OOP programming in R

Necessity Really is a Mother

Paul E. Johnson[1][2]

[1]University of Kansas, Department of Political Science [2]Center for Research Methods and Data Analysis

2016

Outline

Object Oriented Programming

- A re-conceptualization of programming to reduce programmer error
- OO includes a broad set of ideas, only a few of which are directly applicable to R programming
- The "rise" to pre-eminance of OO indicated by the
  - introduction of object frameworks in existing languages (C++, Objective-C)
  - growth of wholly new object-oriented languages (Java)

Decipher R OO by Intuition

Run the command

```
methods(print)
```

What do you see? I see 170 lines, like so:

```
  [1] print.acf*
  [2] print.anova
  [3] print.aov*
  [4] print.aovlist*
...
[167] print.warnings
[168] print.xgettext*
[169] print.xngettext*
[170] print.xtabs*

   Non-visible functions are asterisked
```

170 print.??? Methods.

- Yes: there are really 170 print "methods"
- No: the R team does not expect or want you to know all of them. Users just run

```
print(x)
```

  Try not to worry about "how" the printing is achieved.
- Yes: R team wants package writers to create specialized print methods to control presentation of their output for the object they create.

The R Runtime System Handles the Details

- The user runs

```
print(x)
```

- The R runtime system
  1. notices that x is of a certain type, say "classOfx"
  2. and then the runtime system uses print.classOfX to handle the user's request

print is a "Generic Function"

Definition of "Generic Function": the function that users call which causes an object-specific function to be called for the work to get done. Examples:"print", "plot", "summary", "anova"

- Generic Function is terminology unique to R(AFAIK)
- In the standard case, a generic function does not do any work. It sends the work to the appropriate "implementation" in a method.

> *"A standard generic function does no computation other than dispatching a method, but R generic functions can do other coumputations as well before and/or after method dispatch"(Chambers, Software for Data Analysis, p. 398)*

print is a "Generic Function" ...

- UseMethod() is the function that declares a function as generic: The R runtime system is warned to "be alert" to usage of the function.
- Example: the print generic function from R source (base package).

```
print <- function(x, ...) UseMethod("print")
```

- Example: the plot generic function from R source (graphics package).

```
plot <- function (x, y, ...)  UseMethod("plot")
```

Here's Where R Gains its Analytical Power

- The generic is just a place holder. User runs print(x), then R knows it is supposed to ask x for its class and then the appropriate thing is supposed to happen. No Big Deal.
- But the statisticians in the S & R projects saw enormous simplifying potential in developing a battery is standard generic accessor functions
  - summary()
  - anova()
  - predict()
  - plot()
  - aic()

Object

Object: self-contained "thing". A container for data.

- Operationally, in R: just about anything on the left hand side in an assignment "<-"
- Each "thing" in R carries with it enough information so that generic functions "know what to do.".
- If there is no function specific to an object, the work is sent to a default method (see print.default).

Class

Definition: As far as R with S3 is concerned, class is a characteristic label assigned to an object (an object can have a vector of classes, such as c("lm", "glm")).

- The class information is used by R do decide which method should be used to fulfill the request.
- Run class(x), ask x what class it inherits from.
- In R, the principal importance of the "class" of an object is that it is used to decide which function should be used to carry out the required work when a generic function is used.
- Classes called "numeric", "integer", "character" are all vector classes

Class ...

```
> y <- c(1, 10, 23)
> class(y)
[1] "numeric"
> x <- c("a", "b", "c")
> x
[1] "a" "b" "c"
> class(x)
[1] "character"
> x <- factor(x)
> class(x)
[1] "factor"
> m1 <- lm(y ~ x)
> class(m1)
[1] "lm"
> m2 <- glm( y ~ x, family=Gamma)
> class(m2)
[1] "glm" "lm"
```

Method, a.k.a, "Method Function"

Definition: The "implementation": the function that does the work for an object of a particular type.

- When the user runs print(m1), and m1 is from class "lm", the work is sent to a method print.lm()
- Methods are always named in the format "generic.class", such as "print.default", "print.lm", etc.
- Note: Most methods do not "double-check" whether the object they are given is from the proper class. They count in R's runtime system to check and then call print.whatever for obejcts of type whatever
- That's why many methods are "hidden" (can only access via ::: notation)
- Accessing methods directly

Method, a.k.a, "Method Function" ...

- If a method is "exported", can be called directly via "package::method.class()" format
- If a package is "attached" to the search path, then "method.class()" will suffice, but is not as clear
- If a method is NOT exported, then user can reach into the package and grab it by running "package:::method.class()"

Detour: attributes() Function and Confusing Output

The class is stored as an attribute in many object types. Run attributes()

```
> attributes(x)
$levels
[1] "a" "b" "c"

$class
[1] "factor"

> attributes(m1)
$names
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "
     df.residual"
 [9] "contrasts"     "xlevels"       "call"          "terms"
[13] "model"

$class
[1] "lm"

> attributes(y)
```

Detour: attributes() Function and Confusing Output ...

```
NULL
> is.object(y)
[1] FALSE
```

- puzzle: why has y no attribute? Why is it not an object?
- Honestly, I'm baffled, I thought "everything in R is an object." (And I still do.)

If the object does not have a class attribute, it has an implicit class, "matrix", "array" or the result of "mode(x)" (except that integer vectors have the implicit class "integer"). (from ?class in R-2.15.1)

How Objects get "into" Classes

- In older S3 terminology, user is allowed to simply claim that x is from one or more classes

```
class(x) <- c(``lm'', ``glm'', class(x))
```

- That would say x's class includes "lm" and "glm" as new classes, and also would keep x's old classes as well.
- The class is an attribute, can be set thusly

```
attr(x, ``class'') <- c(``lm'', ``whateverISay'')
```

- When a generic method "run" is called with x, the R runtime will first try to use run.lm. If run.lm is not found, then run.whateverISay will be tried, and if that fails, it falls back to run.default.

How Objects get "into" Classes: S4

- S4 has more structure, makes classes & methods work more like truly object oriented programs.
- S4 classes are defined with a list of variables BEFORE objects are created.
- Variables are typed!
- Example imitates Matloff, p. 223

```
setClass("pjfriend", representation(
name="character",
gender="factor",
food="factor",
age="integer"))
```

- Create an instance of class pjfriend (Note: to declare an integer, add letter "L" to end of number).

How Objects get "into" Classes: S4 ...

```
william <- new("pjfriend", name = "william", gender =
  factor("male"), food=factor("pizza"), age=33L)
william
```

```
An object of class "pjfriend"
Slot "name":
[1] "william"

Slot "gender":
[1] male
Levels: male

Slot "food":
[1] pizza
Levels: pizza

Slot "age":
[1] 33
```

How Objects get "into" Classes: S4 ...

```
jane <- new("pjfriend", name="pumpkin", gender =
    factor("female"), food=factor("hamburger"), age=21L
    )
jane
```

```
An object of class "pjfriend"
Slot "name":
[1] "pumpkin"

Slot "gender":
[1] female
Levels: female

Slot "food":
[1] hamburger
Levels: hamburger

Slot "age":
[1] 21
```

- jane and william are *instances* of class "pjfriend"

How Objects get "into" Classes: S4 ...

- The variables inside an S4 object are called *"slots"* in R
- "slot" would be called "instance variable" in most OO-languages)
- values in slots can be retrieved with symbol @, not $

Implement an S4 method

- Step 1. Write a function that can receive a function of type "pjfriend" and do something with it.
- Step 2. Use setMethod to tell the R system that the function implements the method that is called for.
- setMethod "wraps" a function.

```
setMethod("some-generic-function-name", "pjfriend",
          function(x) {
              #do something with x
              }
)
```

Difficult to Account For Changes between S3 and S4

I think it is difficult to explain some of the notational and terminological changes between S3 and S4.

- If you type an S4 object's name on the command line

```
> x
```

  the R runtime looks for a method "show.class" (where class is the class of x).

- Why change from "print" to "show"? (IDK)
- Why change the "accessor" symbol from $ to @ ?
- Why call things accessed with @ "slots" rather than instance variables?