A practical tutorial on S4 programming

Laurent Gatto*

June 23, 2013

Contents

More material is available at https://github.com/lgatto/TeachingMaterial.

*lg390@cam.ac.uk

[1] http://creativecommons.org/licenses/by-sa/3.0/

1 Introduction

This document[2] introduces R object-oriented (OO) programming using microarrays 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 here. See the appropriate literature, `?ReferenceClasses` or our more thorough introduction to OO programming[3] and references therein for mote details.

In section 2, we present a solution on how to represent microarray data in R using basic data types and conclude with some issues and limitations of this implementation. In section 3, we introduce fundamental concepts of OO programming and present how OO programming is implemented in the S4 system. In sections 4 and 5, we implement the S4 class and methods of our microarray example. Section 6 briefly shows how to learn about existing classes and methods.

2 The microarray example

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

Exercise 1: Based on your understanding of microarrays, the nature of data their produce and the kind of computational analysis the data fill undergo, think of what is going to be needed to describe an experiment and what the type(s) of data structure 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 we 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 that stores the intensities (these could also be fold-changes) is named `marray`.

[2] The latest version is available on the github repository https://github.com/lgatto/S4-tutorial.

[3] https://github.com/lgatto/roo

- 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`.

We will also use the same names for the `marray` columns and the `pmeta` rows as well as the `marray` and `fmeta` rows.

```
> n <- 10
> m <- 6
> marray <- matrix(rnorm(n * m, 10, 5), ncol = m)
> pmeta <- data.frame(sampleId = 1:m,
+                     condition = rep(c("WT", "MUT"), each = 3))
> 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)
```

Finally, to link these related pieces of information together, `marray`, `pmeta` and `fmeta` will all be combined into a `list` that will represent our microarray experiment.

```
> maexp <- list(marray = marray,
+               fmeta = fmeta,
+               pmeta = pmeta)
> rm(marray, fmeta, pmeta) ## clean up
> str(maexp)

List of 3
 $ marray: num [1:10, 1:6] 6.87 10.92 5.82 17.98 11.65 ...
  ..- 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 and manipulate 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.
   5.82    7.27   11.30   10.70   12.80   18.00

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

     A      B      C
13.692 14.719  2.646

> maexp[["marray"]]["probe3", !wt]  ## different syntax

    D     E     F
11.94 13.48 11.71
```

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.

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

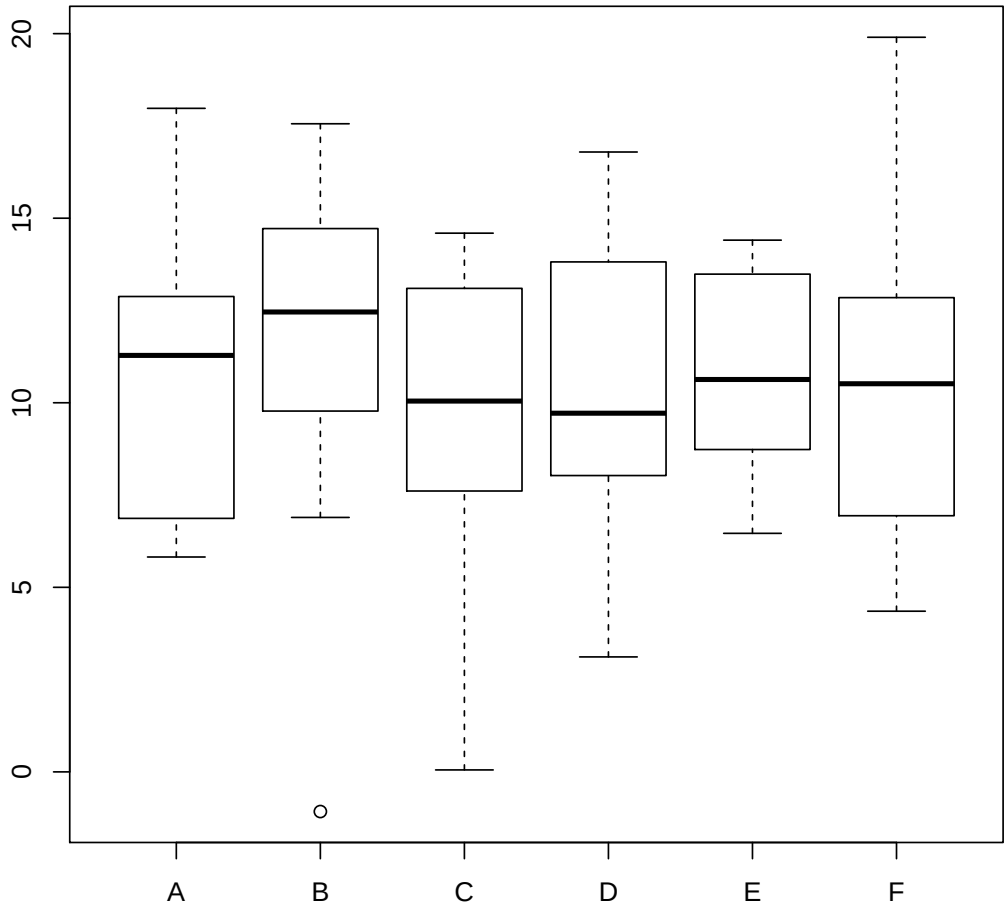

Figure 1: Boxplot representing the intensity distributions of the 10 probes for the 6 samples.

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)  ## clean up
> str(maexp2)

List of 3
 $ marray: num [1:5, 1:3] 6.87 10.92 5.82 17.98 11.65 ...
  ..- 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
```

A simple operation like sub-setting the microarray experiment is very cumbersome and prone to errors. If we were to use this implementation for further work, we would of course want to write a custom function to perform the above.

3 Object-oriented 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 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 lists), but we will use a more elaborated approach that, although more verbose, provides numerous 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 define 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 behaviour based on the class of their input. You have already observed 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.

From the above, it transpires that we have now two different kind of roles. The *developer* is the person who creates the class and knows the implementation while the *user* is the one who uses the class without knowing, or needing to know, the 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 structure. The different elements/fields of an S4 class are called slots[4]. When defining the slots, we provide their respective names and classes as a (named) `vector` or `list`. It will only be possible to create objects with exactly these types of slots.

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

[4]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.

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

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

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

> MArray(marray = 1:2) ## not allowed

Error:  invalid class "MArray" object:  invalid object for slot "marray"
in class "MArray":  got class "integer", should be or extend class "matrix"

> 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      F
probe1   6.868 17.559 14.59489 16.793  9.177 11.991
probe2  10.918 11.949 13.91068  9.486  8.733  6.940
probe3   5.822  6.894 10.37282 11.938 13.485 11.706
probe4  17.976 -1.073  0.05324  9.731 12.783  4.353
probe5  11.648 15.625 13.09913  3.115  6.556 17.165
probe6   5.898  9.775  9.71936  7.925  6.463 19.902
probe7  12.437  9.919  9.22102  8.029 11.823  8.164
probe8  13.692 14.719  2.64624  9.703 13.843  4.779
```

```
probe9  12.879 14.106  7.60925 15.500  9.438 12.849
probe10  8.473 12.970 12.08971 13.816 14.406  9.325

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 to 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 (see section 5.2) to extract (or update using a replace method, section 5.5) 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 their appropriate class-specific implementation. A method `do` will implement behaviour for a specific class `A`, while another implementation of `do`, will define another behaviour 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), as illustrated with the `show` method in section 5.1. If it is the case, we write our new methods. If not, we first create the generic and then proceed with the method.

5.1 The `show` method

The `show` method (it is a method, as it exhibits custom behaviour depending on the class of its argument) is a very helpful one. It allows to define custom summary view of an object when we type its name in the console, instead of having all its (possibly very long content) displayed.

```
> show

standardGeneric for "show" defined from package "methods"

function (object)
standardGeneric("show")
<bytecode: 0x2681ec0>
<environment: 0x32787f0>
Methods may be defined for arguments: object
Use  showMethods("show")  for currently available ones.
(This generic function excludes non-simple inheritance; see ?setIs)

> isGeneric("show")

[1] TRUE
```

```
> hasMethod("show")

[1] TRUE
```

As there is already a `show` generic function, we can immediately proceed with the method definition using the `setMethod` function . To do so we need a few things. First, we need to know for what class we implement the specific `show` method; this is the `MArray` class and will be passed as the `signature` argument in `setMethod`. We also need to know the argument names that are defined in the generic. These must match exactly, as we write a method for that specific generic. The arguments can be found by just typing the name of the generic (as in the previous) code chunk, look at its documentation or directly ask for the arguments with `args(show)`. We see that there is only one argument, `object` (naming the first argument of a generic `object` is a widely applied convention). This is the same name that we will have to use when writing the `definition` of our method.

```
> setMethod("show",
+           signature = "MArray",
+           definition = function(object) {
+               cat("An object of class ", class(object), "\n", sep = "")
+               cat(" ", nrow(object@marray), " features by ",
+                   ncol(object@marray), " samples.\n", sep = "")
+               invisible(NULL)
+           })

[1] "show"

> ma

An object of class MArray
 10 features by 6 samples.
```

5.2 Accessors

As mentioned above, we want to provide customised and controlled access to the class slots. This does not prevent us, as developers, to use the `@` accessor, but does not force others to know the implementation details.

Let's create an accessor for the `marray` slot and call the accessor `marray`. There is no harm in naming the slot and its accessor with the same name but there is no constrain in doing so. There is no such method or generic; just typing `marray` with

tell you that no such object is found. Below, we create a new generic function with `setGeneric`. We define the name of our new generic as well as the name of the argument(s) that will have to be re-used when defining class-specific method.

```
> setGeneric("marray", function(object) standardGeneric("marray"))

[1] "marray"
```

In general, it is considered good practice to add a `...` in the signature of a generic function. It provides the flexibility for other methods to use more arguments.

```
> setGeneric("marray", function(object, ...) standardGeneric("marray"))

[1] "marray"
```

We now proceed in the same way as above, using `setMethod`. The definition of our method (i.e. the actual code that will be executed) is very short and of course uses `@` to access (and return) the slot content.

```
> setMethod("marray", "MArray",
+           function(object) object@marray)

[1] "marray"

> marray(ma)

              A      B        C      D      E      F
probe1    6.868 17.559 14.59489 16.793  9.177 11.991
probe2   10.918 11.949 13.91068  9.486  8.733  6.940
probe3    5.822  6.894 10.37282 11.938 13.485 11.706
probe4   17.976 -1.073  0.05324  9.731 12.783  4.353
probe5   11.648 15.625 13.09913  3.115  6.556 17.165
probe6    5.898  9.775  9.71936  7.925  6.463 19.902
probe7   12.437  9.919  9.22102  8.029 11.823  8.164
probe8   13.692 14.719  2.64624  9.703 13.843  4.779
probe9   12.879 14.106  7.60925 15.500  9.438 12.849
probe10   8.473 12.970 12.08971 13.816 14.406  9.325
```

If we change the underlying implementation by changing the name of the slot or using an environment instead of a matrix, the `ma@marray` is going to break. However, when providing accessors, we can echo the changes in the accessor implementation without affecting the users' behaviour or existing scripts.

Exercise 3: Implement the `fmeta` and `pmeta` accessors.

5.3 The sub-setting operation

Let's now encapsulate the sub-setting of an `MArray` object in a proper method to facilitate this simple operation. In `R` , the default subsetting operator is `[`. Although its syntax looks like it is special, the operator is just a normal function with a bit of extra syntactic sugar.

```
> letters[1:3]

[1] "a" "b" "c"

> `[`(letters, 1:3)

[1] "a" "b" "c"
```

If you type ‘`[`‘ in you `R` console, you will see that this is a primitive function. These internally implemented functions have a special property that, although not explicitly generic functions, they get automatically promoted to generics when a method of the same name is defined. In other words, we we must not create a generic (this would break `[` is all the other cases) and can directly proceed with implementing a specific behaviour of `[` for the `MArray` class.

The documentation `help("[")` shows that, in addition to `x`, the object to be subset, we also have to take the `i` and `j` indices into account and the `drop` argument. When an argument is not relevant, we specify this by declaring that it is `"missing"`.

```
> setMethod("[", "MArray",
+           function(x,i,j,drop="missing") {
+               .marray <- x@marray[i, j]
+               .pmeta <- x@pmeta[j, ]
+               .fmeta <- x@fmeta[i, ]
+               MArray(marray = .marray,
+                      fmeta = .fmeta,
+                      pmeta = .pmeta)
+           })

[1] "["

> ma[1:5, 1:3]

An object of class MArray
 5 features by 3 samples.
```

5.4 The `validity` method

While discussing the design of our microarray data structure in section 2, we have implicitly stated the following validity constrains, schematically represented in figure 2. In terms of dimensions, the number of rows of the expression matrix must be equal to the number of rows in the feature meta-data data frame and the number of columns in the expression matrix must be equal to the number of rows in the sample meta-data data frame. In terms of names, we have also implied that the row names of the expression matrix and feature meta-data data frame were identical and that the column names of the expression matrix and the row names of the sample meta-data data frame were identical. The latter is a good check to make sure that the order in these respective data structures are the same.

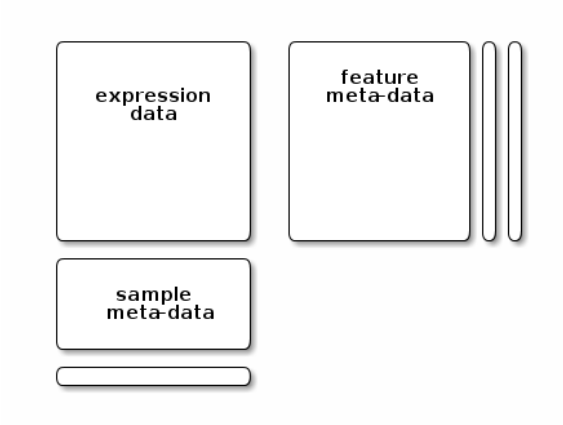

Figure 2: Dimension requirements for the respective expression, feature and sample meta-data slots. (Figure from the pRoloc package vignette.)

It is possible to create a validity method for S4 classes to check that the assumptions about the data are met. This validity method is created using the `setValidity` function and the validity of an object can be checked with `validObject`.

```
> setValidity("MArray", function(object) {
+     msg <- NULL
+     valid <- TRUE
+     if (nrow(marray(object)) != nrow(fmeta(object))) {
+         valid <- FALSE
+         msg <- c(msg,
+                  "Number of data and feature meta-data rows must be identical.")
+     }
+     if (ncol(marray(object)) != nrow(pmeta(object))) {
+         valid <- FALSE
+         msg <- c(msg,
```

```
+                 "Number of data rows and sample meta-data columns must be identi
+     }
+     if (!identical(rownames(marray(object)), rownames(fmeta(object)))) {
+         valid <- FALSE
+         msg <- c(msg,
+                  "Data and feature meta-data row names must be identical.")
+     }
+     if (!identical(colnames(marray(object)), rownames(pmeta(object)))) {
+         valid <- FALSE
+         msg <- c(msg,
+                  "Data row names and sample meta-data columns names must be ident
+     }
+     if (valid) TRUE else msg
+ })

Class "MArray" [in ".GlobalEnv"]

Slots:

Name:      marray      fmeta      pmeta
Class:     matrix data.frame data.frame

> validObject(ma)

[1] TRUE
```

Exercise 4: Try to create a new **invalid** `MArray` object using the constructor `MArray`.

```
> x <- matrix(1:12, ncol = 3)
> y <- fmeta(ma)
> z <- pmeta(ma)
> MArray(marray = x, fmeta = y, pmeta = z)

Error:  invalid class "MArray" object:  1:  Number of data and feature
meta-data rows must be identical.
invalid class "MArray" object:  2:  Number of data rows and sample meta-data
columns must be identical.
invalid class "MArray" object:  3:  Data and feature meta-data row names
must be identical.
```

```
invalid class "MArray" object:  4:  Data row names and sample meta-data
columns names must be identical.
```

5.5 A replacement method

The following section describes how to write a method that is dedicated to the replacement or update of the content of slots. It is of course possible to perform such an operation by accessing the slot content directly, as illustrated below. As discussed in previous sections, this is not advised as it violates the encapsulation of our data and makes it possible to break the validity of an object. Note also that it is not possible to overwrite any slot with data that is not of the expected class.

```
> ma@marray <- 1

Error:  assignment of an object of class "numeric" is not valid for @'marray'
in an object of class "MArray"; is(value, "matrix") is not TRUE

> (broken <- ma)

An object of class MArray
 10 features by 6 samples.

> broken@marray <- matrix(1:9, 3)
> broken

An object of class MArray
 3 features by 3 samples.

> validObject(broken)

Error:  invalid class "MArray" object:  1:  Number of data and feature
meta-data rows must be identical.
invalid class "MArray" object:  2:  Number of data rows and sample meta-data
columns must be identical.
invalid class "MArray" object:  3:  Data and feature meta-data row names
must be identical.
invalid class "MArray" object:  4:  Data row names and sample meta-data
columns names must be identical.
```

There is a special type of method, called a *replacement method*, that can be implemented to obtain the desired behaviour in a clean and controlled way. A replacement method provides the convenient `slot(object)<-` syntax.

Replacement method are always named by concatenating the name of the method and the arrow assignment operator. If we wish to write a method to replace the slot that can be accessed with the `marray` accessor (again, the slot itself is called `marray`, but that does not need to be the case), the corresponding replacement method would be called `marray<-`[5]. Another important specificity of replacement methods is that they always take (at least) two arguments; the object to be updated, that we will name `object` and the replacement data, always called `value`. Finally, as `marray<-` is going to be a method (and there is no existing generic), we first need to define a generics.

```
> setGeneric("marray<-",
+            function(object, value) standardGeneric("marray<-"))

[1] "marray<-"
```

In the definition of the replacement method, we check that the user-provided `value` does not break the validity of `object` with the `validObject` method (see section 5.4) before returning it.

```
> setMethod("marray<-", "MArray",
+           function(object, value) {
+               object@marray <- value
+               if (validObject(object))
+                   return(object)
+ })

[1] "marray<-"
```

Below, we firs try to replace the expression matrix with an invalid value and then test out new replacement method with a valid matrix.

```
> tmp <- matrix(rnorm(n*m, 10, 5), ncol = m)
> marray(ma) <- tmp

Error:  invalid class "MArray" object:  1:  Data and feature meta-data
row names must be identical.
invalid class "MArray" object:  2:  Data row names and sample meta-data
columns names must be identical.

> colnames(tmp) <- LETTERS[1:m]
```

[5]It could actually be called anything followed by `<-`, but that would be confusing for the user.

```
> rownames(tmp) <- paste0("probe", 1:n)
> head(marray(ma), n = 2)

            A     B     C      D     E     F
probe1  6.868 17.56 14.59 16.793 9.177 11.99
probe2 10.918 11.95 13.91  9.486 8.733  6.94

> marray(ma) <- tmp
> head(marray(ma), n = 2)

             A      B     C      D     E     F
probe1 10.9440 11.457 11.66 12.792 18.84 8.036
probe2  0.9752  7.784 15.32  3.617 13.58 8.400
```

Exercise 5: Implement the `fmeta` and `pmeta` replacement methods and show that it works with the following replacement.

```
> pmeta(ma)$sex <- rep(c("M", "F"), 3)
> pmeta(ma)

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

6 Introspection

To find out more about a class you are using without reading its source code, one can use the following functions to get the slot names and the complete class definition.

```
> slotNames(ma)

[1] "marray" "fmeta"  "pmeta"

> getClass("MArray")
```

```
Class "MArray" [in ".GlobalEnv"]

Slots:

Name:      marray      fmeta      pmeta
Class:     matrix data.frame data.frame
```

To obtain all the methods that are available for a given function name of for a given class class, one can use `showMethods`.

```
> showMethods("marray")

Function: marray (package .GlobalEnv)
object="MArray"

> showMethods(classes = "MArray")

Function: [ (package base)
x="MArray"

Function: fmeta (package .GlobalEnv)
object="MArray"

Function: fmeta<- (package .GlobalEnv)
object="MArray"

Function: initialize (package methods)
.Object="MArray"
    (inherited from: .Object="ANY")

Function: marray (package .GlobalEnv)
object="MArray"

Function: marray<- (package .GlobalEnv)
object="MArray"

Function: pmeta (package .GlobalEnv)
object="MArray"

Function: pmeta<- (package .GlobalEnv)
```

```
object="MArray"

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

To obtain the code for a specific method, one can use `getMethod` with the name of the method and the name of the class.

```
> getMethod("marray", "MArray")

Method Definition:

function (object, ...)
{
    .local <- function (object)
    object@marray
    .local(object, ...)
}

Signatures:
        object
target  "MArray"
defined "MArray"
```

7 Conclusion

The Bioconductor project provides S4 implementations for microarray data. As a conclusion to our exercise, let's use the class introspection tools seen in section 6 to study the `ExpressionSet` implementation available in the Biobase package.

```
> library("Biobase")
> getClass("ExpressionSet")

Class "ExpressionSet" [package "Biobase"]

Slots:

Name:      experimentData          assayData
Class:              MIAME          AssayData
```

```
Name:           phenoData        featureData
Class: AnnotatedDataFrame AnnotatedDataFrame

Name:          annotation       protocolData
Class:          character AnnotatedDataFrame

Name:   .__classVersion__
Class:           Versions

Extends:
Class "eSet", directly
Class "VersionedBiobase", by class "eSet", distance 2
Class "Versioned", by class "eSet", distance 3
```

There are of course many more slots, to support description of the experiment itself as well as the microarray platform. The expression data itself is stored in the `assayData` slot that is of class `AssayData`. In practice, this generally equates to an environment that contains one or multiple expression matrices. The feature and sample annotations are stored in the `featureData` and `phenoData` slots, both of class `AnnotatedDataFrame`. An `AnnotatedDataFrame` is a `data.frame` that supports additional variable names annotation. Each of these classes can in turn be inspected with `getClass` or, better, by reading the respective documentation.

We also see that the `ExpressionSet` class extends the `eSet` class, i.e. `ExpressionSet` is a sub-class of the `eSet` class. See the `contains` field in `?setClass` to read more about class sub/super-class extensions.

Although the verbosity of the S4 system might seem like a little overhead in the beginning, it provides improved stability and usability for the future. The design and usage of an efficient class system requires one to think about the needs of the user role before writing code, as it involves some commitment in the design decisions and the resulting interface.

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, parallel, stats, utils
- Other packages: Biobase 2.21.4, BiocGenerics 0.7.2, 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