Introduction to object oriented programming in R, with special emphasis on the ExpressionSet class

Héctor Corrada Bravo
based on slides developed by
Jim Bullard, Kasper Hansen and Margaret Taub

PASI, Guanajuato, México
May 3-4, 2010

OOP

- Object oriented programming (OOP) is a popular programming paradigm. Object oriented programming allows us to construct modular pieces of code which can be utilized as building blocks for large systems.
- R is a functional language, not particular object oriented, but support exists for programming in an object oriented style.
- The Bioconductor project uses OOP extensively, and it is important to understand basic features to work effectively with Bioconductor.
- R has two different OOP systems, known as S3 and S4. These two systems are quite different, with S4 being more object oriented, but sometimes harder to work with.
- In both systems, the object oriented system is much more method-centric than languages like Java and Python - R's system is very Lisp-like.

Why?

As a (Bioconductor) user, it is important to have an understanding of S3 and S4.

- In order to understand and use a package unfamiliar to you.
- In order to diagnose and fix when things break (as they tend to do).

Pay close attention to how to get help, how to examine the definition of a class and a method, and how to examine the code.

S3 Classes

First we will take a look at S3 classes. Base `R` uses S3 more or less exclusively.

- "The greatest use of object oriented programming in R is through print methods, summary methods and plot methods. These methods allow us to have one generic function call, plot say, that dispatches on the type of its argument and calls a plotting function that is specific to the data supplied." – R Manual (referring to the S3 system).
- An S3 class is (most often) a `list` with a `class` attribute. It is constructed by the following code `class(obj) <- "class.name"`.

S3 Classes

```
> xx <- rnorm(1000)
> class(xx)
> plot(xx)
> yy <- ecdf(xx)
> class(yy)
> plot(yy)
> plot
> plot.ecdf
> plot.default
> methods("plot")
> getS3method("plot", "histogram")
```

What plot does, depends on the `class` of the `x` argument. It is a method. `plot.ecdf` is the `ecdf` method for `plot`.

Constructing a new S3 Class

```
> jim <- list(height = 2.54 * 12 * 6/100, weight = 180/2.2,
+     name = "James")
> class(jim) <- "person"
> class(jim)
```

We have now made an object of class `person`. We now define a print method.

```
> print(jim)
> print.person <- function(x, ...) {
+     cat("name:", x$name, "\n")
+     cat("height:", x$height, "meters", "\n")
+     cat("weight:", x$weight, "kilograms", "\n")
+ }
> print(jim)
```

Note the method/class has the "dot" naming convention of `method.class`.

S3 classes are not robust

```
> fit <- lm(rnorm(100) ~ 1)
> class(fit)
> print(fit)
> class(fit) <- "something"
> print(fit)
> class(fit) <- "person"
> print(fit)
```

In case print does not have a method for the class, it dispatches to the default method, print.default.
S3 does not have the concept of type checking – there is no way to formally define a class and ensure that the object conform to the definition.

S3 classes and the help system

S3 classes are traditionally documented in the help page for the function that creates them. Example: `lm`.
Methods have a generic help page (often not very informative), sometimes with more specific help under `?method.class`. Example: `plot.lm`.

Inheritance in S3

In S3, inheritance is achieved by the class attribute being a vector. A canonical example is

```
> fit <- glm(rpois(100, lambda = 1) ~
+     1, family = "poisson")
> class(fit)
> methods("residuals")
> methods("model.matrix")
```

If no method for the first is found, the second class is checked.

Useful S3 Method Functions

- `methods("print")` and `methods(class = "lm")`
- `getS3method("print","person")` : Gets the appropriate method associated with a class, useful to see how a method is implemented. Try: `getS3method("residuals", "lm")`.
- In emacs using ESS or in the R Gui we can use TAB completion to determine what methods are available. This can be quite useful for getting help on the specific method (we will see more of this later). In TextMate use ctrl-shift-H.
- Sometimes, methods are non-visible, because they are hidden in a namespace. Use getS3method or getAnywhere to get around this.

```
> residuals.HoltWinters
> getS3method("residuals.HoltWinters")
> getAnywhere("residuals.HoltWinters")
```

Replacement Methods

- As we have already seen R has a somewhat strange type of function that allows us to modify objects in place.
- It is uncommon to define new replacement functions, however they are used quite frequently in day to day programming of R.
- Two examples are: names and colnames. Type "colnames" into the R window and hit "tab", notice the function "colnames<-"?

```
> a <- matrix(1:16, nrow = 4, ncol = 4)
> colnames(a) <- paste("V", 1:4,
+     sep = ".")
> colnames(a)
> point <- list(x = 1, y = 2)
> x.val <- function(x, value) {
+     x$x <- value
+ }
> "x.val<-" <- function(x, value) {
+     x$x <- value
+     return(x)
```


Replacement Methods

```
+ }
> x.val(point, 10)
> print(point)
> x.val(point) <- 10
> print(point)
```

What does the first print statement print? What about the second?

S4 classes, why?

- Although S3 classes can be quite useful and powerful and fast they do not facilitate the type of modularization and type safety that a true object oriented system intends.
- S4 classes are more a traditional object oriented system with type checking, multiple-dispatch, and inheritance.
- S4 is implemented in the methods package in base R.
- For thorough information on S4, read Chambers (1998) "Programming with data" (also known as the green book) (first chapter available at `http://www.omegahat.org/RSMethods/Intro.pdf`) or Chambers (2008) "Software for Data Analysis: Programming with R".
- There are also several good, short, tutorials on the net.

Defining an S4 class

```
> myRep <- representation(height = "numeric",
+     weight = "numeric", name = "character")
> setClass("personS4", representation = myRep)
> getClass("personS4")
> jimS4 <- new("personS4")
> jimS4
> jimS4 <- new("personS4", height = 2.54 *
+     12 * 6/100, weight = 180/2.2,
+     name = "James")
> jimS4
> jimS4@name
> validObject(jimS4)
> jimS4@height <- "2"
```

Notes on the S4 class example

- It is rare for *users* to define their own S4 classes.
- The use of new to *instantiate* a new member of the class is not always needed, often there are explicit constructor functions (see later).
- The use of `@` to access the class *slots* is heavily discouraged, instead use *accessor* functions (see later).

Defining the print method

For completion, we define the print method for `personS4`. For S4 classes, it is not print, but rather show.

```
> setMethod("show", signature("personS4"),
+     function(object) {
+         cat("name:", object@name,
+             "\n")
+         cat("height:", object@height,
+             "meters", "\n")
+         cat("weight:", object@weight,
+             "kilograms", "\n")
+     })
> jimS4
> getMethod("show", signature("personS4"))
```

S4 Generics

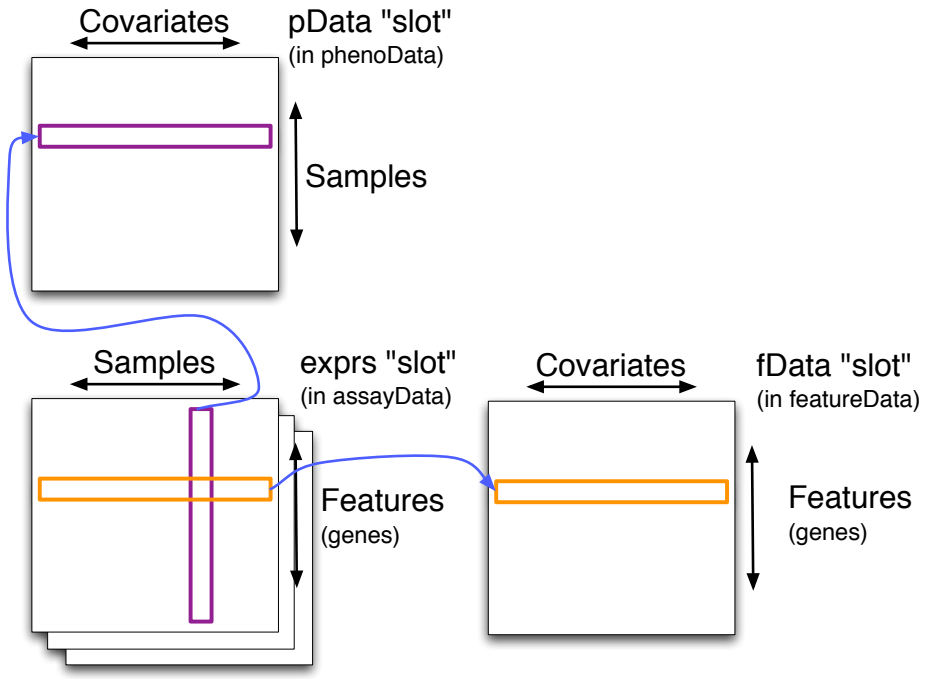
In order to make a new generic we need to call the function setGeneric.

```
> setGeneric("BMI", function(object) standardGeneric("BMI"))
> setMethod("BMI", "personS4", function(object) {
+     object@weight/object@height^2
+ })
> BMI(jimS4)
```


ExpressionSet

In order to get better real-life examples, we will now deconstruct the ExpressionSet from the package Biobase. This is a very important – and complicated – class from Bioconductor. It will be very profitable to feel comfortable with this class, which is also an excellent example of the power of S4 (and sometimes the frustration of S4).
Some history: the ExpressionSet class is a new design, expanding the older (deprecated) exprSet class (which you still see referenced). There is a fair amount of historical baggage associated with this package.

ExpressionSet: Basic idea



Exploring Biobase

Loading

```
> require(Biobase)
> library(help = Biobase)
> getClass("ExpressionSet")
> data(sample.ExpressionSet)
> sample.ExpressionSet
> head(exprs(sample.ExpressionSet))
> head(pData(sample.ExpressionSet))
> head(fData(sample.ExpressionSet))
```

phenoData / AnnotatedDataFrame

An AnnotatedDataFrame is essentially a versioned data.frame with some descriptive labels of the columns.

```
> getClass("AnnotatedDataFrame")
> sample.phenoData <- phenoData(sample.ExpressionSet)
> sample.phenoData
> pData(sample.phenoData)
> varLabels(sample.phenoData)
> sampleNames(sample.phenoData)
> sample.phenoData$type
```

(Note the last one).
This also works directly from the ExpressionSet:

```
> pData(sample.ExpressionSet)
> varLabels(sample.ExpressionSet)
> sampleNames(sample.ExpressionSet)
> sample.ExpressionSet$type
> featureNames(sample.ExpressionSet)
```


eSet and ExpressionSet

The ExpressionSet class is derived from eSet. The main difference between these two classes is that an ExpressionSet provides an exprs method which accesses the expression matrix.

```
> hasMethod("exprs", "eSet")
> hasMethod("exprs", "ExpressionSet")
> getMethod("exprs", "ExpressionSet")
```

exprs is an example of an *accessor* function.

Accessing the ExpressionSet

- Accessing the relevant data involves calling accessor functions. We should try to avoid ever accessing the data directly with the "@" accessor because it is less future-proof. Unlike many object oriented programming languages R does not provide a mechanism for protecting data, such as "private" member variables in many languages.
- Have a look at `?ExpressionSet` to see what other methods are available.

```
> featureNames(sample.ExpressionSet)
> sampleNames(sample.ExpressionSet)
> exprs(sample.ExpressionSet)
> slot(sample.ExpressionSet, "exprs")
> assayData(sample.ExpressionSet,
+     "exprs")
```


ExpressionSet 2

An ExpressionSet contains information

- About characteristics of the samples (`phenoData` / `pData`).
- About gene-level measurements (`assayData` / `exprs`).
- About the microarray (`featureData` / `fData`) (rarely used).

All linked together appropriately. Linking allows for easy subsetting.
The expression matrix has dimension $N_{\text{features}} \times N_{\text{arrays}}$

Subsetting ExpressionSets

We can subset the ExpressionSet object just as we can subset a matrix. Columns refer to samples and rows refer to features.

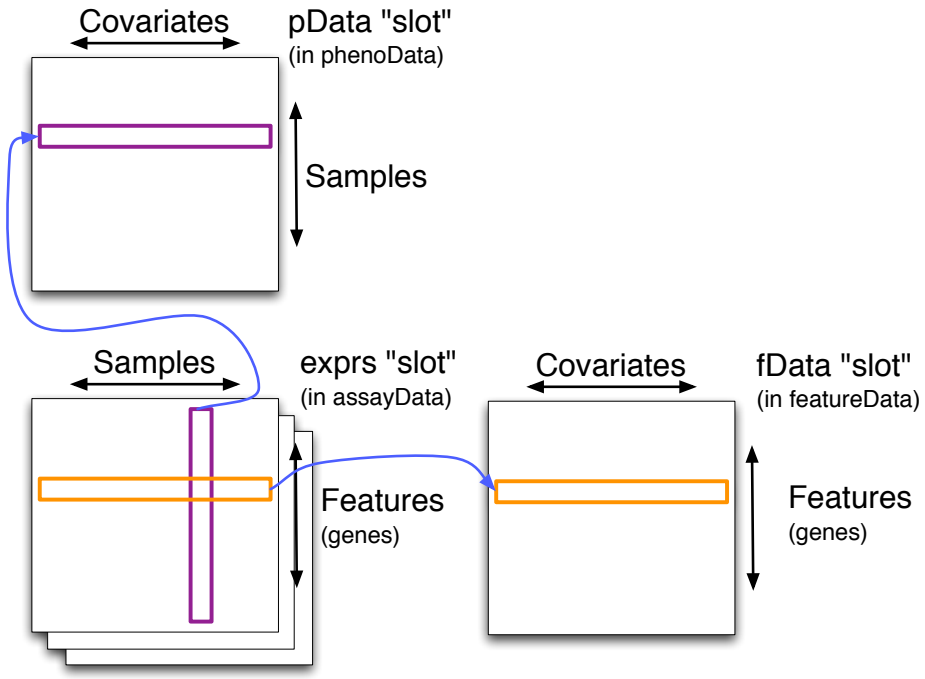
```
> Type <- phenoData(sample.ExpressionSet)$type
> cases <- grep("Case", Type)
> controls <- grep("Control", Type)
> casesEx <- sample.ExpressionSet[,
+     cases]
> controlsEx <- sample.ExpressionSet[,
+     controls]
```

What is the class of casesEx and controlsEx?

```
> sample.ExpressionSet[sample(nrow(sample.ExpressionSet),
+     size = 10, replace = FALSE),
+     5:10]
```

Subsetting is used *a lot*!
Remember to read the "ExpressionSet Introduction" vignette in Biobase.

ExpressionSet: again

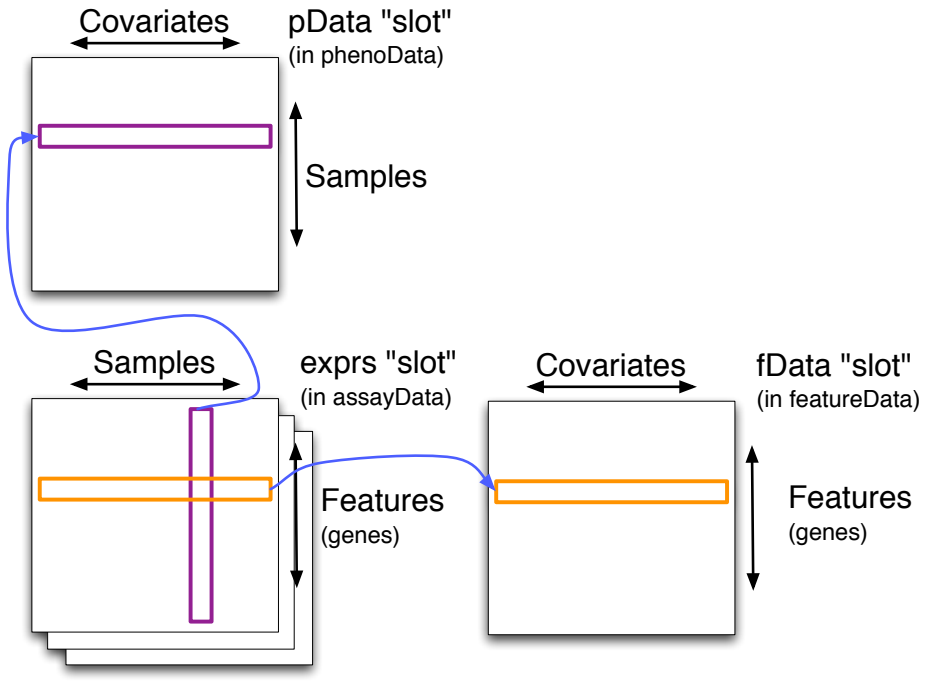


Debugging

Use trace:

```
> trace("show", signature("ExpressionSet"),
+     browser)
> sample.ExpressionSet
> untrace("show")
```


ExpressionSet: again



Exercise: deconstructing AffyBatch

Load the affy package and examine the AffyBatch class. What slots does it have? What do they contain. What does the nrow method return?