A (Not So) Short Introduction to S4

Object Oriented Programming in R

V0.5.1

Christophe Genolini

August 20, 2008

Contents

Part I
Preliminary

1 Introduction

This tutorial is a guide to object programming with R (or S4). It does not require knowing object oriented programming. However, a minimum of knowledge about R and programming in general is necessary. For those who are complete beginners, see section D page 66 for some tutorial or book.

1.1 Preamble: philosophy and computer science...

You are going to read a tutorial on object programming. You will thus learn a new method. Will know there does not exist "one" but "various" ways of programming: on this subject, data processing specialists do not always agree.

This tutorial following a certain vision. Some very competent people agree, some other do not... So this tutorial will present several points of view.

Thus, warned reader, you will have all the elements in hand, you will be able to judge by yourself, and choose your conception freely, in the light of knowledge... Elle n'est pas belle la vie?

1.2 What is S4?

S4 is the 4^{th} version of S. S is a language that has two implementation: S-plus is commercial, R is free. The main characteristic of S4 compared to S3 is the development of functions which allow to consider S as an object language [1]. By extension, S4 stand for object oriented programming with S. And thus with R or S-plus.

1.3 What is object programming?

An *object* is a set of variables and functions which all concern the same topic: the object itself. Is it unclear? Let us take an example: an object `image` will contain the variables which make it possible to define an image (as the size of the image, its mode of compression, the image itself) and the functions used to handle this image (like `blackAndWhite()` or `resizing()`).

1.4 Why would one use object oriented programming?

For the neophyte, object programming is something complicated and the advantages are not obvious: it is necessary to think program in advance, to model the problem, to choose its types, to think of the links which will connect the objects... Numerous disadvantages. Hence, the legitimate question: why shall once use oriented object programming?

[1] *allow to consider* and not *transforms into*. In any case, R is *not* an object oriented language, it remains a traditional language interpreted with a possible further cover. (Can't wait to discover R++...)

1.4.1 Traditional programming

Let's take an example and compare traditional programming with object programming. The BMI, Body Mass Index, is a measurement of thinness or obesity. Is is calculated by dividing the weight (in kilos) by the square size (in centimeters). Then, one concludes:

- $20 <$ BMI < 25: Everything is fine
- $25 <$ BMI < 30: Teddy bear
- $30 <$ BMI < 40: Comfortable Teddy bear
- $40 <$ BMI: Huge Teddy bear, with double softness effect, but who should go to see a doctor very quickly...
- $18 <$ BMI < 20: Barbie
- $16 <$ BMI < 18: Barbie model
- BMI < 16: Barbie skeleton, same diagnosis as the huge Teddy bear, be careful...

So we wish to calculate the BMI. In traditional programming, nothing simpler:

```
> ### Traditional programming, BMI
> weight <- 85
> size <- 1.84
> (BMI <- weight/size^2)
```

```
[1] 25.10633
```

Up to now, nothing very mysterious. If you want to calculate for two people `Me` and `Her`, you will have

```
> ### Traditional programming, my BMI
> weightMe <- 85
> sizeMe <- 1.84
> (BMIMe <- weightMe/sizeMe^2)
```

```
[1] 25.10633
```

```
> ### Traditional programming, her BMI
> weightHer <- 62
> sizeHer <- 1.60
> (BMIHer <- weightMe/sizeHer^2)
```

```
[1] 33.20312
```

It works... except that `Her` is described as "Comfortable Teddy bear" whereas her weight does not appear especially excessive. A small glance with the code reveals an error rather quickly: the calculation of `BMIHer` is false, we divided `weightMe` by `sizeHer` instead of `weightHer` by `sizeHer`. Naturally, R did not detect any error: from its point of view, it just carry out a division between two `numeric`.

1.4.2 Object programming

In object language, the method is different. It is necessary to begin by defining an object `BMI` which will contain two values, `weight` and `size`. Then, it is necessary to define the function `show` which will indicate the BMI[2]

```
> ### Definition of an object BMI
> setClass("BMI", representation(weight="numeric", size="numeric"))

[1] "BMI"

> setMethod("show", "BMI",
+     function(object){cat("BMI=",object@weight/(object@size^2)," \n ")}
+ )

[1] "show"
```

Then, the code equivalent to the one shown in section 1.4.1 page 3 is:

```
> ### Creation of an object for me, and posting of my BMI
> (myBMI <- new("BMI",weight=85,size=1.84))

BMI= 25.10633

> ### Creation of an object for her, and posting of her BMI
> (herBMI <- new("BMI",weight=62,size=1.60))

BMI= 24.21875
```

When initialization is correct (a problem which was also occurs in the traditional programming), no more error will be possible. Here lays the strength of object: the design of the program prevents from a certain number of bugs.

Type: the object also protects from type errors. A type errors consist in using a type where another type would be necessary. For example, adding a `character`to a `numeric`. This is more or less like adding apples and kilometers instead of adding apples and apples. Oriented object programmation prevent that:

```
> ### traditional programming, no type
> (weight <- "Hello")

[1] "Hello"

> new("BMI",weight="Hello",size=1.84)

Error in validObject(.Object) :
  invalid class "BMI" object: invalid object for slot
"weight" in class "BMI": got class "character", should be or extend class "numeric"
```

[2]To make our illustrative example immedialty reproducible, we need to define some objects here. We will do this without any explanation, but naturally, all this will then be explained with many details.

Validity checking: Object enables to use "coherence inspectors" to check if the object follow some rules. For example, one might want to prohibit negative sizes:

```
> ### Traditional programming, without control
> (SizeMe <- -1.84)
[1] -1.84
> ### Object programming, control
> setValidity("BMI",
+     function(object){if(object@size<0){return("negative Size")}else{return(TRUE)}}
+ )
Slots:

Name:   weight    size
Class: numeric numeric
> new("BMI",weight=85,size=-1.84)
Error in validObject(.Object) : invalid class "BMI" object: negative Size
```

Inheritance: object programming enables to define one object like *Inherit* from the properties of another object, thus becoming its *son*. The son object thus profits from all that exists for the father object. For example, one wishes to refine a little our diagnoses according to the sex of the person. We will thus define a new object, `BMIplus`, which will contain three values: `weight`, `size` and `sex`. The first two variables are the same ones as that of the object `BMI`. We will thus define the object `BMIplus` as heir of the object `BMI`. It will thus be able to profit from the function `show` such as we defined it for `BMI` and it will do so without any additional work, since `BMIplus` *inherits* `BMI`:

```
>  ### Definition of the heir
>  setClass("BMIplus",representation(sex="character"),contains="BMI")
[1] "BMIplus"
>  he <- new("BMIplus",size=1.76,weight=84,sex="Male")
>  ### Posting which uses what was defined for "BMI"
>  he
BMI= 27.11777
```

The power of this characteristic will appear more clearly in section 42.

Encapsulation: finally, object programming enables to define all the tools concerning an object and to lock them up in blocks, without having to look after them anymore. That is called *encapsulation*. Cars provide a good example of encapsulation: once the hood closed, one does not need to know anymore about the details of the mechanics to drive. Similarly, once the object finished and closed, a user does not have to worry about its working procedure. Even better, concerning the cars, it is not possible to be mistaken and to put gasoline in the radiator since the radiator is not accessible anymore. In the same way, encapsulation enables to protect what must be protected, and to give access only to what is not at risk.

1.5 Summary

Object programming "forces" the programmer to have a preliminary reflection. It is less possible to write "quick-and-dirty" code, a programming plan is almost essential. In particular:

- The objects must be declared and typed.
- Control mechanisms enable to check the internal coherence of the objects.
- An object can inherit the properties which were defined for another object.
- Lastly, the object allows an encapsulation of the program: once an object and the functions attached to it are defined, one does not need to deal with the object internal tool anymore.

1.6 The dark side of programming

To finish with this long introduction, a small digression: when human beings started to send rockets in space[3] and that they exploded during their flight, they wiped a little tear and sought the causes of the failure. As somebody had to be burned, they look for some human being. And they found... the computer science specialists. "It is not our fault, declared them, it is an established fact intrinsic to computers: all the programs have bugs! " Except that in this case, the bill of the bug was rather high...

So, very experimented and intelligent people started to think to find a way of programming that will prevent bugs apparition. They created new languages and defined programming rules. This is called *clean programming* or *good practices* (You will find a proposal of good practices in the appendix section B page 59). Some operations, some practices, some ways of programming are thus qualified as *good*, *nice* or *clean*; conversely, others are described as *bad*, *dangerous* or even *dirty*[4]. There is nothing like morality jugment is these words, it just qualifies this type of programming as being subject to coding errors. Thus bad practice must be avoided.

[Personnal point of view]: R is not a very clean language. When a programmer (or a researcher) want to add a new package, he is almost free to do whatever he wants. This is a great strength. But he is also free to do very "surprising" things[5] which is a weakness. So R enables the user to make numerous dangerous operations. It will sometimes be possible to make it cleaner by prohibiting oneself to handle some operations, but that will not always be the case, we will be brought to

[3] or when they give control of stock exchange, hospitals, pay slips... to the computers

[4] The exact term devoted by the computer science specialists in French is *crade*... Anythink like that in English ?

[5] Surprising things are things that the user would *not* expect. For example, knowing that `numeric()` denotes the empty `numeric` object, what would you except to denote an empty `matrix`? Anything else than `matrix()` will be a "surprise". R is full of surprise... (See section 4.6 to know how to denote an empty `matrix`).

use unsuitable tools. However, I will indicate the dangerous parts (in my opinion) by the small logo which decorates the beginning of this paragraph.

May you never fall into the dark side of programming...

2 Generality

2.1 Formal definition

An object is a coherent set of variables and functions which revolve around a central concept. Formally, an object is defined through three elements:

- The *class* is the name of the object. It is also its architecture (the list of variables and functions which compose it).
- The variables of the object are called *slots*.
- The functions of the object are called *methods*.

In the introduction, we had defined an object of class `BMI`. Its slots were `weight` and `size`, its method was `show`.

2.1.1 Slots

Slots are simply typed variables. A typed variable is a variable whose nature has been fixed. In R, one can write `weight <- 62` (at this point, `weight` is a `numeric`) then `weight <- "hello"` (`weight` became `character`). The type of the variable `weight` can change.

In object programming, it will not be possible to change the type of slot[6]. This appears to be a constraint, but in fact it is a safety. Indeed, if a variable was created to contain a weight, it does not have any reason to receive a string of character...

2.1.2 Methods

We distinguish 4 types of operations that can be apply on objects:

- **Creation methods**: this category include all the methods use for object creation. The most important one is called the constructor. But its use is a little rough for the user. So the (friendly) programmer writes easy access methods to create an object (for example `numeric` to create numeric variable or `read.csv` to create data.frame).
- **Validation**: in object programming, it is possible to check if slots respect some rules. It is also possible to create slot based on calculations made on the other attributes. All that is part of object validation.

[6]Unfortunately, with R and S4, it is possible. But since it is undoubtedly the most dirty operation of all the history of computer science, we prefer "to forget" it and to pretend it was not.

- **Slot handling**: modify and read slots is not as simple as in traditional programming. Thus, one dedicates methods to handle slot (for example `names` and `names<-`).
- **Others**: all that precedes is a kind of "legal minimum" for each object. Then remain the methods specific to the object, in particular posting methods and calculation methods.

Methods are typed objects. The type of a function consists of two parts: *inpout* and *output*. The input type is simply the set of types of arguments of the function. The output type is the type that the function turns over. The type of the function is `outputType <- inputType`. It is noted `output <- function(inputType)`.

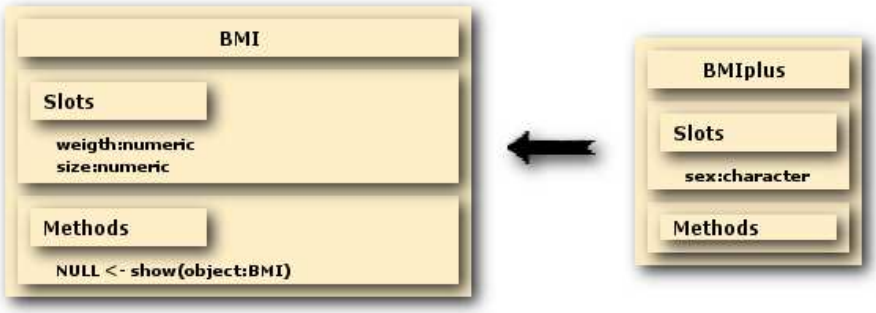
For example, the function `trace` calculates the trace of a matrix. Its type is thus `numeric <- trace(matrix)`.

2.1.3 Drawing is winning!

A good drawing is better than a long discourse. This maxim applies particularly well to object programming. Each class is represented by a rectangle in which are noted slots and methods with their type.

The inheritance (see section 9 page 42) between two classes is noted by an arrow, from the son towards the father.

Our objects BMI will look like follow:



3 Example

Rather than to invent an example with "wiz", "spoun" or complex mathematical concepts full of equations, here is a real case: doctor Tam works on anorectics. Week after week, she measures their BMI (example `17`, then `17.2`, then `17.3`). For a patient, the sequence obtained forms a `trajectory` (example for 3 weeks: `(17, 17.2, 17.3)`). Then, she classifies her patients in groups according to very precise criteria: `[has increased] (Yes)/(No)` ; or `[AskedToSeeTheirParents] (Yes)/(No)/(Refuses)` ; or

`[EatsRapidlyAgain] (Yes)/(No)`. In the end, her goal is to compare various ways of grouping trajectories.

In this tutorial, we will simplify this example and we will keep only what we need to explain S4 concepts. For those interested by the full treatment of this probleme, a package called `KmL` has been build to solve it. It is available on CRAN or on KmL website [**?**].

3.1 Analysis of the problem

This problem can be cut into three objects:

- The first contain the patients' trajectories.
- The second represent the repartition in groups (which we will call a *partition*).
- The third is a mixture of the two previous: trajectories partitioned in groups.

3.2 The object "Trajectories"

Tam warns us that for a given group, measurements are made either every week or every two weeks. The object `Trajectories` must take it into account. It will thus be defined by two slots:

- `times`: times at which measurements are carried out. To simplify, the week at which the follow-up starts is set to 1.
- `traj`: the patients' BMI trajectories

Example:

$$\texttt{times} = (1, 2, 4, 5)$$

$$\texttt{traj} = \begin{pmatrix} 15 & 15.1 & 15.2 & 15.2 \\ 16 & 15.9 & 16 & 16.4 \\ 15.2 & 15.2 & 15.3 & 15.3 \\ 15.7 & 15.6 & 15.8 & 16 \end{pmatrix}$$

What method(s) for this object? Tam says that in this kind of research, there are often missing data. It is important to know how much is missing. We thus define a first method which counts the number of missing values.

The second method lays between the objects `Trajectories` and `Partition` [7]: a traditional way used to building a partition is to distinguish the patients according to their initial BMI: in our example, one could consider two groups, "the low initial BMI" (trajectories I1 and I3) and "high initial BMI" (trajectories I2 and I4). So it is necessary to define a partition method according to the initial BMI and the number of groups wished.

[7] In a traditional object language, each method must belong to a specific class. In R, this has no importance, a a method does not need to be define relatively to a class. But to stick to the classical oriented object programming, we advice to attach each method to an object. So the method we are defining now is include in object `Trajectories`.

Lastly, counting the missing values is nice; imputing them [8], is better! The third method imputes the missing values.

3.3 The object "Partition"

A partition is a set of groups. For example, the groups could be A= {I1, I3 } and B= {I2, I4 }. It is undoubtedly simpler to consider them as a vector with length the patients' number: {A, B, A, B }. In some cases, it is also necessary to know the number of groups, in particular when a group is missing: if Tam classifies her teenagers in three groups and obtains the partition {A, C, C, A }, it is important to keep in mind that group B exists, even if it does not contain anybody in this case. Hence the two slots for partition:

- `nbGroups`: gives the number of groups.
- `part`: the sequence of groups to which the trajectories belong.

Example:

`nbGroups = 3`

`part =` $\begin{pmatrix} A \\ B \\ A \\ C \end{pmatrix}$

Some statisticians (and some software) transform the nominal variables into numbers. We could define `part` like a vector of integer. "It is much more practical", one regularly hears. Except that by doing so, one misleads R on the type of our variables: R is designed to know how to calculate the average of a numeric variable, but to refuse to calculate the average of a nominal variable (a `factor`). If we code `part` with numbers, R will agree to calculate the average of a `Partition`, which does not have any meaning. This would be more or less like calculating the average of some zip code... So a nominal variable must thus be coded by a `factor`, a numerical variable by a `numeric`. Yes, we know, in some cases it would be a little bit more *practical* to code `part` by numbers. But it is much more dangerous. Thus *it is bad*!

3.4 the object "TrajPartitioned"

Consdering a set of trajectories, several types of partitioning might possibly be interesting. So we need an object that groups some `Trajectories` and several `Partitions`. This is `TrajPartitioned`.

He is a heir of `Trajectories`. On the other hand, he is not be heir of `Partition` because both classes do not really share any properties. For more detail, see 9 page 42.

- `times`: times at which measurements are carried out (as in "trajectories")

[8] Imputing missing values is setting their values by "guessing" them according to some other knowledge... Obviously, this "guess" is more or less adequate, depending of the data and the technique uses!

- `traj`: patients' BMI trajectories (as in "trajectories")
- `partitionsList`: a list of partition.

Example:

`times` = $(1, 2, 4, 5)$

`traj` = $\begin{pmatrix} 15 & 15.1 & 15.2 & 15.2 \\ 16 & 15.9 & 16 & 16.4 \\ 15.2 & 15.2 & 15.3 & 15.3 \\ 15.5 & 15.6 & 15.8 & 16 \end{pmatrix}$

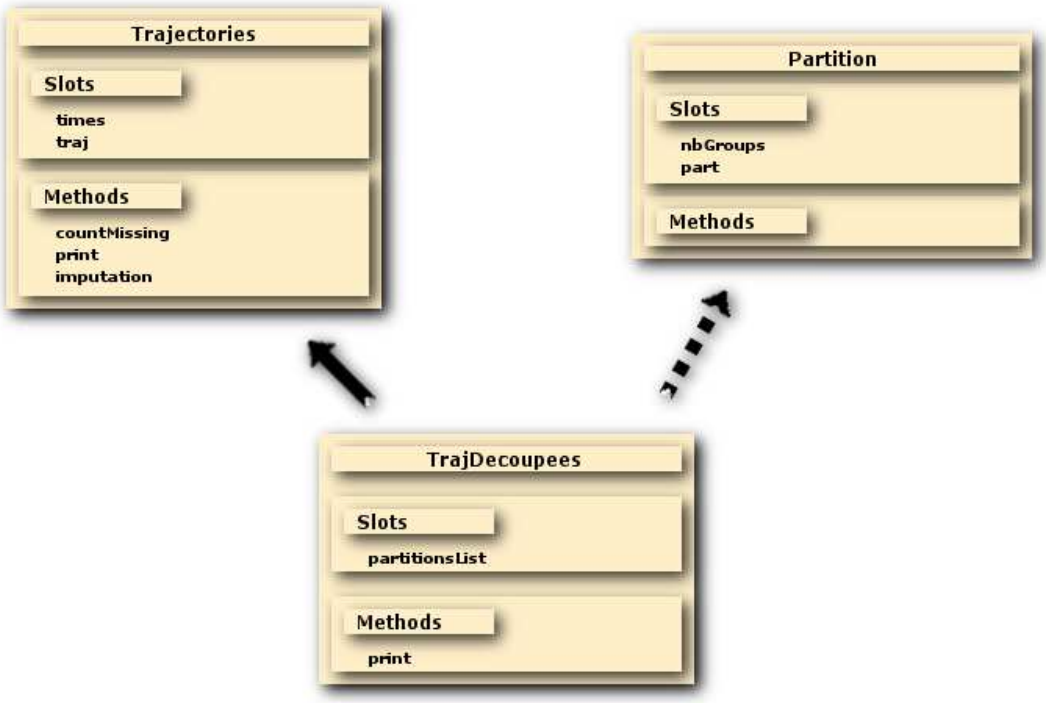
`partitionsList` =

`nbGroups` = 3, `part` = $\begin{pmatrix} A \\ B \\ A \\ C \end{pmatrix}$

`nbGroups` =2, `part` = $\begin{pmatrix} A \\ B \\ A \\ A \end{pmatrix}$

3.5 Drawing is winning !

Here is the schedule of our program:



3.6 Application to R

So, what about R, would you be in right to ask? It is an other characteristics of object languages, we made a relatively thorough analysis (thorough according to the simplicity

of our problem) and there is still nothing about R... Theoretically, one could even choose to code in another language.... But that is not the subject today.

Applied to R, methods defined in 2.1.2 page 7 become:

- **Creation methods**: the main creation method bears the name of the class.
- **Validation**: That depends on the object.
- **Attributes handling**: for each attribute, one will need a method giving access to its value and a method allowing to modify it.
- **Others**: the other methods depend of the object characteristics, except for posting methods. For posting, `show` allows a basic posting of the object when one types its name in the console. `print` gives a more complete posting. `plot` is the graphic posting.

To finish with R's characteristics, the majority of object languages force the programmer to group anything related to an object in the same place. That is called encapsulation. R does not have this property: you can declare an object, then another, then a method for the first object, then declare a third object, and so on. *This is bad!* and easily avoidable by following a simple rule: *an object, a file...* Everything concerning an object should be store in one file, a file shall contain method concerning only one object. So for another object, just open another file.

Part II
Bases of object programming

4 Classes declaration

In most object languages, the definition of the object contains the slot and the methods. In R, the definition contains only the slots. The methods are specified afterwards. It is a pity, it attenuates the power of encapsulation, but that's how it is. The informed user (i.e. you) can correct this "manually", by forcing himself to define slots and methods in the same file, and by using one file per object. It is cleaner...

More advice on the art of encapsulating section B.6 page 63.

4.1 Definition of slots

The first stage is to define the slots of the object itself. That is done by using the instruction `setClass`. `setClass` is a function which takes two arguments (and some other ingredients that we will see later).

- `Class` (with a capital first letter) is the name of the class which we are defining.
- `representation` is the list of attributes of the class.

As we saw in the introductory example, object programming carries out type control. That means that it does not allow a chain of characters to be stored where there should be a integer. So each field must be declared with its own type.

```
> setClass(
+     Class="Trajectories",
+     representation=representation(
+         times = "numeric",
+         traj = "matrix"
+     )
+ )
```

```
[1] "Trajectories"
```

Unfortunately, it is possible to not type (or to type badly) by using lists. It is bad but unfortunately, when using a list, we do not have any other choice.

4.2 Default Constructor

When a class exists, we can create an object of its class using the constructor `new`:

```
> new(Class="Trajectories")
```

```
An object of class "Trajectories"
Slot "times":
numeric(0)

Slot "traj":
<0 x 0 matrix>
```

As you can note, the result is not easy to read... It will be important to define a method to improve it. We'll deal with that in the section 5.2 page 20.

It is possible to define an object by specifying the values of its slots (all of them, or partly). We must each time specify it name of the concerning field:

```
> new(Class="Trajectories",times=c(1,3,4))
```

```
An object of class "Trajectories"
Slot "times":
[1] 1 3 4

Slot "traj":
<0 x 0 matrix>
```

```
> new(Class="Trajectories",times=c(1,3),traj=matrix(1:4,ncol=2))
```

```
An object of class "Trajectories"
Slot "times":
[1] 1 3

Slot "traj":
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

An object can be stored in a variable like any other value of R. To illustrate our statements, we are going to build up a small example. Three hospitals take part to the study. The Pitié Salpêtriere (which has not yet returned its data file, shame on them!), Cochin and Saint-Anne:

```
> trajPitie <- new(Class="Trajectories")
> trajCochin <- new(
+     Class= "Trajectories",
+     times=c(1,3,4,5),
+     traj=rbind (
+         c(15,15.1, 15.2, 15.2),
+         c(16,15.9, 16,16.4),
```

```
+         c(15.2, NA, 15.3, 15.3),
+         c(15.7, 15.6, 15.8, 16)
+     )
+ )
> trajStAnne <- new(
+     Class= "Trajectories",
+     times=c(1: 10, (6: 16) *2),
+     traj=rbind(
+         matrix (seq (16,19, length=21), ncol=21, nrow=50, byrow=TRUE),
+         matrix (seq (15.8, 18, length=21), ncol=21, nrow=30, byrow=TRUE)
+     )+rnorm (21*80,0,0.2)
+ )
```

4.3 To reach a slot

All this section is under the doble sign of danger...

The access to the slots is made with the operator `@`:

```
> trajCochin@times
[1] 1 3 4 5
> trajCochin@times <- c(1,2,4,5)
> trajCochin
An object of class "Trajectories"
Slot "times":
[1] 1 2 4 5

Slot "traj":
     [,1] [,2] [,3] [,4]
[1,] 15.0 15.1 15.2 15.2
[2,] 16.0 15.9 16.0 16.4
[3,] 15.2   NA 15.3 15.3
[4,] 15.7 15.6 15.8 16.0
```

As we will see thereafter, the use of the `@` should be avoided. Indeed, it does not call upon the methods of checking. The use that we present here (posting of a field, and even worse, assignment of a value to a field) should thus be proscribed in most cases.

In any case, the end-user should never have need to use it.

It is also possible to use the functions `attr` or `attributes`, but it is even worse: if one makes a simple typographic error, one modifies the structure of the object. And that is *very very very bad*!

4.4 Default values

One can declare an object by giving it default values. With each creation, if the user does not specify the values of slots, they will have one nevertheless. Therefore, one must add the `prototype` argument to the definition of the object:

```
> setClass(
+     Class = "TrajectoriesBis",
+     representation=representation(
+         time = "numeric",
+         traj = "matrix"
+     ),
+     prototype=prototype(
+         time = 1,
+         traj = matrix (0)
+     )
+ )
[1] "TrajectoriesBis"
```

Default initialization was something necessary in remote times, when if a variable was not initialized, one risked to write in the memory system (and thus to cause a blocking of the computer, the loss of our program and other even more terrible things). Today, with R and with most of the programming language, such a thing is no longer possible. If one does not initialize a field, R gives the object an adequate empty value.

From the philosophical point of view, when an object is created, either one knows its value, in which case it is affected to it, or one does not know it, in which case there is no reason to give it one value rather than another. The use of a default value thus seems to be rather a reminiscence of the past than an actual need. It should not be used anymore.

More over, [[[dans le feu de l'action]]], it can happen that one "forgets" to give an object its true value. If there exists a default value, it will be used. If there is no value by defect, that will probably cause an error... which, in this particular case, is preferable as it will catch our attention on the forgetness and will enable us to correct it. Therefore, values by defect other than the empty values should be avoided.

4.5 To remove an object

In the particular case of the object `trajectories`, there is no real default value which is essential. It is thus preferable to preserve class as it was initially defined. The class `TrajectoriesBis` does not have any utility. One can remove it using `removeClass`:

```
> removeClass("TrajectoriesBis")
[1] TRUE

> new(Class="TrajectoiresBis")
```

```
Error in getClass(Class, where = topenv(parent.frame())) :
  "TrajectoiresBis" is not a
defined class
```

Removing the definition of a class does not remove the methods which are associated to it. To really remove a class, it is necessary to remove the class *then* to remove all its methods...

In particular, imagine that you create a class and its methods. It does not work as you want so you decide to start it all over again and you erase the class. If you recreate it, all the old methods will still be valid.

4.6 The empty object

Some object functionalities call the function `new` without transmitting it any argument. For example, in the section heritage, the instruction `as(tdCochin,"Trajectories")` will be used (see section 9.6 page 49). This instruction calls `new("Trajectories")`. It is thus essential, at the time of the construction of an object, to always keep in mind that `new` must be usable without any argument. As the default values are not advised, it is necessary to envisage the construction of the empty object. That will be important in particular during the construction of the method `show`.

An empty object is an object having all the normal slot of an object, but they are empty, i.e. length zero. Conversely, an empty object has a class. Thus, an empty `numeric` is different from an empty `integer`.

```
> identical(numeric(),integer())
[1] FALSE
```

Here are some rules to be known concerning empty objects:

- `numeric()` and `numeric(0)` are empty `numeric`.
- `character()` and `character(0)` are empty `charater`.
- `integer()` and `integer(0)` are empty `integer`.
- `factor()` is empty `factor`. `factor(0)` indicates a `factor` of length 1 containing the element zero.
- `matrix()` is a `matrix` with one line and one column containing NA. In any case, it is not an empty matrix (its attribute `length` is 1). To define an empty matrix, it is necessary to use `matrix(nrow=0,ncol=0)`.
- `array()` is an `array` with one line and one column contain NA.
- `NULL` represents the null object. Thus, `NULL` is of class `NULL` whereas `numeric()` is of class `numeric`.

To test if an object is an empty object, its attribute `length` should be tested. In the same way, if we decide to define the method `length` for our object, it will be necessary to make sure that `length(myObject)==0` is true only if the object is empty (in order to ensure a coherence to R).

4.7 To see an object

You have just created your first class. Congratulation! To be able to check what you have just made, several instructions can be used "to see" the class and its structure. For those who want to brag a little (or more simply who like to be precise), the erudite term indicating what allows the program to see the contents or the structure of the objects is called introspection[9].

- `slotNames` gives the name of the slots as a vector of type `character`.
- `getSlots` gives the name of the slots and their type.
- `getClass` gives the names of slots and their type, but also heirs and ancestors. As the heritage is still "terra incognita" for the moment, it doesn't make any difference.

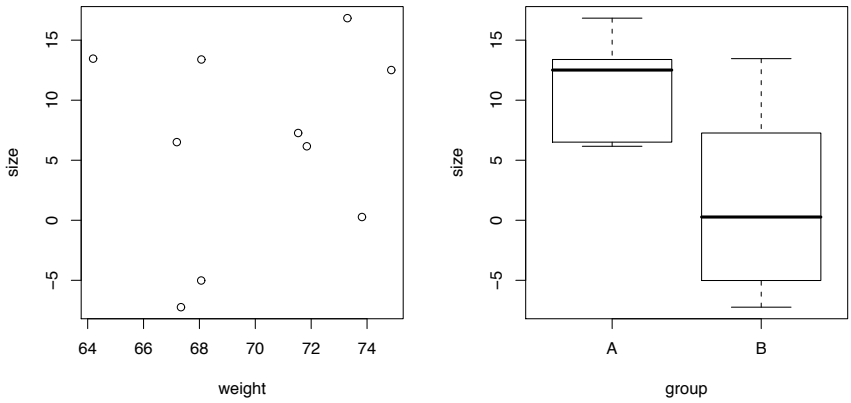
```
> slotNames("Trajectories")
[1] "times" "traj"
> getSlots ("Trajectories")
    times      traj
"numeric"  "matrix"
> getClass ("Trajectories")
Slots:

Name:    times    traj
Class: numeric  matrix
```

5 Methods

One of the interesting characteristics of object programming is to be able to define functions which will adapt their behavior to the object. An example that you already know, the function `plot` reacts differently according to the class of it arguments:

[9] It is a fact, I love R. However, my great love does not prevent me from critic its weakness... It will not be the case here. Concerning introspection, R is better than many other object languages.

```
> size <- rnorm(10,1.70,10)
> weight <- rnorm(10,70,5)
> group <- as.factor(rep(c("A","B"),5))
> par(mfrow=c(1,2))
> plot (size~weight)
> plot (size~group)
```



The first `plot` draws a cloud dots, the second draws boxplot. In the same way, it will be possible to define a specific behavior for our trajectories.

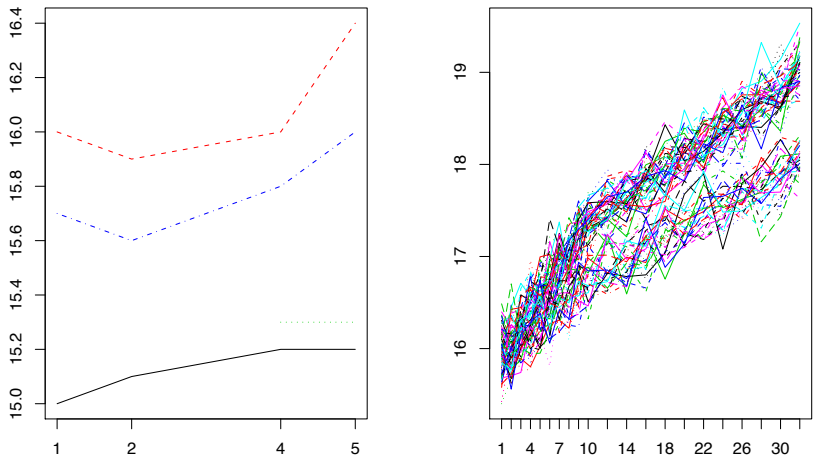
5.1 "setMethod"

For that, one uses the function `setMethod`. It takes three arguments:

1. `F` is the name of the function which we are redefining. In our case, `plot`
2. `signature` is the class of the object to which it applies. We will speak more above that section 8.2 page 38
3. `definition` is the function to be used. In our case, it will simply be a `matplot` that will take in account times of measurements.

```
> setMethod(
+     f= "plot",
+     signature= "Trajectories",
+     definition=function (X,y,...){
+         matplot(x@times,t(x@traj),xaxt="n",type="l",ylab= "",xlab="", pch=1)
+         axis(1,at=x@times)
+     }
+ )
> par(mfrow=c (1,2))
> plot(trajCochin)
> plot(trajStAnne)
```

```
[1] "plot"
```



Note: during the redefinition of a function, R imposes to use same the arguments as the function in question. To know the arguments of the `plot`, one can use `args`

```
>  args (plot)
function (x, y, ...)
NULL
```

We are thus obliged to use `function(x,y,...)` even if we already know that the argument `y` does not have any meaning. From the point of view of the cleanliness of programming, it is not very good: it would be preferable to be able to use the suitable names of the variables in our functions. Moreover, the default names are not really standardized, some functions use `object` while others use `.Object`...

5.2 "show" and "print"

In the same way, we define `show` and `print` for the trajectories. `args(print)` indicates that `print` takes for argument `(x,...)`. Thus:

```
> setMethod ("print","Trajectories",
+     function(x,...){
+         cat("*** Class Trajectories, method Print *** \n")
+         cat("* Times ="); print (x@times)
+         cat("* Traj = \n"); print (x@traj)
+         cat("******* End Print (trajectories) ******* \n")
+     }
+ )
```

```
[1] "print"
```

```
> print(trajCochin)
```

```
*** Class Trajectories, method Print ***
* Times =[1] 1 2 4 5
* Traj =
     [,1] [,2] [,3] [,4]
[1,] 15.0 15.1 15.2 15.2
[2,] 16.0 15.9 16.0 16.4
[3,] 15.2   NA 15.3 15.3
[4,] 15.7 15.6 15.8 16.0
******* End Print (trajectories) *******
```

For Cochin, the result is correct. For Saint-Anne, `print` will display too much information. So we need a second method.

`show` is the default method used to show an object when its name is write in the console. We thus define it by taking into account the size of the object: if there are too many trajectories, `show` post only part of them.

```
> setMethod("show","Trajectories",
+     function(object){
+         cat("*** Class Trajectories, method Show *** \n")
+         cat("* Times ="); print(object@times)
+         nrowShow <- min(10,nrow(object@traj))
+         ncolShow <- min(10,ncol(object@traj))
+         cat("* Traj (limited to a matrix 10x10) = \n")
+         print(formatC(object@traj[1:nrowShow,1:ncolShow]),quote=FALSE)
+         cat("******* End Show (trajectories) ******* \n")
+     }
+ )
```

```
[1] "show"
```

```
> trajStAnne
```

```
*** Class Trajectories, method Show ***
* Times = [1]  1  2  3  4  5  6  7  8  9 10 12 14 16 18 20 22 24 26 28 30 32
* Traj (limited to a matrix 10x10) =
      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9]  [,10]
 [1,] 15.91 15.97 16.58 16.52 16.69 16.49 17.15 16.86 17.21 17.35
 [2,] 16.2  15.65 16    16.78 16.64 16.89 17    17    16.96 17.34
 [3,] 16.07 15.97 16.37 16.07 16.6  17.02 17.03 16.94 17.32 17.34
 [4,] 15.89 16.47 16.44 16.75 16.93 16.99 16.94 17.17 16.97 17.34
 [5,] 15.97 16.21 16.14 16.09 16.62 16.78 16.97 17.04 17.2  17.47
 [6,] 16.15 15.95 15.96 16.31 16.43 17.16 16.84 16.81 17.09 17.45
 [7,] 16.04 16.07 16.32 16.08 16.5  16.74 17.21 17.13 17.51 17.13
 [8,] 16.2  16.15 16.28 16.14 16.25 16.96 16.81 16.92 17.27 16.95
 [9,] 15.62 16.17 16.37 16.41 16.15 17.01 17.01 16.95 17.54 17.33
[10,] 15.71 15.93 16.28 16.37 16.62 16.67 16.68 17.05 17.19 17.24
******* End Show (trajectories) *******
```

Better, isn't it?

A small problem must still be regulated. We saw in section 4.6 page 17 that `new` should be usable without any argument. However, it is no longer true:

```
> new("Trajectories")

*** Class Trajectories, method Show ***
* Times =numeric(0)
* Traj (limited to a matrix 10x10) =
Error in print(formatC(object@traj[1:nrowShow, 1:ncolShow]), quote = FALSE) :
  erreur
lors de l'évaluation de l'argument 'x' lors de la sélection d'une méthode pour la
fonction 'print'
```

Indeed, `new` creates an object, then display it using `show`. In the case of `new` without any argument, the empty object is send to `show`. However, `show` as we conceived it cannot treat the empty object.

More generally, all our methods must take into account the fact that they may have to deal with the empty object:

```
> setMethod("show","Trajectories",
+     function(object){
+         cat("*** Class Trajectories, method Show *** \n")
+         cat("* Times = "); print (object@times)
+         nrowShow <- min(10,nrow(object@traj))
+         ncolShow <- min(10,ncol(object@traj))
+         cat("* Traj (limited to a matrix 10x10) = \n")
+         if(length(object@traj)!=0){
+             print(formatC(object@traj[1:nrowShow,1:ncolShow]),quote=FALSE)
+         }else{}
+         cat("******* End Show (trajectories) ******* \n")
+     }
+ )

[1] "show"

> new("Trajectories")

*** Class Trajectories, method Show ***
* Times = numeric(0)
* Traj (limited to a matrix 10x10) =
******* End Show (trajectories) *******
```

It works!

We thus have two posting methods. By default, `show` display the object or part of the object if it is too large; `print` enables to see the entire object.

5.3 "setGeneric"

Up to now, we did nothing but define methods which already existed (`print` existed for the `numeric`, for the `character`...) for the object `Trajectories`. We now need to define a new method that is specific to `Trajectories`. Therefore, it is necessary for us to declare it. This can be done by using the function `setGeneric`. This function requires two arguments:

- `name` is the name of the method which we define.
- `def` is an example of function which is used to define it.

It is not necessary to type the function. More precisely, it is not possible to type it, it must be generic i.e. usable for several different classes.

```
> setGeneric (
+     name= "countMissing",
+     def=function(object){standardGeneric("countMissing")}
+ )
[1] "countMissing"
```

This add `countMissing` to the list of the methods that R knows. We can now define more specifically `countMissing` for the object `trajectories`:

```
> setMethod(
+     f= "countMissing",
+     signature= "Trajectories",
+     definition=function(object){
+         return(sum(is.na(object@traj)))
+     }
+ )
[1] "countMissing"
> countMissing(trajCochin)
[1] 1
```

There is no control over the existence of a `setGeneric`: if a `setGeneric` existed, the new definition destroyed the old one - in the same way as when you assign a value to a variable it destroys the preceding one -. A redefinition is often a mistake, the programmer was unaware that the function already existed... To protect oneself from this problem, it is possible to "lock" the definition of a method by using `lockBinding`:

```
> lockBinding("countMissing",.GlobalEnv)
NULL
```

```
> setGeneric(
+     name="countMissing",
+     def=function(object,value){standardGeneric("countMissing")}
+ )

Error in assign(name, fdef, where) :
  impossible de changer la valeur d'un lien
verrouillé pour 'countMissing'

NULL
```

It is not possible anymore to erase "by mistake" the `setGeneric`.
Another solution is to define a function

```
> setGenericVerif <- function(x,y){if(!isGeneric(x)){setGeneric(x,y)}else{}}
```

and then to never use `setGeneric` but use `setGenericVerif` instead.

5.4 To see the methods

Our class becomes more complex. It is time to take a little break and to admire our work. `showMethods` is the method of the situation. There are several ways of using it. One among those allows to see all the methods which we have defined for one class of data:

```
> showMethods(class="Trajectories")
Function: initialize (package methods)
.Object="Trajectories"
    (inherited from: .Object="ANY")

Function: plot (package graphics)
x="Trajectories"

Function: print (package base)
x="Trajectories"

Function: show (package methods)
object="Trajectories"
```

Now that we listed what exists, we can look at a particular method: `getMethod` enables to see the definition (the contents of the body function) of a method for a given object. If the method in question does not exist, `getMethod` indicates an error:

```
> getMethod(f="plot",signature="Trajectories")
Method Definition:

function (x, y, ...)
{
    matplot(x@times, t(x@traj), xaxt = "n", type = "l", ylab = "",
```

```
        xlab = "", pch = 1)
    axis(1, at = x@times)
}

Signatures:
        x
target  "Trajectories"
defined "Trajectories"

> getMethod(f="plot",signature="Trjectoires")

Error in getMethod(f = "plot", signature = "Trjectoires") :
  No method found for
function "plot" and signature Trjectoires
```

More simply, `existsMethod` indicates if a method is or is not defined for a class:

```
> existsMethod(f="plot",signature="Trajectories")
[1] TRUE
> existsMethod(f="plot",signature="Partition")
[1] FALSE
```

This is not really useful for the user, that is more for programmers who can write things such as: "IF(such a method exists for such an object) THEN(adopt behavior 1) ELSE(adopt behavior 2)".

6 Construction

Constructors are some tools which enable to build a correct object, that is methods of creation themselves (methods which store the values in slots) and checking methods (methods which check that the values of the slots are conform to what the programmer wishes).

6.1 Inspector

The inspector is there to control that there is no internal inconsistency in the object. One gives it rules, then, at each object creation, it will check that the object follows the rules.

Therefore, the rules have to be include in the definition of the object itself via the argument `validity`. For example, in the object `Trajectories`, one can want to check that the number of groups present in the `cluster` is lower or equal to the number of groups declared in `nbCluster`.

```
> setClass(
+     Class="Trajectories",
+     representation(times="numeric",traj="matrix"),
+     validity=function(object){
+         cat("~~~ Trajectories: inspector ~~~ \n")
+         if(length(object@times)!=ncol(object@traj)){
+             stop ("[Trajectories: validation] the number of temporal measurements does not corr
+         }else{}
+         return(TRUE)
+     }
+ )
[1] "Trajectories"
> new(Class="Trajectories",times=1:2,traj=matrix(1:2,ncol=2))
~~~ Trajectories: inspector ~~~
*** Class Trajectories, method Show ***
* Times = [1] 1 2
* Traj (limited to a matrix 10x10) =
[1] 1 2
******* End Show (trajectories) *******
> new(Class="Trajectories",times=1:3,traj=matrix(1:2,ncol=2))
~~~ Trajectories: inspector ~~~
Error in validityMethod(object) :
  [Trajectories: validation] the number of temporal
measurements does not correspond to the number of columns of the matrix
```

Note that function `validity` such we just define it does not take any precaution concerning the empty object. But that is not so important since `new()` does not call the inspector.

It is also possible to define the class, and then later to define its validity using `setValidity`. In the same way, it is possible to define the `representation` and the `prototype` externally. But this way of proceeding is conceptually less clean. Indeed, the design of an object must be think and not made up of an addition of different things...

The inspector is called ONLY during the initial creation of the object. If it is then modified, no control is done. We will be able to correct that by using the "setters". For the moment, it is interesting to note the importance of proscribing the use of `@`: direct modification of a field is not subject to checking:

```
> trajStLouis <- new(Class="Trajectories",times=c(1),traj=matrix(1))
~~~ Trajectories: inspector ~~~
> ### No checking, the number of temporal measurements will no longer
> ###    correspond to the trajectories
> (trajStLouis@times <- c(1,2,3))
[1] 1 2 3
```

6.2 The initializator

The inspector is a simplified version of a more general tool called the initializator. The initializator is a method that build an object and set all the slots to their value. It is called at each object construction, i.e. with each use of the function `new`.

Let us take again our trajectories. It would be rather pleasant that the columns of the matrix of the trajectories have names, the names of measurements times. In the same way, the lines could be subscripted by a number of individual:

```
     T0    T1    T4    T5
I1   15   15.1  15.2  15.2
I2   16   15.9   16   16.4
I3  15.2   NA   15.3  15.3
I4  15.5  15.6  15.8   16
```

The initializator will allow us to do all that. The initializator is a method which, when one calls for `new`, constructs the object by setting all the slot to the value we want to give them.

The name of the method is `initialize`. `initialize` uses a function (defined by the user) which has for argument the object that is being built and the various values that have to be assigned to the slots of the object.

This function works on a local version of the object. It must end by an assignment of the slots and then by a `return(object)`.

```
> setMethod(
+     f="initialize",
+     signature="Trajectories",
+     definition=function(.Object,times,traj){
+         cat("~~~ Trajectories: initializator ~~~ \n")
+         rownames(traj) <- paste("I",1:nrow(traj),sep= "")
+         .Object@traj <- traj      # Assignment of the slots
+         .Object@times <- times
+         return(.Object)           # return of the object
+     }
+ )

[1] "initialize"

> new(Class="Trajectories",times=c(1,2,4,8),traj=matrix(1:8,nrow=2))

~~~ Trajectories: initializator ~~~
*** Class Trajectories, method Show ***
* Times = [1] 1 2 4 8
* Traj (limited to a matrix 10x10) =
   [,1] [,2] [,3] [,4]
I1 1    3    5    7
I2 2    4    6    8
******* End Show (trajectories) *******
```

The definition of an initializator disable the inspector. In our case, `times` can again contain more or less values than columns in `traj`.

```
> new(Class="Trajectories",times=c(1,2,48),traj=matrix(1:8,nrow=2))
```

```
~~~ Trajectories: initializator ~~~
*** Class Trajectories, method Show ***
* Times = [1]  1  2 48
* Traj (limited to a matrix 10x10) =
   [,1] [,2] [,3] [,4]
I1 1    3    5    7
I2 2    4    6    8
******* End Show (trajectories) *******
```

To use an initializator and an inspector in the same object, it is thus necessary to call "manually" the inspector by using the instruction `validObject`. Our initializator *including* the inspector becomes:

```
> setMethod (
+     f="initialize",
+     signature="Trajectories",
+     definition=function(.Object,times,traj){
+         cat ("~~~~~ Trajectories: initializator ~~~~~ \n")
+         if(!missing(traj)){
+             colnames(traj) <- paste("T",times,sep="")
+             rownames(traj) <- paste("I",1:nrow(traj),sep="")
+             .Object@times <- times
+             .Object@traj <- traj
+             validObject(.Object)    # call of the inspector
+         }
+         return(.Object)
+     }
+ )
```

```
[1] "initialize"
```

```
> new(Class="Trajectories",times=c(1,2,4,8),traj=matrix(1:8,nrow=2))
```

```
~~~~~ Trajectories: initializator ~~~~~
~~~ Trajectories: inspector ~~~
*** Class Trajectories, method Show ***
* Times = [1] 1 2 4 8
* Traj (limited to a matrix 10x10) =
   T1 T2 T4 T8
I1 1  3  5  7
I2 2  4  6  8
******* End Show (trajectories) *******
```

```
> new(Class="Trajectories",times=c(1,2,48),traj=matrix(1:8,nrow=2))
```

```
~~~~~ Trajectories: initializator ~~~~~
Error in dimnames(x) <- dn :
  la longueur de 'dimnames' [2] n'est pas égale à l'étendue
du tableau
```

Note the condition related to `missing(traj)` to take into account the empty object... This new definition removed the old one.

A constructor does not necessarily take the slots of the object as argument. For example, if we know (that is not the case in reality, but let us imagine so) that the BMI increases by 0.1 every week, we could build trajectories by providing the number of weeks and the initial weights:

```
> setClass (
+     Class="TrajectoriesBis",
+     representation(
+         times = "numeric",
+         traj = "matrix"
+     )
+ )
```

```
[1] "TrajectoriesBis"
```

```
> setMethod ("initialize",
+     "TrajectoriesBis",
+     function(.Object,nbWeek,BMIinit){
+         traj <- outer(BMIinit,1:nbWeek,function(init,week){return(init+0.1*week)})
+         colnames(traj) <- paste("T",1:nbWeek,sep="")
+         rownames(traj) <- paste("I",1:nrow(traj),sep="")
+         .Object@times <- 1:nbWeek
+         .Object@traj <- traj
+         return(.Object)
+     }
+ )
```

```
[1] "initialize"
```

```
> new(Class="TrajectoriesBis",nbWeek=4,BMIinit=c(16,17,15.6))
```

```
An object of class "TrajectoriesBis"
Slot "times":
[1] 1 2 3 4

Slot "traj":
     T1   T2   T3   T4
I1 16.1 16.2 16.3 16.4
I2 17.1 17.2 17.3 17.4
I3 15.7 15.8 15.9 16.0
```

There can be only one initializator by class. It is thus necessary for it to be as global as possible. The definition above would prohibit the construction of a trajectory based on a matrix. It is thus not advised to be too specific. Lastly, it is better to leave this kind of transformation general public constructors.

6.3 Constructor for user

As we said in the introduction, the (nice) programmer, being aware that `new` is not a *friendly* function, adds "user friendly" constructors. This can be carried out by using a function (a traditional function, not a S4 method) generally bearing the name of the class. In our case, it will be the function `trajectories`. For those who want to saving their fingers and not have too much to write on their keyboard, it is possible to define two constructors, the real one and its shortened form:

```
> tr <- trajectories <- function(times,traj){
+     cat ("~~~~~ Trajectories: constructor ~~~~~ \n")
+     new (Class="Trajectories",times=times,traj=traj)
+ }
> trajectories(time=c(1,2,4),traj=matrix(1:6,ncol=3))
~~~~~ Trajectories: constructor ~~~~~
~~~~~ Trajectories: initializator ~~~~~
~~~ Trajectories: inspector ~~~
*** Class Trajectories, method Show ***
* Times = [1] 1 2 4
* Traj (limited to a matrix 10x10) =
   T1 T2 T4
I1 1  3  5
I2 2  4  6
******* End Show (trajectories) *******
```

The interesting point is to be able to carry out some more sophisticated treatment. For example, in a great number of cases, Tam measures the trajectories every week and stores these measurements in a matrix. She thus wishes to have the choice: either to define the object `trajectories` simply by giving a matrix, or by giving a matrix and times:

```
> trajectories <- function(times,traj){
+     if(missing(times)){times <- 1:ncol(traj)}
+     new(Class="Trajectories",times=times,traj=traj)
+ }
> trajectories(traj=matrix(1:8,ncol=4))
~~~~~ Trajectories: initializator ~~~~~
~~~ Trajectories: inspector ~~~
*** Class Trajectories, method Show ***
* Times = [1] 1 2 3 4
* Traj (limited to a matrix 10x10) =
   T1 T2 T3 T4
```

```
I1 1  3  5  7
I2 2  4  6  8
******* End Show (trajectories) *******
```

R sometimes accepts that two different entities have the same name. So it is possible to define a function bearing the same name as a class. One of the disadvantages is that one does not know anymore about what one speaks. Thus, it is possible to give a name with a Capital letter to the class and the same name without the Capital letter to the constructor function.

Contrary to the initializator, one can define several constructors. Always under the assumption that the BMI increases by 0.1 every week, one can define `regularTrajectories`:

```
> regularTrajectories <- function(nbWeek,BMIinit) {
+     traj <- outer(BMIinit,1:nbWeek,function(init,week){return(init+0.1*week)})
+     times <- 1: nbWeek
+     return(new(Class="Trajectories",times=times,traj=traj))
+ }
> regularTrajectories(nbWeek=3,BMIinit=c(14,15,16))
~~~~~ Trajectories: initializator ~~~~~
~~~ Trajectories: inspector ~~~
*** Class Trajectories, method Show ***
* Times = [1] 1 2 3
* Traj (limited to a matrix 10x10) =
   T1   T2   T3
I1 14.1 14.2 14.3
I2 15.1 15.2 15.3
I3 16.1 16.2 16.3
******* End Show (trajectories) *******
```

Note that the two constructors both call upon the initializator. Hence the importance of defining a *global* initializator that ca deal with all the cases.

6.4 Summary

During the construction of an object, there are three places where it is possible to carry out operations: in the construction's function, in the initializator and in the inspector. The inspector can do nothing but check, it does not allow to modify the object. However, it can be called for an object which has been built. In order not to mix everything, it is thus preferable to specialize each one of these operators and to reserve each one precise tasks. Here is a possibility:

- **the construction function** is to be called by the user. It is the most general, can take various arguments, possibly arguments which are not object attributes. It then transforms its arguments in attributes. Personally, I entrust it the transformation

of its arguments into future slots (like `regularTrajectories` prepared arguments for a call of `new("Trajectories")`).

The function of construction always ends by `new`.

- **the initializator** is called by `new`. It is In charge of giving to each slot its value, after possible modifications. I give it the tasks which must be carried out for all the objects, whatever the constructors that call it (like the renaming of the lines and the columns of `Trajectories`).

  If the initializator was not defined, R calls the default initializator which assigns the values to the slot, and then calls the inspector.

  Once the object created, the initializator calls the inspector (the default initializator calls the inspector, the initializator defined by the user must end by an explicit call).

- **the inspector** controls the internal coherence of the object. It can for example prohibit some values with some slots, to check that the size of slot is in conformity with what is awaited,... It cannot modify the values of the slot, it must just check that they follow the rules.

Another possibility, suggested by some (high level) programmers, is to not use the initializator at all and let the default initializator (more efficient) to be called. The construction function will do all the preparation, then will call `new("Trajectories")`. The inspector will check the internal coherence. This way seems a very efficient option.

For the neophyte, to know which, when and where method is called is very complicated. `new` uses the initializator if it exists, the inspector if not (but not both except explicit call as we did). Other instructions, like `as` (section 9.6 page 49) call initializators and inspectors. When in doubt, in order to understanding well who is called and when, one can added a small line at the head of the function which posts the name of the method used. "Very ugly" commented some lectors. Alas, they are right! But here, pedagogy precedes aesthetics... Moreover, when we will come down to heritage, things will become a little more complicated and this "very ugly" will be a little more necessary...

7 Accessor

We already spoke about it, use `@` apart from a method, is bad... However, it is necessary to be able to recover the values of slots. That is the role of *accessors*.

7.1 get

`get` is a method which returns the value of a slot. In traditional programming, a great number of functions take a variable and return a part of its arguments. For example, `names` applied to a `data.frame` returns the names of the columns. For our trajectories, we can define several getters: of course it is necessary to have one which returns `times`

and one which returns `traj`. Our getters being new methods for R, they should be declared by using `setGeneric`. Then we simply define them by using a `setMethod`:

```
> ### Getter for "times"
> setGeneric("getTimes",function(object){standardGeneric ("getTimes")})
[1] "getTimes"
> setMethod("getTimes","Trajectories",
+     function(object){
+         return(object@times)
+     }
+ )
[1] "getTimes"
> getTimes(trajCochin)
[1] 1 2 4 5
> ### Getter for "traj"
> setGeneric("getTraj",function(object){standardGeneric("getTraj")})
[1] "getTraj"
> setMethod("getTraj","Trajectories",
+     function(object){
+         return(object@traj)
+     }
+ )
[1] "getTraj"
> getTraj(trajCochin)
     [,1] [,2] [,3] [,4]
[1,] 15.0 15.1 15.2 15.2
[2,] 16.0 15.9 16.0 16.4
[3,] 15.2   NA 15.3 15.3
[4,] 15.7 15.6 15.8 16.0
```

But it is also possible to create more sophisticated getters. For example one can regularly need the BMI at inclusion:

```
> ### Getter for the inclusion BMI (first column of "traj")
> setGeneric("getTrajInclusion",function(object){standardGeneric("getTrajInclusion")})
[1] "getTrajInclusion"
> setMethod ("getTrajInclusion","Trajectories",
+     function(object){
+         return(object@traj[,1])
+     }
+ )
[1] "getTrajInclusion"
> getTrajInclusion(trajCochin)
[1] 15.0 16.0 15.2 15.7
```

7.2 set

A setter is a method which assigns a value to a slot. With R, the assignment is made by `<-`. Without entering the meanders of the program, the operator `<-` calls a specific method. For example, when one uses `names(data) <- c("A","B")`, R calls the function `"names<-"`, this function duplicates the object `data`, modifies the attribute `names` of this new object then overwrite `data` by this new object. We will do the same thing for the slots of our `trajectories`. `"setTime<-"` will enable the modification of the slot `times`. For this purpose, we shall use the function `setReplaceMethod`

```
> setGeneric("setTimes<-",function(object,value){standardGeneric("setTimes<-")})
[1] "setTimes<-"
> setReplaceMethod(
+     f="setTimes",
+     signature="Trajectories",
+     definition=function(object,value){
+         object@times <-value
+         return (object)
+     }
+ )
[1] "setTimes<-"
> getTimes(trajCochin)
[1] 1 2 4 5
> setTimes(trajCochin) <- 1:3
> getTimes(trajCochin)
[1] 1 2 3
```

Interesting part of the is that we can introduce some kind of control: As in `initialize`, we can explicitly call the inspector.

```
> setReplaceMethod(
+     f="setTimes",
+     signature="Trajectories",
+     definition=function(object,value){
+         object@times <- value
+         validObject(object)
+         return(object)
+     }
+ )
[1] "setTimes<-"
> setTimes(trajCochin) <- c(1,2,4,6)
~~~ Trajectories: inspector ~~~
> setTimes(trajCochin) <- 1:4
```

```
~~~ Trajectories: inspector ~~~
Error in validityMethod(object) :
  [Trajectories: validation] the number of temporal
measurements does not correspond to the number of columns of the matrix
```

7.3 The operator "["

It is also possible to define the getters by using the operator `[`. This can be made as for an unspecified method by specifying the class and function to be applied. This function takes for argument `x` (the object), `i` and `j` (which will be between "[" and "]") and `drop`. `i` will indicate the field which we want to reach. If the slot is a complex object (a matrix, a list,...), `j` will make it possible to specify a particular element. [[[I do not know the use of drop? Can someone help ???]]]

In our example, it can simply return the field corresponding to `i`

```
> setMethod(
+      f= "[",
+      signature="Trajectories",
+      definition=function(x,i,j,drop){
+          if(i=="times"){return(x@times)}else {}
+          if(i=="traj"){return(x@traj)}else {}
+      }
+  )
[1] "["
> trajCochin["times"]
[1] 1 2 4 6
> trajCochin["traj"]
     [,1] [,2] [,3] [,4]
[1,] 15.0 15.1 15.2 15.2
[2,] 16.0 15.9 16.0 16.4
[3,] 15.2   NA 15.3 15.3
[4,] 15.7 15.6 15.8 16.0
```

In the same way, one can define the setters by using the operator `[<-`. One can define it by using `setReplaceMethod`. Here again, a function describes the behavior to be adopted:

```
> setReplaceMethod(
+     f="[",
+     signature="Trajectories",
+     definition=function(x,i,j,value){
+         if(i=="times"){x@times<-value}else{}
+         if(i=="traj"){x@traj<-value}else{}
+         validObject(x)
+         return (x)
+     }
+ )
```

```
[1] "[<-"
> trajCochin["times"] <- 2:5
~~~ Trajectories: inspector ~~~
```

In our redefinition of `[` and `[<-`, we enumerated the various possible attributes for `i`. It would also be possible to number them: `if(i==1){return(x@times)}`. It is obviously completely dirty, because one would be exposed to any typographical error `trajCochin[2]` instead of `trajCochin[1]`.

7.4 "[", "@" or "get"?

When shall we use `get`, when shall we use `@`, when shall we use `[`?

`@` is should be used exclusively inside internal methods of the class: if a method for `Partition` needs to access to a slot of another class (example: `traj`), it is strongly recommended to not use `@` to reach the field directly but to go through `get` (example: `getTraj`).

Inside a class (if a `Partition` method needs `nbClusters`), there are two ways of proceeding: those who use `@` all the time and those who uses `get` all the time. Mixing is not recommended.

Between `get` and `[`, there is no big difference. Which one should we choose? Personally, I prefer to reserve `[` to direct access in a vector or in a matrix and to define the accessors `get` explicitly for the objects. But the difference between `getTimes(trajCochin)` and `trajCochin["time"]` is subtle... It is more a question of taste. It's up to you!

Part III
To go further

Next steps include signatures, inheritance and some other advanced concepts. If you are really a beginner in object programming, it is probably time for you to take a break and integrate what has been presented before. What we have just seen is enough to create some little classes... Your difficulties, amongst other things, will allow you to better understand what will follow...

You're done? Then, here are the next step:

8 Methods using several arguments

We made our first object. The following is more about interactions between objects. Thus, it is time to define some other objects. At first, `Partition`.

8.1 The problem

This is a tutorial. So we will not define `Partition` and all the methods which are require for each new class but only what we need:

```
> setClass(
+     Class="Partition",
+     representation=representation (
+         nbGroups="numeric",
+         part="factor"
+     )
+ )
[1] "Partition"
> setGeneric("getNbGroups",function(object){standardGeneric("getNbGroups")})
[1] "getNbGroups"
> setMethod("getNbGroups","Partition",function(object){return(object@nbGroups)})
[1] "getNbGroups"
> setGeneric("getPart",function(object){standardGeneric("getPart")})
[1] "getPart"
> setMethod("getPart","Partition",function(object){return(object@part)})
[1] "getPart"
> partCochin <- new(Class="Partition",nbGroups=2,part=factor(c("A","B","A","B")))
> partStAnne <- new(Class="Partition",nbGroups=2,part=factor(rep(c("A","B"),c(50,30))))
```

We will suppose that `part` is always composed of capital letters going from `A` to `LETTERS[nbGroups]` (it will be necessary to specify in the documentation of this class that the number of groups must be lower to 26). We can allow us such an assumption by programming `initialize` and `part<-` to always check that it is the case.

We thus have an object `trajCochin` of class `Trajectories` and a partition of this object in an object `partCochin` of class `Partition`. When we represent `trajCochin` graphically, we obtain a plot with multicoloured curves. Now that the trajectories are associated with groups, it would be interesting to draw the curves by giving a color to each group. Therefore, we would like the function `plot` to draw an object `Trajectories` according to some information that are in an object `Partition`. It is possible through the use of `signature`.

8.2 Signature

The signature, as we saw during the definition of our first methods section 5.1 page 19, is the second argument provided to `setMethod`. Up to now, we used simple signatures made up of only one class. In

`setMethod(f="plot",signature="Trajectories",definition=function....)`,

the signature is simply `"Trajectories"`

In order to have functions whose behaviors depend on several objects, we need to define a "signature vector" that will contain several classes. Then, when we call a method, R seeks the signature which corresponds the best to it and applies the corresponding function. A little example will make thing clearer. We will define a function `test`

```
> setGeneric("test",function(x,y,...){standardGeneric("test")})
[1] "test"
```

This function has a behavior if its argument is `numeric`, another behavior if it is a `character`.

```
> setMethod("test","numeric",function(x,y,...){cat("x is numeric =",x,"\n")})
[1] "test"
> ### 3.17 being a numeric, R will apply the test method for the test numeric
> test(3.17)
x is numeric = 3.17
> ### "E" being a character, R will not find a method
> test("E")
Error in function (classes, fdef, mtable)  :
  unable to find an inherited method for
function "test", for signature "character"
> setMethod("test","character",function(x,y,...){cat("x is character = ",x,"\n")})
[1] "test"
```

```
> ### Since "E" is a character, R now apply the method for character.
> test("E")

x is character =  E

> ### But the method for numeric is still here:
> test(-8.54)

x is numeric = -8.54
```

More complicated, we wish that `test` shows a different behavior if one combines a `numeric` and a `character`.

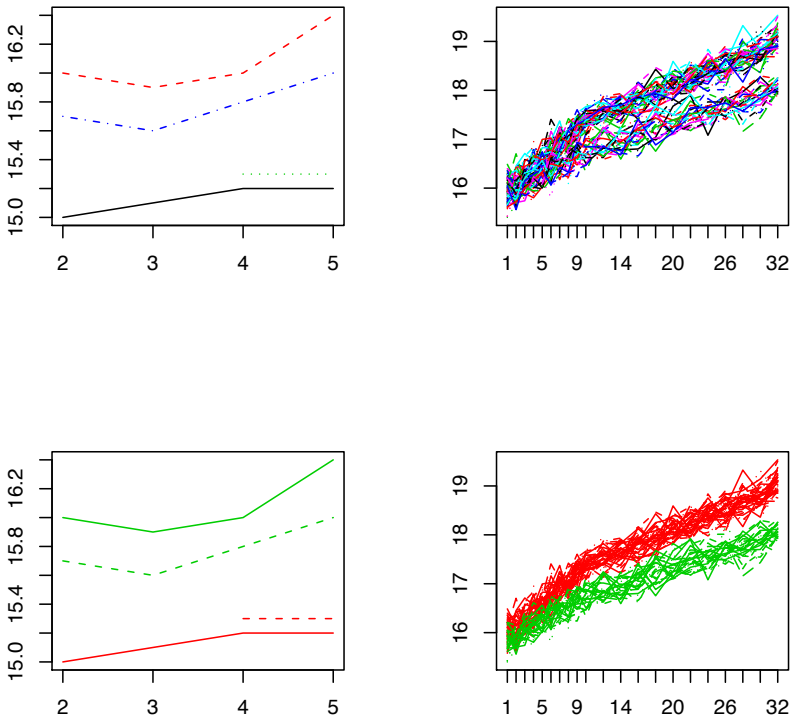
```
> ### For a method which combines numeric and character:
> setMethod(
+     f="test",
+     signature=c(x="numeric",y="character"),
+     definition=function(x,y,...){
+         cat("more complicated: ")
+         cat("x is numeric =",x," AND y is a character = ",y, "\n")
+     }
+ )

[1] "test"

> test(3.2, "E")

more complicated: x is numeric = 3.2  AND y is a character =  E

> ### The previous definition are still available
> test(3.2)

x is numeric = 3.2

> test("E")

x is character =  E
```

Now, R knows three methods to be applied to `test`: the first one is applied if the argument of `test` is a `numeric`; the second one is applied if the argument of `test` is a `character`; the third one is applied if `test` has two arguments, a `numeric` and then a `character`.

Back to our half real example. In the same way that we defined `plot` for the signature `Trajectories`, we now will define `plot` for the signature `c("Trajectories","Partition")` :

```
> setMethod(
+     f="plot",
+     signature=c(x="Trajectories",y="Partition"),
+     definition=function(x,y,...){
+         matplot(x@times,t(x@traj[y@part=="A",]),ylim=range(x@traj,na.rm=TRUE),
+                 xaxt="n",type="l",ylab="",xlab="",col=2)
+         for(i in 2:y@nbGroups){
```

```
+             matlines(x@times,t(x@traj[y@part==LETTERS[i],]),xaxt="n",type="l",col=i+1)
+         }
+         axis(1,at=x@times)
+     }
+ )
[1] "plot"
> par(mfrow=c(2,2))
> ### Plot for "Trajectory"
> plot(trajCochin)
> plot(trajStAnne)
> ### Plot for "Trajectory" plus "Partition"
> plot(trajCochin,partCochin)
> plot(trajStAnne,partStAnne)
```



Isn't it great?

8.3 Number of argument of a signature

Without entering the R's meanders, here are some rules concerning the signatures: a signature must count as many arguments as the methods to which it corresponds, neither

more, nor less. That means that it is not possible to define a method for `print` which will take into account two arguments since the argument of `print` is `x`. In the same way, `plot` can be defined only for two arguments, it is impossible to specify a third one in the signature.

8.4 "ANY"

Reversely, the signature must count all the arguments. Until now, we were not pay attention to that, for example we defined `plot` with only one argument. It is just a friendly user writing: R worked after us and added the second argument. As we had not specified the type of this second argument, it concluded that the method was to apply whichever the type of the second argument. To declare it explicitly, there is a special argument, the original class, the first cause: `ANY` (more detail on `ANY` section 9.1 page 42). Therefore, when we omit an argument, R gives it the name `ANY`.

The function `showMethods`, the same one which enabled us to see all the existing methods for an object section 5.4 page 24, makes possible to see the lists of the signatures that R knows for a given method:

```
> showMethods(test)
Function: test (package .GlobalEnv)
x="character", y="ANY"
x="character", y="missing"
    (inherited from: x="character", y="ANY")
x="numeric", y="ANY"
x="numeric", y="character"
x="numeric", y="missing"
    (inherited from: x="numeric", y="ANY")
```

As you can see, the list does not contain the signatures that we defined, but supplemented signatures: arguments that were not specified are replaced by `"ANY"`.

More precisely, `ANY` is used only if no different argument is appropriate. In the case of `test`, if `x` is a `numeric`, R hesitates between two methods. Initially, it tests to see whether `y` has a type which is defined. If `y` is a `character`, the method used will be the one corresponding to `(x="numeric",y="character")`. If `y` is not a `character`, R does not find the exact matching between `y` and a type, it thus uses the method hold-all: `x="numeric",y="ANY"`.

8.5 "missing"

It is also possible to define a method having a behavior if it has a single argument, another behavior if it has several. For that, we need to use `missing`. `missing` is true if the argument is not present:

```
> setMethod (
+     f="test",
+     signature=c(x="numeric",y="missing"),
```

```
+     definition=function(x,y,...){cat("x is numeric = ",x," and y is 'missing' \n")}
+ )

[1] "test"

>  ### Method without y thus using the missing
> test(3.17)

x is numeric =  3.17  and y is 'missing'

> ### Method with y='character'
> test(3.17, "E")

more complicated: x is numeric = 3.17  AND y is a character =  E

> ### Method with y='numeric'. y is not missing, y is not character, therefore "ANY" is used
> test (3.17, 2)

x is numeric = 3.17
```

9 Inheritance

Inheritance is at least 50 % of the power of the object... Put your belt on !

We will now define `TrajPartitioned`. We will be supposed to define the object, the constructors, the setters and the getters, posting... Everything. The smarter reader are already thinking that it will be a good start to do a copy-and-paste from to stick methods created for `Trajectories` and to adapt most of the code. Object programming provide some mechanism more efficient than that: inheritance.

9.1 Inheritance tree

A class `Son` can inherit a class `Father` when `Son` contains at least all the slots of `Father` (and may bo some others). Inheriting makes all the methods of `Father` available for `Son`.

More precisely, each time we use a method on an object of class `Son`, R will seek if this method exists. If it does not find it in the list of the specific methods `Son`, it will look for in the methods of `Father`. If it finds it, it will apply it. If it does not find it, it will look for in the methods which `Father` inherits. And so on.

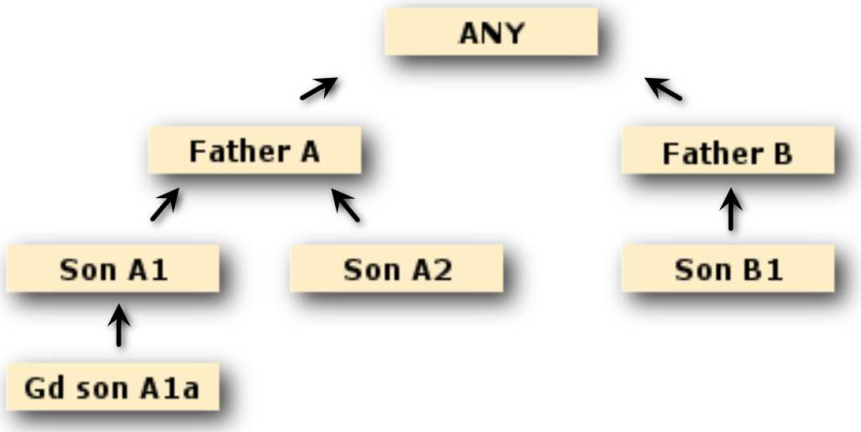
This raises the question of the origin, the ultimate ancestor, the root of roots. For human beings, it is - according to certain non checked sources- Adam. For objects, it is `ANY`. `ANY` is the first class, the one from which all the others inherit. Therefore, if a method is not found for the class `Son`, it will be sought in the class `Father`, then in `ANY`.

One represents the link which unifies the father and the son by an arrow going from the son towards the father. This symbolizes that when a method is not found for the son, R seeks in the father's methods.

`ANY <- classFather <- classSon`.

Usually the father is placed on the left of the son (or above him) since he is defined before.

Several classes can inherit from the same father. Finally, the chart of bonds linking the classes creates a tree (a data-processing tree, sorry for the poets who hoped for some green in this tutorial):



It is theoretically possible to inherit from several fathers. In this case, the graph is not a tree anymore. This practice seems to be natural but can be dangerous. Let us take an example: the class `CalculatingMachine` contains for example methods giving the precision of a calculation. The class `Plastic` explains the physical properties of plastic, for example what happens when it burns, its resistance... a computer is both a `CalculatingMachine` and something build in `Plastic`. It is thus interesting to have access at the same time to the method `precision` to know the power of the computer and the method `burning` to know how it will react in a fire. The class computer must thus inherit from two fathers.

On the other hand, the multiple heritage presents a danger, the non determinism of heritage. Let us suppose that A inherits from B and C. Let us suppose that an object of class A calls upon a method not existing in A but existing at the same time in B and C. Which method will be called for, the one for object B or the one for object C?

This is a source of confusion. It is particularly problematic for the use of `callNextMethod` (see 9.5 page 47).

Therefore, multiple heritage appears to be useful but dangerous. Anyway, it is possible to avoid it using `as`, as we will see section 9.6 page 49.

9.2 contains

Let's return to our classes. We want to define `TrajPartitioned` as heiress of `Trajectories`. For that, we have to declare the object adding the argument `contains` followed by the name of the father class.

```
> setClass(
+     Class="TrajPartitioned",
+     representation=representation(listPartitions="list"),
+     contains= "Trajectories"
+ )
[1] "TrajPartitioned"
> tdPitie <- new("TrajPartitioned")
~~~~~ Trajectories: initializator ~~~~~
```

9.3 unclass

`TrajPartitioned` contains all the slots of `Trajectories` plus its personal slot `listPartitions` (field which will contain one list of partition).

For the moment, it is not possible to check it directly. Indeed, if we try "to see" `tdCochin`, we only obtain a `Trajectories`, and not a `TrajPartitioned`.

```
> tdPitie
*** Class Trajectories, method Show ***
* Times = numeric(0)
* Traj (limited to a matrix 10x10) =
******* End Show (trajectories) *******
```

Why is it? `TrajPartitioned` is a heir of `Trajectories`. Each time a function is called, R seeks this function for the signature `TrajPartitioned`. If it does not find, it seeks the function for the parent class, namely `Trajectories`. If it does not find it, it seeks the function by defect.

An object is see using `show`.

As `show` does not exist for `TrajPartitioned`, it is `show` for `Trajectories` which is called instead. So writing `tdPitie` does not show us the object such as it really is, but via the prism of `show` for `Trajectories`. It is then urgent to define a method `show` for `TrajPartitioned`.

Nevertheless, it would be interesting to be able to look into the object that we have just created. In order to do this, we can use `unclass`. `unclass` removes the class of an object. So `unclass(tdPitie)` is calling the method `show` as if `tdPitie` has no class, that is the generic function (`show` for class `ANY`): The result is not very nice to look at, but the object is fully present.

```
> unclass(tdPitie)
<S4 Type Object>
attr(,"listPartitions")
list()
attr(,"times")
numeric(0)
attr(,"traj")
<0 x 0 matrix>
```

```
<S4 Type Object>
attr(,"listPartitions")
list()
attr(,"times")
numeric(0)
attr(,"traj")
<0 x 0 matrix>
```

So we can now check that the object thus comprises the slots `traj`, `times` (like its father `Trajectories`) plus a slot for a list `listPartitions`.

9.4 See the method by authorizing heritage

Heritage is a strength, but it can lead to strange results. Let us create a second `TrajPartitioned` object:

```
> partCochin2 <- new("Partition",nbGroups=3,part=factor(c("A","C","C","B")))
```

```
> tdCochin <- new(
+     Class="TrajPartitioned",
+     times=c(1,3,4,5),
+     traj=trajCochin@traj,
+     listPartitions=list(partCochin,partCochin2)
+ )

Error in getClass(Class, where = topenv(parent.frame())) :
  "TrajDecoupees" is not a
defined class
```

It doesn't work... Why? In order to find out, it is possible to post the method `initialize` called by `new`:

```
>  getMethod("initialize","TrajPartitioned")
```

```
[1] FALSE
```

```
Error in getMethod("initialize", "TrajPartitioned") :
  No method found for function
"initialize" and signature TrajPartitioned
```

It still does not work, but this time the cause is simpler to identify: we did not define `initialize` for `Partition`.

```
> existsMethod("initialize","TrajPartitioned")
```

```
[1] FALSE
```

Here is the confirmation of our doubts.

However, R appears to still run some code since it find an error somewhere. So what on earth is going on here[10]?

It is one of the adverse effects of the inheritance, a kind of involuntary inheritance. Indeed, when `new("TrajPartitioned")`, is called, `new` seeks the function `initialize` for `TrajPartitioned`. As it does not find it AND that `TrajPartitioned` inherits `Trajectories`, it replaces the missing method by `initialize` for `Trajectories`.

To check that, two methods: `hasMethods` enable to know if a method exists for a given class by taking into account the heritage. Remember, when `existsMethod` did not find a method, it indicated `False`. When `hasMethod` does not find a method, it seeks in the father, then in the grandfather and so on:

```
> hasMethod("initialize","TrajPartitioned")
[1] TRUE
```

Confirmation, `new` is indeed re-directed to an inherited method. To see this method, one can use `selectMethod`. `selectMethod` has the same overall behavior as `getMethod`. The only difference is that when it does not find a method, it seeks among ancestors... like `hasMethod`

```
> selectMethod ("initialize", "TrajPartitioned")
Method Definition:

function (.Object, ...)
{
    .local <- function (.Object, times, traj)
    {
        cat("~~~~~ Trajectories: initializator ~~~~~ \n")
        if (!missing(traj)) {
            colnames(traj) <- paste("T", times, sep = "")
            rownames(traj) <- paste("I", 1:nrow(traj), sep = "")
            .Object@times <- times
            .Object@traj <- traj
            validObject(.Object)
        }
        return(.Object)
    }
    .local(.Object, ...)
}

Signatures:
        .Object
target  "TrajPartitioned"
defined "Trajectories"
```

[10]By the way, please note all the treasures of imagination developed by the author to keep some suspense in a tutorial on statistics...

Our assumption was right, `TrajPartitioned` uses the initializator of `Partition`. As the latter does not know the field `listPartitions`, it indicates an error. The mystery is now solved.

So to be able to define a little more complex `TrajPartitioned` objects, it is necessary as a preliminary to define `initialize` for `TrajPartitioned`

```
> setMethod("initialize","TrajPartitioned",
+     function(.Object,times,traj,listPartitions){
+         cat("~~~~TrajPartitioned: initializator ~~~~ \n")
+         if(!missing(traj)){
+             .Object@times <- times
+             .Object@traj <- traj      # Assignment of attributes
+             .Object@listPartitions <- listPartitions
+         }
+         return(.Object)           # return of the object
+     }
+ )
[1] "initialize"
> tdCochin <- new(
+     Class="TrajPartitioned",
+     traj=trajCochin@traj,
+     times=c(1,3,4,5),
+     listPartitions=list(partCochin,partCochin2)
+ )
~~~~TrajPartitioned: initializator ~~~~
```

Here we are!

9.5 "callNextMethod"

We have just seen it: when R does not find a method, it has a mechanism allowing to replace it by an inherited method. It is possible to control this mechanism and to force a method to call the method inherited. This allows, amongst other things, to use a code without having to type it again. This is possible by using `callNextMethod`. `callNextMethod` can only be used in a method. It calls *the method which would be used if the current method did not exist*. For example, let us consider the method `print` for `TrajPartitioned`. Currently, this method does not exist. However, a call for `print(tdCochin)` would be carried out by using inheritance:

```
> print (tdCochin)
*** Class Trajectories, method Print ***
* Times =[1] 1 3 4 5
* Traj =
     [,1] [,2] [,3] [,4]
[1,] 15.0 15.1 15.2 15.2
[2,] 16.0 15.9 16.0 16.4
```

```
[3,] 15.2   NA 15.3 15.3
[4,] 15.7 15.6 15.8 16.0
******* End Print (trajectories) *******
```

As `print` does not exist for `TrajPartitioned`, it is `print` for `Trajectories` which is called. In other words, the `nextMethod` of `print` for `TrajPartitioned` is `print` for `Trajectories`.

A little example could be useful. We will define `print` for `TrajPartitioned` [11].

```
> setMethod(
+     f="print",
+     signature="TrajPartitioned",
+     definition=function(x,...){
+         callNextMethod()
+         cat("the object also contains",length(x@listPartitions),"partition")
+         cat("\n ***** Fine of print (TrajPartitioned) ***** \n")
+         return(invisible())
+     }
+ )
```

```
[1] "print"
```

```
> print(tdCochin)
```

```
*** Class Trajectories, method Print ***
* Times =[1] 1 3 4 5
* Traj =
     [,1] [,2] [,3] [,4]
[1,] 15.0 15.1 15.2 15.2
[2,] 16.0 15.9 16.0 16.4
[3,] 15.2   NA 15.3 15.3
[4,] 15.7 15.6 15.8 16.0
******* End Print (trajectories) *******
the object also contains 2 partition
 ***** Fine of print (TrajPartitioned) *****
```

`callNextMethod` can either take explicit arguments, or no argument at all. In this case, the arguments which were given to the current method are completely shifted to the following method.

Whose method is the following one? Here lies all the difficulty and the ambiguity of `callNextMethod`. In most cases, very experimented people know, the less experienced don't. But where the method becomes completely unsuitable is when the following method depends on the structure of another class. Finally, nobody can know. Example: consider a class `A` that we program. Let assume than `A` inherits from class `B`, a class that somebody else programmed. Which is the method following `initialize` for `A`? It all depends. As `A` inherits, R seeks in the order:

[11] Naturally, in a real case, we would make much more than to print the trajectories and the number of partition.

- `initialize` for `B`
- Default `initialize` (the one for `ANY`). This method ends in a call for `validObject`.
- `validObject` for `A`
- `validObject` for `B`
- Default `validObject` (the one for `ANY`).

From then, it is *not* possible to know *because it does not depend on us anymore!* If the programmer of `B` defined an `initialize`, it will be called and `validObject` for `B` will perhaps be called, and perhaps not. If not, it is the default `initialize` which is called, and the `validObject` for `A`, for `B` or the default one, according to what exists. It is thus very difficult to know what the program really does.

Worse, if the programmer of `B` removes or adds an `initialize`, it can change the behavior of our method radically. For example, if `initialize` for `B` does not exist and `validObject` for `A` exists, the default `initialize` is called, and then `validObject` for `A`. If the `initialize` method for B is created, default `initialize` will not be used anymore and `validObject` might not be used anymore either. Probable... But one does not really know.

Unpleasant uncertainty, isn't it? For this reason the use of `callNextMethod` should be limited, especially knowing that `as` and `is` enable to do almost the same thing...

9.6 "is", "as" and "as<-"

When an object inherits from another, we can require that it adopts temporarily the behavior which its father would have. For that, it is possible to transform it into an object of the class of its father by using `as`. For example, if one wants to print `tdPitie` by considering only its `Trajectories` aspect:

```
> print(as(tdPitie,"Trajectories"))
~~~~~ Trajectories: initializator ~~~~~
*** Class Trajectories, method Print ***
* Times =numeric(0)
* Traj =
<0 x 0 matrix>
******* End Print (trajectories) *******
```

That will be useful to us in the definition of `show` for `TrajPartitioned`:

```
> setMethod (
+     f="show",
+     signature="TrajPartitioned",
+     definition=function(object){
+         show(as(object,"Trajectories"))
+         lapply(object@listPartitions,show)
+     }
+ )
```

```
[1] "show"
```

We can check if an object is the heir of another by using `is`. `is` checks that the attributes of the object are present in the son's class:

```
> is(trajCochin,"TrajPartitioned")
[1] FALSE
> is(tdCochin,"Trajectories")
[1] TRUE
```

Lastly, `as<-` enables to modify only the slot which an object inherits from its father. `as(objectSon,"ClassFather") <- objectFather` affects the contents of the slots that `objectSon` inherit from its father.

```
> ### Creation of empty TrajPartitioned
> tdStAnne <- new("TrajPartitioned")
~~~~TrajPartitioned: initializator ~~~~
> ### Assignment of Trajectories to the attributes of TrajPartitioned
> as(tdStAnne,"Trajectories") <- trajStAnne
> tdStAnne
~~~~~ Trajectories: initializator ~~~~~
*** Class Trajectories, method Show ***
* Times =  [1]  1  2  3  4  5  6  7  8  9 10 12 14 16 18 20 22 24 26 28 30 32
* Traj (limited to a matrix 10x10) =
      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9]  [,10]
 [1,] 15.91 15.97 16.58 16.52 16.69 16.49 17.15 16.86 17.21 17.35
 [2,] 16.2  15.65 16    16.78 16.64 16.89 17    17    16.96 17.34
 [3,] 16.07 15.97 16.37 16.07 16.6  17.02 17.03 16.94 17.32 17.34
 [4,] 15.89 16.47 16.44 16.75 16.93 16.99 16.94 17.17 16.97 17.34
 [5,] 15.97 16.21 16.14 16.09 16.62 16.78 16.97 17.04 17.2  17.47
 [6,] 16.15 15.95 15.96 16.31 16.43 17.16 16.84 16.81 17.09 17.45
 [7,] 16.04 16.07 16.32 16.08 16.5  16.74 17.21 17.13 17.51 17.13
 [8,] 16.2  16.15 16.28 16.14 16.25 16.96 16.81 16.92 17.27 16.95
 [9,] 15.62 16.17 16.37 16.41 16.15 17.01 17.01 16.95 17.54 17.33
[10,] 15.71 15.93 16.28 16.37 16.62 16.67 16.68 17.05 17.19 17.24
******* End Show (trajectories) *******
```

9.7 "setIs"

In the case of a heritage, `as` and `is` are defined "naturally", as we have just seen before. It is also possible to specify them "manually". For example, the class `TrajPartitioned` contains a list of `Partitions`. It does not inherit directly from `Partition` (it would be a case of unsuitable multiple heritage), therefore `is(tdCochin,"Partitions")` and `as(tdCochin,"Partitions")` are not defined by default.

It is nevertheless possible to "force" it. For example, we want to be able to look at an object of class `TrajPartitioned` such as a `Partition`, the one which has the greatest number of groups [12].

This can be done with the instruction `setIs`. `setIs` is a method which takes four arguments

- `class1` is the class of the initial object, the one which must be transformed.
- `class2` is the class into which the object must be transformed
- `coerce` is the function used to transform `class1` into `class2`. It uses 2 arguments, `from` correspond to `class1` and `to` correspond to `class2`.

```
> setIs(
+     class1="TrajPartitioned",
+     class2="Partition",
+     coerce=function(from,to){
+         numberGroups <- sapply(tdCochin@listPartitions,getNbGroups)
+         Smallest <- which.min(-numberGroups)
+         to<-new("Partition")
+         to@nbGroups <- getNbGroups(from@listPartitions[[Smallest]])
+         to@part <- getPart(from@listPartitions[[Smallest]])
+         return(to)
+     }
+ )
> is(tdCochin,"Partition")
[1] TRUE
> as(tdCochin,"Partition")
An object of class "Partition"
Slot "nbGroups":
[1] 3

Slot "part":
[1] A C C B
Levels: A B C
```

It works for what we needed. But a `Warning` appears. R indicates that `as<-` is not defined. `as<-` is the operator used to affect a value to an object *while* it is considered as another object. In our case, `as<-` is the operator used to modify `TrajPartitioned` whereas it is regarded as `Trajectories`. This can be carried out by using the fourth argument `setIs`:

[12] Actually, such a use would be unsuitable: a `TrajPartitioned` should not become a `Partition`. If we want to reach a particular partition, we should define a getter (for example `getPartitionMax`. Sorry for breaking the rules of clean programming, it is just to illustrate the use of `setIs` without introducing a new class.)

- `replace` is the function used for assignments.

In this case, one wants to replace the partition having the smallest number of groups by a new one:

```
> setIs(
+     class1="TrajPartitioned",
+     class2="Partition",
+     coerce=function(from,to) {
+         numberGroups <- sapply(tdCochin@listPartitions,getNbGroups)
+         largest <- which.min(-numberGroups)
+         to <- new("Partition")
+         to@nbGroups <- getNbGroups(from@listPartitions[[largest]])
+         to@part <- getPart(from@listPartitions[[largest]])
+         return(to)
+     },
+     replace=function(from,value){
+         numberGroups <- sapply(tdCochin@listPartitions,getNbGroups)
+         smallest <- which.min(numberGroups)
+         from@listPartitions[[smallest]] <- value
+         return(from)
+     }
+ )
> as(tdCochin,"Partition")
An object of class "Partition"
Slot "nbGroups":
[1] 3

Slot "part":
[1] A C C B
Levels: A B C

> as(tdCochin,"Partition") <- partCochin2
```

No `Warning` anymore, everything is fine!

Optionally, one can add a fifth argument, the function `test`: it subordinates the transformation of `class1` into `class2` following a condition.

9.8 Virtual classes

It can happen that classes are close to each other without one being the extension of the other. For example, we can conceive two types of partition: partition which "labels" individuals without judging them and those which evaluate individuals. The first type of partition will not be ordered (same as `Partition`) and the second type of partition will be ordered (for example, it will classify the individuals in `Insufficient, Medium, Good`). Slots of this second class will be `nbGroups`, an `integer` which indicates the number of modalities and `part`, an ordered variable which will indicate the membership group. Clearly, the slots are not the same in the two classes since `part` is ordered in one

and not ordered in the other. Thus one cannot have inherited from the other. However, the methods will undoubtedly be similar in a considerable number of cases.

In order to prevent the user from having to program them twice, one can used a virtual class. A virtual class is a class for which it is not possible to create objects but which can have heirs. The heirs profit from the methods created for the class. In our case, we will create a virtual class `PartitionFather` then two son classes `PartitionSimple` and `PartitionEval`. All the common methods will be created for `PartitionFather`, both sons will inherit from it.

```
> setClass(
+     Class="PartitionFather",
+     representation=representation(nbGroups="numeric","VIRTUAL")
+ )
```

```
[1] "PartitionFather"
```

```
> setClass(
+     Class="PartitionSimple",
+     representation=representation(part="factor"),
+     contains="PartitionFather"
+ )
```

```
[1] "PartitionSimple"
```

```
> setClass(
+     Class="PartitionEval",
+     representation=representation(part="ordered"),
+     contains="PartitionFather"
+ )
```

```
[1] "PartitionEval"
```

```
> setGeneric("nbMultTwo",function(object){standardGeneric("nbMultTwo")})
```

```
[1] "nbMultTwo"
```

```
> setMethod("nbMultTwo","PartitionFather",
+     function(object){
+         object@nbGroups <- object@nbGroups*2
+         return (object)
+     }
+ )
```

```
[1] "nbMultTwo"
```

```
> a <- new("PartitionSimple",nbGroups=3,part=factor(LETTERS[c(1,2,3,2,2,1)]))
> nbMultTwo(a)
```

```
An object of class "PartitionSimple"
Slot "part":
[1] A B C B B A
Levels: A B C

Slot "nbGroups":
[1] 6
```

```
> b <- new("PartitionEval",nbGroups=5,part=ordered(LETTERS[c(1,5,3,4,2,4)]))
> nbMultTwo(b)

An object of class "PartitionEval"
Slot "part":
[1] A E C D B D
Levels: A < B < C < D < E

Slot "nbGroups":
[1] 10
```

Here we are...

9.9 For dyslexics...

For dyslexics (of which I am) who mix their neurons rather quickly and who never know if the heir can use the techniques of the father or if it is the opposite, here is a little story: it comes from "Contes et Légende de la naissance de Rome"[**?**] ("Tales and Legends of the Birth of Rome"). The author explains that, *contrary to human beings*, Gods do not like to have children who become more powerful than them. This is why Saturn devours his children (for those who are emotional, be reassured: it is not too serious for a god to be devoured by his father since when Saturn get finally cut into pieces by his son Jupiter, all the children come out from his big (!) stomach and are quite alive!). For human beings, it is the opposite: the fathers are rather proud when their children are taller, beat them at tennis, obtain a greater level of study... For objects, it is similar to the behavior of human beings: *the sons are stronger than the fathers*[13] So a son can do everything the father does plus what it is able to do itself...

10 Internal modification of an object

The continuation requires to go down a little deeper in R's meanders... (don't get lost!)

10.1 R internal working procedure: environments

An environment is a workspace that store variables. To simplify, R works by using two environments: *global* and *local*. The global is the one to which we have access when we enter instructions in the console. The local is an environment which is created with each call of function. Then when the function is finished, the local is destroyed. Example:

```
> func <- function () {
+     x <- 5
+     cat(x)
```

[13] For the meta-dyslexics who will remember this story but who will not know if the objects behave like Gods or like human beings, I can't do anything... [14]

[14] Suggestion of a lector: objects are full of bugs, *like human beings*! (Whereas everyone knows that Gods are perfect.) This is for the meta-dyslexics.

```
+     return(invisible ())
+ }
> ### Creation of x in "global"
> x <- 2
> ### Call of the function: "local" x is create and printed
> func()

5

> ### Return to global: "local" x is removed
> x

[1] 2
```

What did happen there? The function `func` is defined in the global environment. Still in the global, `x` receives value 2. Then function `func` is called. R creates the local environment. In this environment, it gives `x` the value 5 (only in the local environment). At this stage, there exists *two* `x`: a global one which is 2 and a local one which is 5.

The function print the local `x`, then finishes. The local environment is destroyed. Thus, the `x` which was 5 disappears. There only remains `x` which is worth 2, the global one. It is print.

10.2 Method to modify a field

Let us return to our trajectories. There remains a third method to define, the one which imputes the variables. To simplify, we will impute while replacing by the mean values [15].

```
> meanWithoutNa <- function (x){mean(x,na.rm=TRUE)}
> setGeneric("impute",function (.Object){standardGeneric("impute")})

[1] "impute"

> setMethod(
+     f="impute",
+     signature="Trajectories",
+     def=function(.Object){
+         average <- apply(.Object@traj,2,meanWithoutNa)
+         for (iCol in 1:ncol(.Object@traj)){
+             .Object@traj[is.na(.Object@traj[,iCol]),iCol] <- average[iCol]
+         }
+         return(.Object)
+     }
+ )

[1] "impute"

> impute(trajCochin)
```

[15] For trajectories, this method is not very good, it would be better to use the average of the values above and below the missing value. But as the aim is to learn S4, we choose the simplest way.

```
*** Class Trajectories, method Show ***
* Times = [1] 2 3 4 5
* Traj (limited to a matrix 10x10) =
     [,1] [,2]  [,3] [,4]
[1,] 15   15.1  15.2 15.2
[2,] 16   15.9  16   16.4
[3,] 15.2 15.53 15.3 15.3
[4,] 15.7 15.6  15.8 16
******* End Show (trajectories) *******
```

The method `impute` works correctly. On the other hand, it does not modify `trajCochin`.

```
> trajCochin
*** Class Trajectories, method Show ***
* Times = [1] 2 3 4 5
* Traj (limited to a matrix 10x10) =
     [,1] [,2] [,3] [,4]
[1,] 15   15.1 15.2 15.2
[2,] 16   15.9 16   16.4
[3,] 15.2 NA   15.3 15.3
[4,] 15.7 15.6 15.8 16
******* End Show (trajectories) *******
```

In the light of what we have just seen on environments, what does this method do? It creates *locally* an object `.Object`, it modifies its trajectories (impute by average) then it return an object. Thus, `impute(trajCochin)` did not have any effect on `trajCochin`.

Conceptually, it is a problem. Of course, it is easy to circumvent it simply by using

```
> trajCochin <- impute(trajCochin)
```

But the role of the function `impute` is to modify the internal slot of the object, not to create a new object and to reallocate it. In order to give it back its original meaning, we can use `assign`.

`assign` is one of the most dirty existing operators... In local environment, it enables to modify variables at a global level. It is *very very* bad! But in the present case, it is precisely what R does not offer, and which is very frequent in object programming. Therefore, we allow ourselves a little trespassing of the rules (do not tell anyone!)

So we rewrite `impute` by adding two things: `deparse(substitute())` allows to know the name of the variable which was transmit as an argument to the function in charge of the execution at the global level.

`assign` makes possible the modification of a variable of higher level. More precisely, there is not "a" local level, but local levels. For example, a function *in* another function creates a *local in the local*, a kind of *under-local* or of *local level 2*. The use of `assign` which we propose here does not affect the global level, but the level above the actual level. To modify the global one directly would be even more unsuitable...

Let try on a toy example:

```
> testCarre <- function(x){
+     nameObject <- deparse(substitute(x))
+     print(nameObject)
+     assign(nameObject,x^2,envir=parent.frame())
+     return(invisible())
+ }
> a<-2
> testCarre(a)
[1] "a"
> a
[1] 4
```

It works. So here is the new `impute`. To use it, no more need for allocation. For `Trajectories`, we get:

```
> setMethod(
+     f="impute",
+     signature="Trajectories",
+     def=function(.Object){
+         nameObject<-deparse(substitute(.Object))
+         average <- apply(.Object@traj,2,meanWithoutNa)
+         for (iCol in 1:ncol(.Object@traj)){
+             .Object@traj[is.na(.Object@traj[,iCol]),iCol] <- average [iCol]
+         }
+         assign(nameObject,.Object,envir=parent.frame())
+         return(invisible())
+     }
+ )
[1] "impute"
> impute(trajCochin)
```

It works. Once again, `assign` is a dirty operator which should be handled with much precaution, it can cause catastrophes perfectly counter intuitive. But for internal modification, it is the only solution we found up to now.

Part IV
Appendices

[[[Note: all the appendices (except acknowledgments) are under construction and perfectibles. Well, all chapters are perfectibles, but appendices are even more perfectibles than other parts...]]]

A Acknowledgments

A.1 We live in a wonderful time!

When people are living a historical period, a revolution, a key date, one is not always aware of it. I think that the creation of Internet will be considered by future historians as a major development, something as enormous as the writing invention or printing press. Writing is the conservation of information. Printing press is the diffusion of information to an elite, then to everyone, but with a cost. Internet is instantaneity, it can be shared by all, experts' knowledge becomes available to all... Forums are the end of questions without answers...

Three months ago, I knew of S4 only the name. In a few months, I was able to acquire a knowledge thanks to people that I do not know, but who helped me. Of course, I read. But as soon as I had a question, I posted it in the evening, I slept the sleep of the just, and the following morning, I had my answer! It is free, it is altruism, it is simply beautiful! We're living a wonderful time...

A.2 Thanks so much!

So I wish to thanks many people that help me learning S4 or writing this tutorial, some of them I did not even know. Pierre Bady, a priceless reader, made me very relevant remarks, in particular on the general structure of the document that was not very well organized... Martin Morgan who not only knows EVERYTHING about S4, but also replies more quickly than his shade when a question needs an answer on r-help... Many thanks also to the CIRAD team which animates the forum for the Group of the R Users. None is ever left aside, nobody is told RTFM or GIYF [16]. That is really cool. I also wish to thank Antoine who will probably have to read it all over again, even if he does not know it yet... Thanks to Rebecca and Laura for the translation. Thanks to Bruno who gave me (by decreasing order of importance) a team [17], his love of R, a research topic, ideas, contacts, an office, a very powerful computer... Without him, I would probably still be vegetating in a dead end. Thanks to R-core TEAM for "the free gift of R". And last but not least, thanks to all the reader (may be you?) that will post me english corrections, comments, suggestion of improvement, needs...

[16] *Read The Fucking Manual* or *Google Is Your Friend*, typical answers given to those who ask questions without having taken the time to look for answers by themselves.

[17] The BEEEEEEEEEST of all teams!

B Good practices

Good practices are a set of rules that one chooses to follow in order to decrease the number of bugs present in one's program. Some of these rules are adaptated form classic set of good practices that we can find on the web or in textbooks ; some other are from R help list, french R forum [**?**] or (see [**?**]).

B.1 Code structuration

- Use a smart editor (with bracket highlighting and indentation).
- Do indent line (add initial space) to underline the architecture of your code.
- Do use space in your code, specially around `<-` or `==`.
- Each closing brace or bracket must be under the instruction corresponding to the opening one.
- Do not omit the optional braces.
- Separate the blocks from the instruction by jumping lines.
- All your conditions must comprise an `else`, even if it is empty one. If not: `if (cond1) if (cond2) cat ("A") else cat("E")` is undefined, we do not know to which if the `else` is referring to `cond1` or `cond2` (it is called "ambiguous" and can be compiler dependent).
- Segment your code.
- Do not copy-and-paste. If some code is share by two application, make a function instead.

In France, after an exam, if you get a note lower than 8, you fail ; upper than 10, you succeed ; between 8 and 10, you can take the exam again two weeks later.

What is wrong in the following code ?

```
for(x in 1:100){
if(note[x]<10){if(note[x]<8){cat("Fail")
}else{cat("You get it!")}})
```

Here is the same code than above after applying the rules we just define:

```
for(x in 1:100){

    if(note[x]<10){
        if(note[x]<8){
            cat("Fail")
    }else{
        cat("You get it!")
    }
```

The `{` of the `for` does not have a closing `}`
The `if(note[x]<8){` has no `else`, so the second `else` is taken instead.
The correct code would be :

```
for(x in 1:100){

    if(note[x]<10){
        if(note[x]<8){
            cat("Fail")
        }else{}
    }else{
        cat("You get it!")
    }

}
```

B.2 Variables

- Explicitly name your variables by beginning each word with a capital letter except the first word (`numberOfSons`)
- Keep names of reasonable size: `numberOfSonsWithAgeToGoToUniversity` is not a good name...
- ... but `ndfeaaau` is not explicit.
- Use names starting with a lower case letter for the variables and functions and by a capital letter for the classes.
- Alternative (Hungarian notation): all the variables must start with a letter which gives their type: `iNumberSons`, `nSize`, `lListInvited`
- In S4, do not use a default value for slots. A variable which is not initialized HAVE TO cause an error.
- The arguments of a function can be specified according to the order established in the definition of the function. However, it is highly recommended to specify the arguments by their name.
- Never use global variables. A function shall work even if its' environment changes. For example, if you copy-paste a function in another program, it shall still work.

What is the following code about? Are there any bugs? What is `m`?

```
b <- c(16,16.1,16.4,16.5)
a <- 15
n <- 4
m <- sum(b)/a
```

Here is the same code with some explicit variable names.

```
BmiLyse <- c(16,16.1,16.4,16.5)
ageLyse <- 15
numberOfMesure <- 4
meanBmi <- sum(bmiLyse)/ageLyse
```

It is easy to see what is wrong: to evaluate the `meanBmi`, we should have divided by `numberObMeasure` instead of `ageLyse`.

B.3 Comment and documentation

- Comment on your code.
- Comment on your code intelligently: on the line `I <- 2`, to add the comment `# Set I to 2` is not useful at all.
- Document the entries and exits of each function. A user must be able to use your function without having to read the code. It is thus essential to specify well what he must enter and what the function returns.
- Do not use abbreviations. For example, use FALSE/TRUE and not 0/1 or F/T

Package `kml` [**?**] defines several functions that work with trajectories. One of them is about imputation. Here are the comments:

```
### imputeTraj needs two arguments
###  - matrixToImpute is a matrix with or without missing values
###  - method is a character string in "LOCF", "mean", "multiple"
###
### ImputeTraj returning a matrix without missing values
###
imputeTraj <- function(matrixToImpute,method){
    ....
    return(matrixImputed)
}
```

So even without reading the code, one may be able to use this function.

B.4 Programming tricks

- Test your code. Test your code regularly (do not write a long code then test it ; write small pieces of code (short function) and test each as you go along).
- Instead of `x[ind ,]`, use `x[ind, , drop = FALSE]`.
- Instead of `x == NA`, use `is .na(x)`
- Instead of `1:length(obj)`, use seq(along = obj)

- Do not attach `data.frame`.
- Do not use `=`, use `<-` instead.
- Do not try to write optimal code. Write the code clearly and simply. Later, when it is working, much much latter, optimization can begin.
- The loops are ineffective in R; it is more effective to delegate the loops to the functions `lapply` and `sapply`. (Not really a good practice, but rather a good habit)

B.5 Debugging of methods

[Personal opinion]: It is easier to debug a function than a method. [[[This needs to be checked. Can it give false information? Is the behavior in vivo the same one as in vitro?]]] Hence, it is simpler to define a function used by a method *apart*, then we can checked it in order to see if it works correctly before finally integrating it in the method. Lastly, once the method is declared, the function can be removed without affecting the method.

Why should we remove it? Simply in order not to leave variables, functions or objects which will no longer be used in the workspace. It is a bit like cleaning up your house: the cleaner it is, the fewer bugs there are...

Therefore, instead of:

```
> setMethod (
+     f= " methodA ",
+     signature= " classA ",
+     definition=function () {cat ("Blah")}
+ )
```

we can write:

```
> .classA.methodA <- function () {cat ("Blah")}
> ### Here, we test .classA.methodA:
> ###   - seek bugs
> ###   - detection of global variables with findGlobals
> ###   - testing values
> ###   - ...
>
> ### Then the method definition
> setMethod (
+     f="methodA",
+     signature="classA",
+     definition=.classA.methodA
+ )
> ### And the cleaning
> rm(.classA.methodA)
```

B.6 Partition of the classes into files

As we saw in the introduction, it is cleaner to encapsulate methods concerning the same object together in one file. It is also necessary to test our code. It is better to separate the testing part from the method definition. The reason for this is that when you finish to develop your code, you no longer want to run tests. If the tests are in the same file as the method definition, you have to remove them one by one. If you decide to change your code, you will have to write these tests again.

The alternate way is to write your tests in separate file. During the development of your code, you will run the test files and the method definition files. When the development is over, simply run the method definition files.

So the general structure of your program may look like:

- A file `ClassA.R` for each class `A`. It contains comments on the class, the definition of the slot and all the methods for the class. It does not contain any tests.
- A file `testA.R` containing tests (called *regression test*) for any methods define in `ClassA.R`. This file should start with the line `source("ClassA.R")` (or `source("../R/ClassA.R")` if your test files and your code files are not in the same directory).
- A file `main.R` calling all the test files (`source("testA.R");source("testB.R");`) during the development step, all the classes' definition when the development is over (`source("ClassA.R");source("ClassB.R");`).

To run your program, you just have to write `source("main.R")`.

C Memo

So here we are, you read it all, understood it all and you are now programming your own objects. Just a little memory failure, what on earth was the name of the third field of `validObject`?

This is what this memo is for.

C.1 Creation

```
> ### Class creation
> setClass (
+     Class= "NewClass",
+     representation=representation(x="numeric",y="character"),
+     prototype=prototype (x=1,y="A"),
+     contains=c("FatherClass"),
+     validity=function(object){return(TRUE)}
+ )
> ### Object creation
> new(Class="NewClass")
> A <- new(Class="NewClass",x=2,y="B")
```

```
> ### Slot manipulation
> A@x <- 4 # (iark!)
> ### Class destruction (partial)
> removeClass("NewClass")
> ### Constructor
> newClass <- function(){
+     ....
+     return(new(Class="NewClass"))
+ }
```

C.2 Validation

```
> setMethod(f="initialize",signature="NewClass",
+     definition=function(.Object,value){
+         if(....){stop("initialize (NewClass): Error")}else{}
+         .Object@x <- value;
+         validObject(.Object)
+         return(.Object)
+     }
+ )
```

C.3 Accessor

```
> ### Getter
> setGeneric(name="getX",def=function(object){standardGeneric("getX")})
> setMethod(f="getX",signature="NewClass",
+     definition=function(object){return(object@x)}
+ )
> ### Setter
> setGeneric(name="setX<-",def=function(object,value){standardGeneric("setX<-")})
> setReplaceMethod(f="setX",signature="NewClass",
+     def=function(object,value){object@x<-value;return(object)}
+ )
```

C.4 Methods

```
> ### To create a generic method
> setGeneric(f="newFunction",def=function(z,r){standardGeneric("newFunction")})
> ### To declare a method
> setMethod(f="newFunction",signature="NewClass",
+     def=function(z,r){....;return(....)}
+ )
> ### To get the arguments of a function
> args(NewFunction)
```

C.5 Some essential functions

- `args(print)` : (x,...)
- `args(show)` : (object)
- `args(plot)` : (x,y,...)
- `args(summary)` : (object,...)
- `args(length)` : (x)

C.6 To see objects

- `slotNames("Trajectories")`: gives the name of the slots of the object (but not their type).
- `getSlots("Trajectories")`: gives the slots of the object and their type.
- `getClass("Trajectories")`: gives the slots of the object, their type, the heirs and ancestors.
- `getClass("Trajectories",complete=FALSE)`: gives the slots of the object, son and father.
- `getMethod(f="plot",signature="Trajectories")`: gives the definition of `plot` for the `Trajectories` object (without inheritance).
- `getMethods("plot")`: gives the definition of `plot` for all the possible signatures.
- `existsMethod(f="length",signature="Trajectories")`: returns `True` if the method exists (without inheritance) for the studied class.
- `selectMethod(f="length",signature="Trajectories")`: gives the definition of `length` for the `Trajectories` object (with inheritance).
- `asMethod("length","Trajectories")`: returns `True` if the method exists for `Trajectories` (with inheritance).
- `showMethods("plot")`: prints all the signatures which can be used for the `plot` method.
- `showMethods(classes="Trajectories")`: prints all the methods defined for the `Trajectories` classes.
- `findMethod ("plot", "Trajectories")`: gives the environment in which `plot` for `Trajectories` is defined.

D Further reading

D.1 On R

- **For the complete rookie:** I have not managed to find an R tutorial to start learning R for individuals who are completely unfamiliar with programming.
- **Novice in R but knowing programming:** *An introduction to R* [**?**] (free, available on CRAN).
- **Going deeper in R knowledge:** *S Programming* [**?**] (around 70 Euro) This book is great: very intense and complete.
- **For graphical procedure:** *R Graphics* [**?**] (around 55 Euro)

Other references are available on CRAN `http://cran.r-project.org/`.

D.2 On S4

- *S4 Classes in 15 pages, more or less* [**?**]: good tutorial to start with.
- *S4 Classes and Methods* [**?**]: same as above, simple and clear, an other good tutorial to start.
- *S Programming* [**?**] Although the chapter on S4 is not very long, it is nevertheless very clear, and therefore it is worth mentioning this book.
- *Programming with Data* [**?**]: very hard to understand.

For more information, there is a R wiki dedicated to S4 [**?**].