# A pratical tutorial on S4 programming

Laurent Gatto[*]

Cambridge Center for Proteomics
University of Cambridge

June 21, 2013

[*] lg390@cam.ac.uk

## Contents

## 1 Introduction

This document introduces the `R` object-oriented programming paradigm using the microarray as a use case. The introduction is purely practical and does not aim for an exhaustive guide to `R` object-oriented programming. We will concentrate on the S4 system and only mention the older S3 system and the recent S4 reference class infrastructure. See the appropriate literature, `?ReferenceClasses` or our more thorough introduction to OO programming[1] and references therein for mote details.

In section 2, we present a solution on how to represent microarray data in `R` using simple data types and conclude with some issues with this implementation. In section 3, we introduce fundamental concepts of OO programming and introduce how OO programming is implemented in S4 (and S3) system.

## 2 The microarray example

We assume the reader is familiar with the concept of microarrays and the type of data that is obtained from such an experiment. Before embarking in any serious programming task, in particular when modelling data and defining data structures (using a OO class or not), to carefully think about how to best represent and store the data.

[1] `https://github.com/lgatto/roo`

**Exercise 1:** Based on your understanding of microarrays and the kind of data that is to be used to computational analysis, think of what is going to be needed to describe an experiment and what the types of data structures available in `R` (`data.frame`, `matrix`, `vector`, ...) are most appropriate. Ideally, one would want everything (data, meta-data, ...) to be stored together as a single variables.

There are of course multiple valid solutions to the above question. Below are three pieces of information that consider essential along with their respective `R` data structure.

- We choose to represent the microarray results as a `matrix` of size $n \times m$, where $n$ is the number of probes on the microarray and $m$ is the number of samples. The matrix is named `marray`.
- The sample annotation (meta-data) is described using a `data.frame` with exactly $m$ rows and any number of columns. It is named `pmeta`.
- The feature (probe) annotation (meta-data) is described using a `data.frame` with exactly $n$ rows and any number of columns. Let's call it `fmeta`.

```
> n <- 10
> m <- 6
> marray <- matrix(rnorm(n * m), ncol = m)
> pmeta <- data.frame(sampleId = 1:m,
+                     condition = rep(c("WT", "MUT"), each = 3))
```

We will manually use the same names for intensity matrix columns and the sample meta-data rows as well as the matrix rows and feature meta-data row. Finally, to keep these pieces of information together, they will all be combined into a `list` that will represent our microarray experiment.

```
> rownames(pmeta) <- colnames(marray) <- LETTERS[1:m]
> fmeta <- data.frame(geneId = 1:n,
+                     pathway = sample(LETTERS, n, replace = TRUE))
> rownames(fmeta) <-
+     rownames(marray) <- paste0("probe", 1:n)
> maexp <- list(marray = marray,
+               fmeta = fmeta,
+               pmeta = pmeta)
> rm(marray, fmeta, pmeta)
> str(maexp)
```

```
List of 3
 $ marray: num [1:10, 1:6] -0.626 0.184 -0.836 1.595 0.33 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:10] "probe1" "probe2" "probe3" "probe4" ...
  .. ..$ : chr [1:6] "A" "B" "C" "D" ...
 $ fmeta :'data.frame': 10 obs. of  2 variables:
  ..$ geneId : int [1:10] 1 2 3 4 5 6 7 8 9 10
  ..$ pathway: Factor w/ 8 levels "E","F","L","M",..: 8 4 4 1 7 3 5 2 2 6
 $ pmeta :'data.frame': 6 obs. of  2 variables:
  ..$ sampleId : int [1:6] 1 2 3 4 5 6
  ..$ condition: Factor w/ 2 levels "MUT","WT": 2 2 2 1 1 1
```

We can access the respective elements of our microarray experiment with the `$` operator.

```
> maexp$pmeta

  sampleId condition
A        1        WT
B        2        WT
C        3        WT
D        4       MUT
E        5       MUT
F        6       MUT

> summary(maexp$marray[, "A"])

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
 -0.836  -0.546   0.257   0.132   0.554   1.600 

> wt <- maexp$pmeta[, "condition"] == "WT"
> maexp$marray["probe8", wt]

      A       B       C 
 0.7383  0.9438 -1.4708 

> maexp[["marray"]]["probe3", !wt]

     D      E      F 
0.3877 0.6970 0.3411 
```

```
> boxplot(maexp$marray)
```

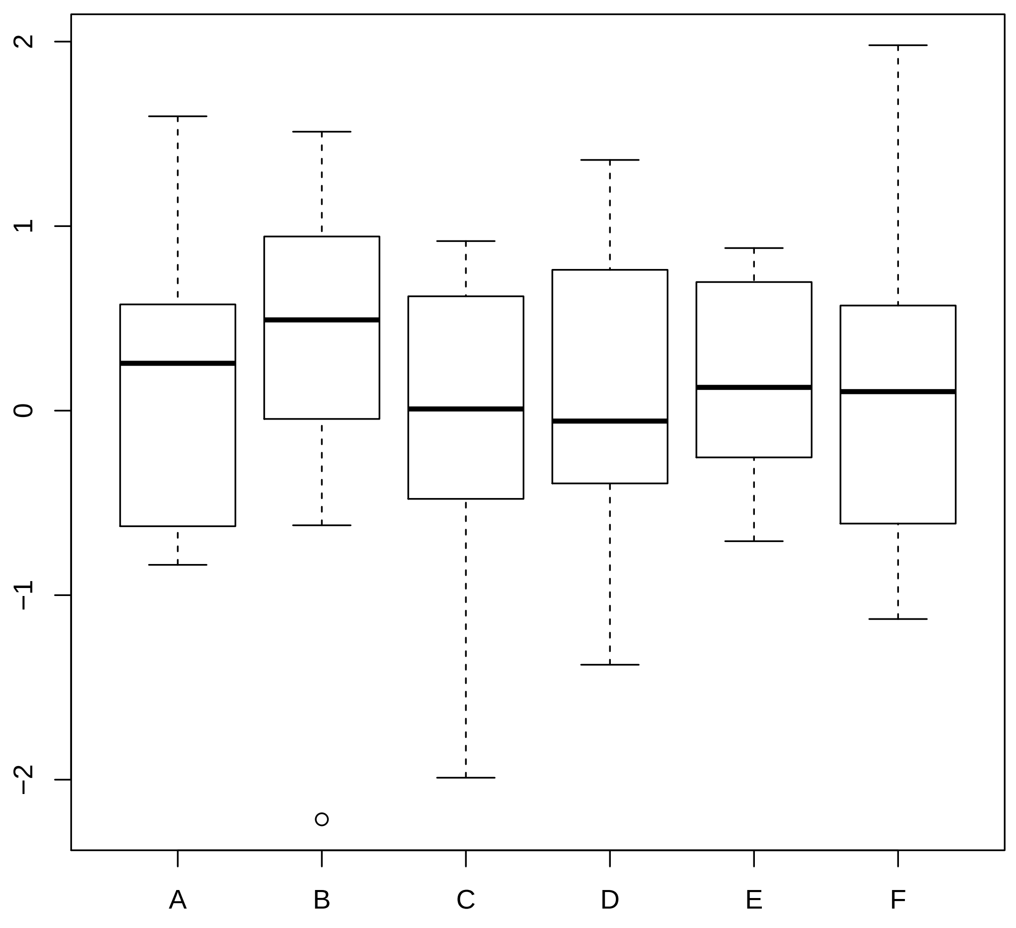

**Figure 1:** Boxplot representing the intensity distributions of the 10 probes for the 6 samples.

**Exercise 2:** But what if we want to subset the experiment. How would we extract the 10 first probes for the 3 first samples?

We have to manually subset the individual elements of our list, making sure that the number of rows of the `marray` and `fmeta` elements remain identical as well as the number of columns of `marray` and the number of columns of `pmeta`.

```
> x <- 1:5
> y <- 1:3
> marray2 <- maexp$marray[x, y]
> fmeta2 <- maexp$fmeta[x, ]
> pmeta2 <- maexp$pmeta[y, ]
> maexp2 <- list(marray = marray2, fmeta = fmeta2, pmeta = pmeta2)
> rm(marray2, fmeta2, pmeta2)
> str(maexp2)

List of 3
 $ marray: num [1:5, 1:3] -0.626 0.184 -0.836 1.595 0.33 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:5] "probe1" "probe2" "probe3" "probe4" ...
  .. ..$ : chr [1:3] "A" "B" "C"
 $ fmeta :'data.frame': 5 obs. of  2 variables:
  ..$ geneId : int [1:5] 1 2 3 4 5
  ..$ pathway: Factor w/ 8 levels "E","F","L","M",..: 8 4 4 1 7
 $ pmeta :'data.frame': 3 obs. of  2 variables:
  ..$ sampleId : int [1:3] 1 2 3
  ..$ condition: Factor w/ 2 levels "MUT","WT": 2 2 2
```

The above solution does not provide a clean syntax. As a user, we have to know the names or positions of the respective elements of the microarray list elements to directly access the parts of interest. Finally, a simple operation like subsetting the microarray experiment is very cumbersome and prone to errors.

## 3 Using OO programming

Object-oriented programming is based on two important concepts, abstraction and encapsulation. We want to represent the microarray concept in a way that makes most sense to the users without distracting them with unnecessary technicalities. These technicalities refer to the underlying implementation. Do the users really need to know that we used a list and that the first element, called `marray` is the matrix? We want the users to comprehend microarrays in `R` like they know them in real life, i.e. manipulate the abstract concept microarray while keeping all the underlying technical details, the implementation, hidden, or encapsulated.

These goals are achieved in two steps. First, we defined a class that represents (abstracts) the concept of a microarray. This is very similar to what we have done with the `list` above (the S3 system does use list), but we will use a more elaborated approach that, although more verbose, provides numerious benefits that will be described in the next sections. The class represents a data container and is defined on its own. An instance of a specific class, that contains data arranged in the specific container, is called an object.

Once we have created a class, we will want to defined a set of specific behaviours, that make sense in the eyes of the users. These behaviours will be implemented by special functions, called methods. Methods are functions that tune their befaviour based on the class of their input. You have already obseved this in your every day usage of `R` : whether we ask to produce the boxplot of a `matrix` (for example `boxplot(maexp[[1]])`) or provide a `data.frame` and a `formula` like `boxplot(sampleId ~ condition, data = maexp[[3]])`, `R` automatically does the right thing.

It now becomes obvious that we have two different kind of roles. The *developer* is the one that creates the class and knows the implementation and the *user* is the one that uses the class without knowing, or needing to know, its actual underlying representation.

## 4 The `MArray` class

We can define a class with the `setClass` function. Our class is defined by a name, `MArray`, and a content. The different elements of an S4 class are called slots[2].

```
> MArray <- setClass("MArray",
+                      slots = c(marray = "matrix",
+                          fmeta = "data.frame",
+                          pmeta = "data.frame"))
```

The `setClass` function returns a special function called a constructor, that can be used to create an instance of the class.

```
> ## an empty object
> MArray()

An object of class "MArray"
Slot "marray":
<0 x 0 matrix>

Slot "fmeta":
data frame with 0 columns and 0 rows

Slot "pmeta":
data frame with 0 columns and 0 rows
```

[2]Note that the usage of `slots` to define the representation of the class is the preferred way to define a class; the `representation` function is deprecated from version 3.0.0 and should be avoided.

```
> ma <- MArray(marray = maexp[[1]],
+              pmeta = maexp[["pmeta"]],
+              fmeta = maexp[["fmeta"]])
> class(ma)

[1] "MArray"
attr(,"package")
[1] ".GlobalEnv"

> ma

An object of class "MArray"
Slot "marray":
              A        B        C        D       E
probe1  -0.6265  1.51178  0.91898  1.35868 -0.1645
probe2   0.1836  0.38984  0.78214 -0.10279 -0.2534
probe3  -0.8356 -0.62124  0.07456  0.38767  0.6970
probe4   1.5953 -2.21470 -1.98935 -0.05381  0.5567
probe5   0.3295  1.12493  0.61983 -1.37706 -0.6888
probe6  -0.8205 -0.04493 -0.05613 -0.41499 -0.7075
probe7   0.4874 -0.01619 -0.15580 -0.39429  0.3646
probe8   0.7383  0.94384 -1.47075 -0.05931  0.7685
probe9   0.5758  0.82122 -0.47815  1.10003 -0.1123
probe10 -0.3054  0.59390  0.41794  0.76318  0.8811
              F
probe1   0.3981
probe2  -0.6120
probe3   0.3411
probe4  -1.1294
probe5   1.4330
probe6   1.9804
probe7  -0.3672
probe8  -1.0441
probe9   0.5697
probe10 -0.1351


Slot "fmeta":
        geneId pathway
probe1       1       Z
probe2       2       M
```

```
probe3       3       M
probe4       4       E
probe5       5       T
probe6       6       L
probe7       7       N
probe8       8       F
probe9       9       F
probe10     10       P

Slot "pmeta":
  sampleId condition
A        1        WT
B        2        WT
C        3        WT
D        4       MUT
E        5       MUT
F        6       MUT
```

To access individual slots, we need to use the `@`. This is equivalent of using the `$` for a list.

```
> ma@pmeta

  sampleId condition
A        1        WT
B        2        WT
C        3        WT
D        4       MUT
E        5       MUT
F        6       MUT
```

But this is something we do not want a user to do. To access a slot like this, one needs to know its name, i.e. the underlying plumbing of the class. This breaks the notion of encapsulation. Instead, the developer will provide the user with specific accessor methods to extract (or update using a replace method) specific slots.

## 5 `MArray` methods

Before proceeding, we need to explain the concept of generic function. A generic function, or generic for short, is a function that dispatches methods to the appropriate class-specific implementation. A method `do` will implement behaviour for a specific class `A`, while another implementation of `do`, will define another behavior for class `B`. The generic `do` is the link between the class and its dedicated implementation. If we have `do(a)` (where `a` is of class `A`), than the generic will make sure that the `A`-specific code of `do` will be executed.

Before we define a method with `setMethod`, we will always want to first check if such a method does not exists (in which case there is already a generic function). If it is the case, we write our new methods. If not, we first create the generic with `setGeneric` and then proceed with the method.

### 5.1 Accessors

### 5.2 The `show` method

### 5.3 The `dim` method

### 5.4 The subsetting operation

### 5.5 The `validity` method

### 5.6 A `replace` method

## Session information

All software and respective versions used to produce this document are listed below.

- R Under development (unstable) (2013-06-16 r62969), `x86_64-unknown-linux-gnu`
- Locale: `LC_CTYPE=en_GB.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_GB.UTF-8`, `LC_COLLATE=en_GB.UTF-8`, `LC_MONETARY=en_GB.UTF-8`, `LC_MESSAGES=en_GB.UTF-8`, `LC_PAPER=C`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_GB.UTF-8`, `LC_IDENTIFICATION=C`
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Other packages: codetools 0.2-8, knitr 1.2
- Loaded via a namespace (and not attached): digest 0.6.3, evaluate 0.4.3, formatR 0.7, stringr 0.6.2, tools 3.1.0