Classes and Methods in R

Biostatistics 140.776

Classes and Methods

- A system for doing object oriented programming
- R is rare because it is both interactive *and* has a system for object orientation.
  - Other languages which support OOP: C++, Java, Lisp, Python, Perl
- In R much of the code for supporting classes/methods is written by John Chambers himself.
  - Chambers, J. M. (1998) *Programming with Data: A Guide to the S Language*, Springer.

Two styles of classes and methods

S3 classes/methods

- Included with version 3 of the S language.
- Informal, a little kludgey
- Sometimes called *old-style* classes/methods

S4 classes/methods

- more formal and rigorous
- Included with S-PLUS 6, R 1.4.0
- Also called *new-style* classes/methods

Two worlds living side by side

- For now (and the forseeable future), S3 classes/methods and S4 classes/methods are separate systems.
- Each system can be used fairly independently of the other.
- Developers of new projects (you!) are encouraged to use the S4 style classes/methods.
  - Used extensively in the Bioconductor project
- But many developers still use S3 classes/methods because they are "quick and dirty".
- Oh well....

Object Oriented Programming in R

- A *class* is a description of an thing. A class can be dened using `setClass()`.
- An *object* is an instance of a class. Objects can be created using `new()`.
- A generic function is an R function which dispatches methods. A generic function typically encapsulates a "generic" concept (e.g. `plot`, `mean`, `logLik`, `residuals`, `predict`, ...)
- The generic function does not actually do any computation.
- A *method* is the implementation of a generic function for an object of a particular class.

Things to look up

- The help files for the 'methods' package are extensive — do read them.
- Check out `?setClass`, `?setMethod`, `?setGeneric`, `?Methods`
- Some of it gets technical, but try your best for now—it will make sense in the future as you keep using it.

Classes

All objects in R have a class which can be determined by the class function

```
> class(1)
[1] "numeric"
> class(TRUE)
[1] "logical"
> class(rnorm(100))
[1] "numeric"
> class(NA)
[1] "logical"
> class("asdf")
[1] "character"
>
```

Classes (cont'd)

```
> x <- rnorm(100)
> y <- x + rnorm(100)
> fit <- lm(y ~ x)
> class(fit)
[1] "lm"
>
```

Generics/Methods in R

- S4 and S3 style generic functions look different but conceptually, they are the same (they play the same role).
- When you program you can write new methods for an existing generic OR create your own generics and associated methods.

An S3 generic function (in the 'base' package)

```
> mean
function (x, ...)
UseMethod("mean")
<environment: namespace:base>
```

S3 methods

```
> methods("mean")
[1] mean.Date        mean.POSIXct
[3] mean.POSIXlt     mean.data.frame
[5] mean.default     mean.difftime
>
```

An S4 generic function (from the 'methods' package)

```
> show
standardGeneric for "show" defined from package "methods"

function (object)
standardGeneric("show")
<environment: 0x1913bc8>
Methods may be defined for arguments: object
Use  showMethods("show")  for currently available ones.
>
```

S4 methods

```
> showMethods("show")
Function: show (package methods)
object="ANY"
object="MethodDefinition"
object="MethodWithNext"
object="ObjectsWithPackage"
object="classRepresentation"
object="function"
    (inherited from: object="ANY")
object="genericFunction"
object="signature"
object="standardGeneric"
    (inherited from: object="genericFunction")
object="traceable"
```

Generic/method mechanism

The first argument of a generic function is an object of a particular class (there may be other arguments)

1. The generic function checks the class of the object.
2. A search is done to see if there is an appropriate method for that class.
3. If there exists a method for that class, then that method is called on the object and we're done.
4. If a method for that class does not exist, a search is done to see if there is a default method for the generic. If a default exists, then the default method is called.
5. If a default method doesn't exist, then an error is thrown.

Example 1

```
> x <- rnorm(100)
> mean(x)
[1] -0.06846675
```

1. The class of x is "numeric"
2. But there is no mean method for "numeric" objects!
3. So we call the default function `mean.default`.

```
> mean.default
function (x, trim = 0, na.rm = FALSE, ...)
{
        ## ... Skip 18 lines ...
        .Internal(mean(x))
}
<environment: namespace:base>
>
```

Example 2

```
> df <- data.frame(x = rnorm(100), y = rnorm(100, 1))
> mean(df)
x y
0.002565053 0.972148319
```

1. The class of df is "data.frame".
2. There is a method for "data.frame" objects!
3. We call `mean.data.frame` on `df`.

```
> mean.data.frame
function (x, ...)
sapply(x, mean, ...)
<environment: namespace:base>
>
```

NOTE: Generally, you should not call methods directly. Rather, use the generic function and let the method be dispatched automatically.

Write your own methods!

If you write new methods for new classes, you'll probably end up writing methods for the following generics:

- print/show
- summary
- plot

You could write a new method for an existing class, but more likely you'll want to write a method for a class that you create.

Why would you want to create a new class?

- To represent new types of data (e.g. gene expression, space-time, hierarchical, sparse matrices)
- New concepts/ideas (e.g. a fitted point process model, mixed-effects models)
- To abstract implementation details from the user

I say things are "new" meaning that R does not know about them (not that they are new to the statistical community).

Example: A Sparse Matrix

```
# Sparse general matrix in triplet format
setClass("tripletMatrix",
         representation(i = "integer",
                        j = "integer",
                        x = "numeric",
                        Dim = "integer"))
setMethod("crossprod",
          signature(x = "tripletMatrix",
                    y = "tripletMatrix"),
          ## code for cross products
          )
```

Example: A polygon class

```
setClass("polygon",
         representation(x = "numeric",
                        y = "numeric"))
setMethod("plot", "polygon",
          function(x, y, ...) {
                  xlim <- range(x@x)
                  ylim <- range(x@y)
                  plot(0, 0, type = "n", xlim = xlim,
                       ylim = ylim , ...)
                  xp <- c(x@x, x@x[1])
                  yp <- c(x@y, x@y[1])
                  lines(xp, yp)
          })
```

Polygon class

```
> setClass("polygon", [ ...OMITTED... ]
[1] "polygon"
>
> setMethod("plot", "polygon", [ ...OMITTED... ]
Creating a new generic function for "plot" in ".GlobalEnv"
[1] "plot"
> p <- new("polygon", x = c(1,2,3,4), y = c(1,2,3,1))
> plot(p)
```

Where to look, places to start

- The best way to learn this stuff is to look at examples.
- There are now quite a few examples on CRAN which use S4 classes/methods.
- My suggestions:
  - Bioconductor (http://www.bioconductor.org) — a rich resource, even if you know nothing about bioinformatics
  - Some packages on CRAN (as far as I know) — SparseM, gpclib, flexmix, its, lme4, orientlib, pixmap
  - The `stats4` package (comes with R) has a bunch of classes/methods for doing maximum likelihood analysis.