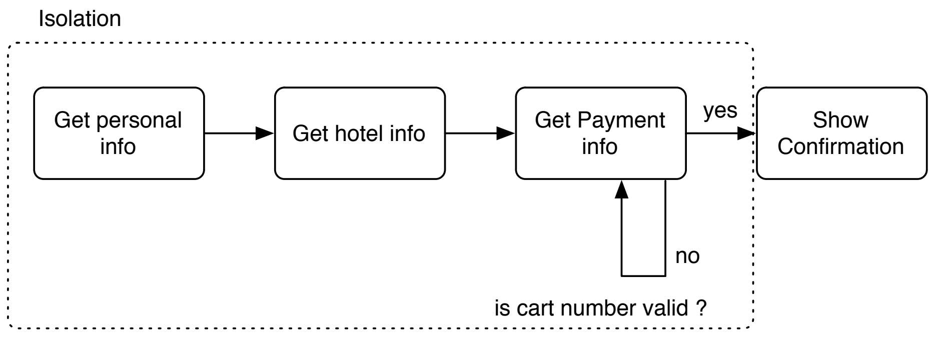


Figure 14.1:

## 14.2 Application Building Blocks

### 14.2.1 The Entry Point: RCMain

The control flow of the application has to be described in a task's go method. This method also represent the entry point of the application. Thus a name like RCMain sounds appropriated (RC stands for RegConf).

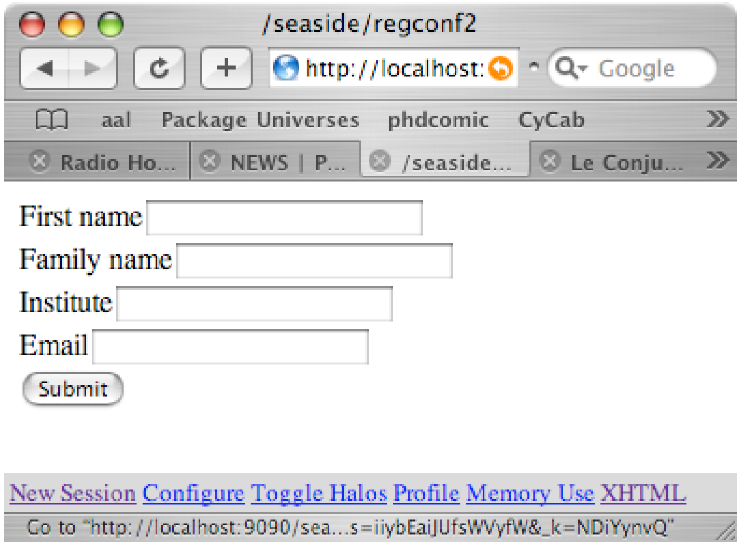
**Your job:** Create a task RCMain with a go method that describes the control flow of the application.

**Your job:** Start the web server on by executing WAKom startOn: 9090.

**Your job:** Create an initialize method on the class side to register your application in Seaside under the name regconf.

### 14.2.2 Getting User Information: RCGetUserInfo

All the control flow is defined in the class you previously defined. Getting user information is implemented as a normal seaside component (i.e., subclass of WAComponent). Instance variables of this class should reflect the structure of a user. Pressing the *submit* button returns to the caller component using answer:. Fetching the participant's informations can be done using text fields and submit button. Here is an example:



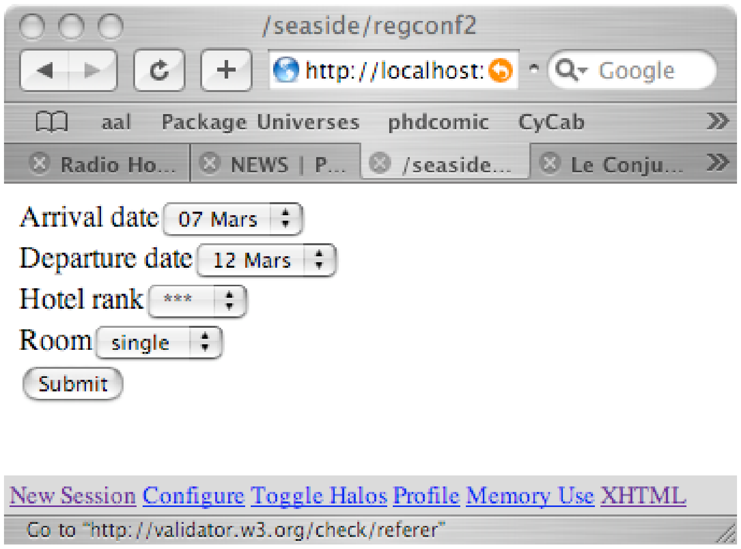
**Your job:** Write the method renderContentOn: in RCGetUserInfo.

**Your job:** Try your application using your favorite web browser. Make it point to http://localhost:9090/seaside/regconf.

The information passed around different states of the application can be contained in a dictionary. A more advanced design would require a class User for which an instance is passed around through.

### 14.2.3 Getting Hotel Information: RCGetHotelInfo

A list of choices is pleasant to fetch informations of the hotel.



Arrival date 07 Mars

Departure date 12 Mars

Hotel rank ***

Room single

Submit

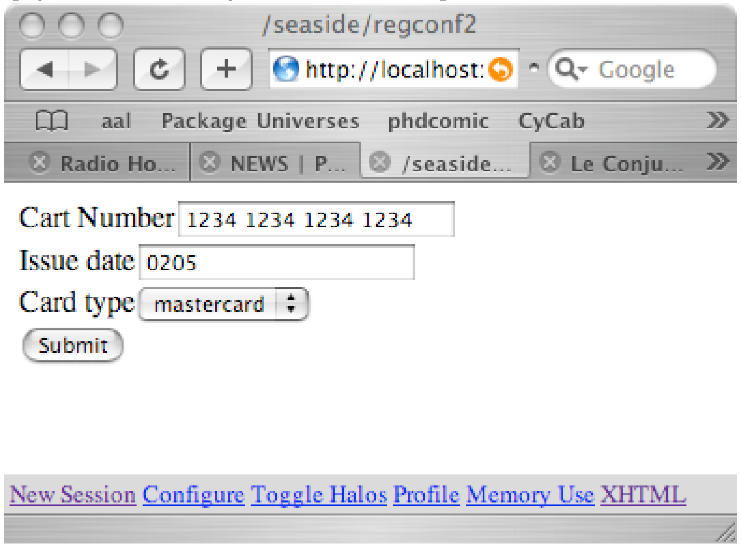
New Session Configure Toggle Halos Profile Memory Use XHTML

Go to "http://validator.w3.org/check/referer"

**Your job:** Write the class RCGetHotelInfo

### 14.2.4 Payment: RCPayment

The payment is valid only if 16 number was provided and if the issue date is not over.



Cart Number 1234 1234 1234 1234

Issue date 0205

Card type mastercard

Submit

New Session Configure Toggle Halos Profile Memory Use XHTML

**Your job:** Write the class RCPayment

### 14.2.5 Confimation: RCConfirmation

Once the payment is done, it is nice to show a summary of what was done.

**Your job:** Write the class RCConfirmation

## 14.3 Extensions

**Your job:** Study the class MiniCalendar of Seaside. Create a calendar starting from today. **Your job:** Use the mini calendar to add the possibility to say when and until which day the person wants to keep the room.