R: Object Oriented ProgrammingStatistics 506

Object Oriented Programming

When we attach values to names in an **R** environment we generally refer to the name and value collectively as an 'object'. We should, however, distinguish between base objects and object-oriented objects where the latter are those with a non-null class attribute. Compare the following:

```r
# A base object with NULL class attribute
attr(1:5, 'class')
```

```
NULL
```

```r
# An object of class factor.
attr(factor(1:5), 'class')
```

```
[1] "factor"
```

Object oriented programming is a programming paradigm built around the notions of classes, methods, and, of course, objects. There are a wide variety of object oriented languages and R has (at least) three *object oriented* (OO) systems you should be aware of:

1. S3 - R's original, informal OOP system;
2. S4 - a more formal less flexible version of S3;
3. RC - a reference class OOP system that more closely resembles the paradigm used in languages like C++.

We will focus on the S3 and S4 systems which predominate in R.

Before digging into R's OO systems it will be helpful to define a few terms.

- An object's **class** defines its structure and behavior using *attributes* as well as its relationship with other classes.
- **Methods** are functions that have definitions which depend on the class of an object.
- Classes are often organized into hierarchies with "child" classes defined more strictly than its "parents". Child classes often **inherit** from their parents meaning a parent's structure or methods serve as a default when not explicilty defined for the child. More formally, these are known as **superclasses** and **subclasses**.

The S3 system in R

The S3 system in **R** is based on the idea of **generic functions**. The core idea is that a **generic function** is used to **dispatch** a **class-specific method** taken from an object passed to it. Some common S3 generic functions in **R** inlcude, `print`, `summary`, `plot`, `mean`, `head`, `tail`, and `str`. Consider `summary`. The functions operates quite differently depending on its input:

```
vec <- 1:3
mat <- matrix(1:4, nrow = 2)
mod <- lm(mpg ~ wt, data = mtcars)
summary(vec)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    1.0     1.5     2.0     2.0     2.5     3.0 
```

```
summary(mat)
```

```
       V1              V2      
 Min.   :1.00   Min.   :3.00  
 1st Qu.:1.25   1st Qu.:3.25  
 Median :1.50   Median :3.50  
 Mean   :1.50   Mean   :3.50  
 3rd Qu.:1.75   3rd Qu.:3.75  
 Max.   :2.00   Max.   :4.00  
```

```
summary(mod)
```

```
Call:
lm(formula = mpg ~ wt, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max 
-4.5432 -2.3647 -0.1252  1.4096  6.8727 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  37.2851     1.8776  19.858  < 2e-16 ***
wt           -5.3445     0.5591  -9.559 1.29e-10 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.046 on 30 degrees of freedom
Multiple R-squared:  0.7528,	Adjusted R-squared:  0.7446 
F-statistic: 91.38 on 1 and 30 DF,  p-value: 1.294e-10
```

How did `summary` know what the right thing to do was? One way it could work would be a series of conditionals.

```r
summary <- function(object) {
  if (is(object, "vector")) {
    ...
  } else if (is(object, "matrix")) {
    ...
  } else if (is(object, "lm")) {
    ...
  }
}
```

This isn't tenable for two reasons:

1. This functions would be become overwhelmingly large.
2. Any new types of objects would require modifying the original function.

Instead, R (and most object-oriented languages) use *generics* and *dispatching*. Let's look at the code for `summary`:

```r
summary
```

```
function (object, ...)
```

```
UseMethod("summary")
<bytecode: 0x12e839190>
<environment: namespace:base>
```

When `UseMethod()` is called **R** searches for an S3 method based on the name of the generic function and the class of its first argument. The specific function it looks for follows the naming pattern `function.class()` – this is why it is advisable not to use dots when naming functions, classes, or other objects. There may exist functions `summary.vector()`, `summary.matrix()`, and `summary.lm()` - and if a new class type needs a `summary` function, you need only define `summary.newclass()`, rather than modify the existing `summary` functions.

As an example, let's examine a call to `head(mat)` in mode detail:

```
class(mat)
```

```
[1] "matrix" "array"
```

```
head(mat)
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

The object `mat` has class `matrix` so `UseMethod("head")` searches for a function (method) called `head.matrix()` to apply to `mat`:

```
identical(head(mat), head.matrix(mat))
```

```
[1] TRUE
```

```
head.matrix
```

```
function (x, n = 6L, ...)
{
    checkHT(n, d <- dim(x))
    args <- rep(alist(x, , drop = FALSE), c(1L, length(d), 1L))
    ii <- which(!is.na(n[seq_along(d)]))
    args[1L + ii] <- lapply(ii, function(i) seq_len(if ((ni <- n[i]) <
        0L) max(d[i] + ni, 0L) else min(ni, d[i])))
    do.call(`[`, args)
}
<bytecode: 0x13d3427a8>
<environment: namespace:utils>
```

You can see all the methods associated with a generic function using `methods()`.

```r
methods(head)
```

```
[1] head.array*      head.data.frame* head.default*    head.ftable*
[5] head.function*   head.matrix
see '?methods' for accessing help and source code
```

The * following some methods is used to denote methods that are not exported as part of the namesapce of the packages in which they are defined. For instance, the `head.data.frame` method is defined in the (base) package `utils`, but is not exported.

```r
## Try utils::head.data.frame. Why does it fail?
getS3method("head", "data.frame")
```

```
function (x, n = 6L, ...)
{
    checkHT(n, d <- dim(x))
    args <- rep(alist(x, , drop = FALSE), c(1L, length(d), 1L))
    ii <- which(!is.na(n[seq_along(d)]))
    args[1L + ii] <- lapply(ii, function(i) seq_len(if ((ni <- n[i]) <
        0L) max(d[i] + ni, 0L) else min(ni, d[i])))
    do.call(`[`, args)
}
<bytecode: 0x12d3352e0>
<environment: namespace:utils>
```

When an object has more than one class, **R** searches successively until a suitable method is found. If a suitable method is not found, S3 generics revert to a default method when defined and throw an error if not.

```r
getS3method('head', 'default')
```

```
function (x, n = 6L, ...)
{
    checkHT(n, dx <- dim(x))
    if (!is.null(dx))
        head.array(x, n, ...)
    else if (length(n) == 1L) {
        n <- if (n < 0L)
            max(length(x) + n, 0L)
        else min(n, length(x))
        x[seq_len(n)]
    }
    else stop(gettextf("no method found for %s(., n=%s) and class %s",
        "head", deparse(n), sQuote(class(x))), domain = NA)
}
<bytecode: 0x15c250058>
<environment: namespace:utils>
```

Defining a new S3 method is as simple as defining a function and naming it accordingly. Here we define a method `head.green`.

```
##' @title Head of a `green` object
##' @param obj A `green` object
head.green <- function(obj) {

  # Check if its green
  if ("green" %in% class(obj)) {
    if (length(class(obj)) > 1) {
      next_class <- class(obj)[-grep("green", class(obj))][1]
      cat("This is a green ", next_class, ".\n", sep = "")

      # This calls the next available method, allowing us to offload work in
      # a method for the subclass to an existing method for the superclass.

      NextMethod("head")

    } else {
      cat("This a generic green object.\n")
    }
  } else {
    cat("The object is not green!\n")
  }
```

```r
}
```

Now we can test it under various conditions.

```r
## We previously assigned
class(mat)
```

```
[1] "matrix" "array"
```

```r
head(mat)
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```r
## Test head.green for generic class
class(mat) <- c("green", class(mat))
head(mat)
```

```
This is a green matrix.
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```r
## Test on a non-green object
red_obj <- 1:100
class(red_obj) <- "red"
head.green(red_obj)
```

```
The object is not green!
```

```r
head(red_obj)
```

```
[1] 1 2 3 4 5 6
```

In our definition of `head.green`, notice the use of `NextMethod()` to dispatch a method previously defined on one of the parent classes.

We can similarly define our own S3 generic functions via `UseMethod()`.

Note that the first argument to both `UseMethod()` and `NextMethod()` should be the a character vector with the name of the generic.

```r
##' @title Return color of an object
##' @param obj Any colored object
##' @return The color of the object
##' @rdname getColor
getColor <- function(obj) {
  UseMethod("getColor")
}

##' @rdname getColor
getColor.default <- function(obj) {
  # Are any classes colors?
  ind <- class(obj) %in% colors()
  if (any(ind)) {
     # Yes. Return color with highest class predence.
     class(obj)[which(ind)[1]]
  } else {
    # default to black
    "black"
  }
}

##' @rdname getColor
getColor.green <- function(obj) {
  "darkgreen"
}

obj <- 1
getColor(obj)
```

```
[1] "black"
```

```r
class(obj) <- "blue"
getColor(obj)
```

```
[1] "blue"
```

```r
class(obj) <- "green"
getColor(obj)
```

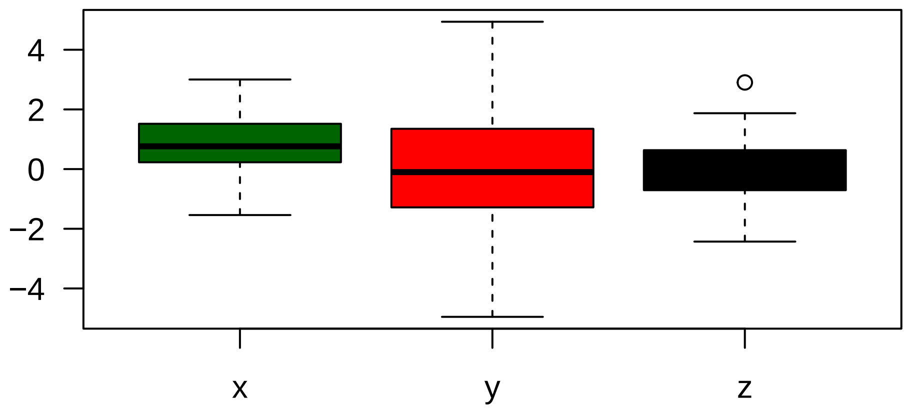
```
[1] "darkgreen"
```

Note the use of the Roxygen `@rdname`. This specifies that these functions should have their help on the same shared page.

As a quick, somewhat contrived, example of how we might use this, we could define a `col_boxplot` function to pick colors according to the class of the object passed.

```r
##' A box plot function that uses the class attribute to define colors.
##' @param dat A list of vectors
##' @param ... Additional arguments to `boxplot`
col_boxplot <- function(dat, ...) {
  stopifnot(is.list(dat))
  col <- sapply(dat, getColor)
  boxplot(dat, col = col, ...)
}
```

```r
# Define some iid data
x <- rnorm(100, 1, 1)
class(x) <- 'green'
y <- rnorm(100, 0, 2)
class(y) <- 'red'
z <- rnorm(100, 0, 1)
col_boxplot(list(x = x, y = y, z = z), las = 1)
```



You should be aware that the class of the object returned by some generic functions (especially primitives) can depend on the input class.

```
class(x + y)
```

```
[1] "green"
```

```
class(y + x)
```

```
[1] "red"
```

```
class(mean(x))
```

```
[1] "numeric"
```

An Example S3 class

Let's define an S3 class to define numbers which carry their own rounding information. As we've seen, defining an S3 class is a very casual affair - we simply assign a `class` to an object. Most S3 classes are built off one of the base R types - `vector`, `matrix`, `list` or `data.frame`, and then we define some functions along with them. We'll create a helper function first to generate our new `rounded_numeric` class.

```r
##' @title Create `rounded_numeric`
##' @param n Numeric
##' @param digits Digits to round to
##' @return New `rounded_numeric` object
new_rounded_numeric <- function(n, digits) {
  out <- list(n = n,
              digits = digits)
  class(out) <- c("rounded_numeric", class(out))
  return(out)
}
```

Note the extension of the `class` for fall back.

```r
rn <- new_rounded_numeric(pi, 3)
class(rn)
```

```
[1] "rounded_numeric" "list"
```

```r
print(rn)
```

```
$n
[1] 3.141593

$digits
[1] 3

attr(,"class")
[1] "rounded_numeric" "list"
```

Now we simply define a `print.rounded_numeric` class which operates as we'd like.

```r
print.rounded_numeric <- function(rn) {
  return(round(rn$n, rn$digits))
}
print(rn)
```

```
[1] 3.142
```

The S4 System

The S3 system described above is very informal and very flexible, making it easy to work with, but at the expense of the safety and uniformity of a more formal OO system.

The S4 system is a more formal OO system in R. One key difference is that S4 classes have formal definitions and classes, methods, and generics must all be explicitly defined as such. The functionality of the S4 object system comes from the (base) “methods” package.

Defining an S4 class

S4 classes are defined using the `setClass` function:

```r
setClass("color_vector",
         slots = c(data = "numeric",
                   color = "character"))
```

Create a new instance of an S4 class using `new`:

```r
x <- new("color_vector", data = 3, color = "darkgreen")
x
```

```
An object of class "color_vector"
Slot "data":
[1] 3

Slot "color":
[1] "darkgreen"
```

The function `new` is used above as a *constructor* for creating an object with the desired class. Most S4 classes defined in packages you download have their own constructors which you should use when defined. We can create a default constructor by assigning the output of `setClass` a name:

```
color_vector <- setClass("color_vector",
                         slots = c(data = "numeric",
                                   color = "character"))
y <- color_vector(data = 1:4,
                  color = "red")
y
```

```
An object of class "color_vector"
Slot "data":
[1] 1 2 3 4

Slot "color":
[1] "red"
```

You can also define a constructor as a whole new function, thus giving you more flexibility in argument names.

```
color_vector2 <- function(x, color) {
  return(new("color_vector", data = x, color = color))
}
z <- color_vector2(5:10, "green")
z
```

```
An object of class "color_vector"
Slot "data":
[1]  5  6  7  8  9 10

Slot "color":
[1] "green"
```

Accessing slots in an S4 object

You can access and set attributes for an S4 object using an `@` symbol, the `slot` function, or an `attr(obj, 'name')` construction:

```
x@color
```

```
[1] "darkgreen"
```

```
x@data <- rnorm(10, 1, 1)
slot(x, "color")
```

```
[1] "darkgreen"
```

```
attr(x, "color") <- "purple"
names(attributes(x))
```

```
[1] "data"  "color" "class"
```

Validator

A validator is a function that ensures an object is a valid member of a given class. Here is an example validator for our "color_vector" class.

```
setValidity("color_vector", function(object){
  if (!(object@color %in% colors())) {
    stop(paste("@color = ", object@color, "is not a valid color"))
  }
  return(TRUE)
})
```

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

Slots:

Name:       data     color
Class:   numeric character
```

```
color_vector2(1:3, color = "A")
```

```
Error in validityMethod(object): @color =  A is not a valid color
```

The validator gets run anytime a new object of the class is created, or you can run it at will with the `validObject()` function.

```
y@color <- "a"
y
```

```
An object of class "color_vector"
Slot "data":
[1] 1 2 3 4

Slot "color":
[1] "a"
```

```
validObject(y)
```

```
Error in validityMethod(object): @color =  a is not a valid color
```

S4 Methods

We can control how an object of class `color_vector` gets displayed by defining a `show` method (the S4 equivalent of `print`).

```
## This is an S4 generic
show(x)
```

```
An object of class "color_vector"
Slot "data":
 [1] -0.26736859  1.47403991 -0.32615213  1.73214276  1.68835838  1.88292755
 [7]  0.06139153  1.30562774  1.35769505  0.58370283

Slot "color":
[1] "purple"
```

```
##' @title Display a `color_vector` object
##' @param object A `color_vector` object
setMethod("show", "color_vector",
  function(object) {
    cat(object@color, ":")
    str(object@data)
    cat("\n")
    return(invisible(object))
  }
)
```

The use of `return(invisible(...))` ensures that something sensible gets returned. If you truly want nothing returned, try `return(invisible(NULL))`.

Now, when we call `show` on an object of class `color_vector` **R** will use the custom method.

```
show(x)
```

```
purple : num [1:10] -0.267 1.474 -0.326 1.732 1.688 ...
```

```
# Note: show, like print, is the default method for an unassigned object.
x
```

```
purple : num [1:10] -0.267 1.474 -0.326 1.732 1.688 ...
```

In addition, there are often so-called "getter" and "setter" functions for obtaining and modifying slots. These are often more formally called "accessors" and "mutators".

First, we define a new S4 generic.

```
setGeneric("color",
           function(object) {
             standardGeneric("color")
           })
```

```
[1] "color"
```

Next, we can define a version for `color_vector`.

```
##' @title Return the color of a `color_vector`
##' @param object A `color_vector` object
##' @return It's color
setMethod("color", "color_vector",
          function(object) {
            return(slot(object, "color"))
          })

color(x)
```

```
[1] "purple"
```

```r
color(LETTERS)
```

```
Error: unable to find an inherited method for function 'color' for signature 'object = "chara
```

Note that there isn't the idea of a default dispatch like there is in S3 - There's nothing like `color.default`. There are some hacky ways around this (such as `setMethod(..., signature(object = "any"), ...)`, but generally you need to write a specific method for each class.

We can similarly define a method that allows the user to change the value in the color slot, the "setter" or "mutator"

```r
setGeneric("color<-",
           function(object, value) {
             standardGeneric("color<-")
           })
```

```
[1] "color<-"
```

```r
##' @title Set the color of a `color_vector`
##' @param object A `color_vector` object
##' @param value New color.
##' @return The updated object
setMethod("color<-", "color_vector",
  function(object, value) {
    object@color <- value
    validObject(object) # Re-run validity check
    return(object)
  }
)

color(x) <- "purple"
color(x)
```

```
[1] "purple"
```

```r
show(x)
```

```
purple : num [1:10] -0.267 1.474 -0.326 1.732 1.688 ...
```

```
color(x) <- "a"
```

```
Error in validityMethod(object): @color =  a is not a valid color
```

```
x # unchanged since "setter" failed
```

```
purple : num [1:10] -0.267 1.474 -0.326 1.732 1.688 ...
```

We can also define math operations in the same fashion.

```
getMethod("+")
```

```
function (e1, e2)  .Primitive("+")
```

```
##' @title `color_vector` arithmetic.
##'
##' New object always have the color of the left-side object.
##' @param e1 A `color_vector`
##' @param e2 A `color_vector`
##' @return A `color_vector`
setMethod("+", signature(e1 = "color_vector",
                         e2 = "color_vector"),
          function(e1, e2) {
            return(color_vector2(e1@data + e2@data,
                                 e1@color))
          })
x
```

```
purple : num [1:10] -0.267 1.474 -0.326 1.732 1.688 ...
```

```
y <- color_vector2(5, "blue")
y
```

```
blue : num 5
```

```
x + y
```

```
purple : num [1:10] 4.73 6.47 4.67 6.73 6.69 ...
```

Inheritance

In addition to slots, S4 classes can inherit from another class. The benefit of doing this is that, similar to how S3 functions dispatch on the list of classes, functions on an S4 class that inherit another class will dispatch to those class's functions.

```
newclass1 <- setClass("newclass1",
                      slots = c(a = "numeric",
                                b = "character"))
newclass2 <- setClass("newclass2",
                      contains = "numeric",
                      slots = c(b = "character"))
nc1 <- newclass1(a = 5, b = "c")
nc1
```

```
An object of class "newclass1"
Slot "a":
[1] 5

Slot "b":
[1] "c"
```

```
nc2 <- newclass2(5, b = "c")
nc2
```

```
An object of class "newclass2"
[1] 5
Slot "b":
[1] "c"
```

The "contained" class can be accessed via the special slot `@.Data`:

```
rbind(slotNames(nc1), slotNames(nc2))
```

```
     [,1]    [,2]
[1,] "a"     "b"
[2,] ".Data" "b"
```

```
nc2@.Data
```

```
[1] 5
```

Neither new class has any methods defined for it, but since `newclass2` "contains" a numeric, method dispatch will work for it:

```
sum(nc1)
```

```
Error in sum(nc1): invalid 'type' (S4) of argument
```

```
sum(nc2)
```

```
[1] 5
```

This can be very powerful when inheriting more "advanced" classes such as`data.frame`, or even more powerfully, something like `lm` - this allows you to write your own custom function for working with a model output, without having to re-implement all the existing generic functions such as `coefficients`, `predict`, or `residuals`.

You can of course have a custom S4 class inherit another custom S4 class.

```
newclass1 <- setClass("newclass1",
                      slots = c(a = "logical"))
newclass2 <- setClass("newclass2",
                      contains = "newclass1",
                      slots = c(b = "numeric"))
c1 <- newclass1(a = TRUE)
c2 <- newclass2(c1, b = 5)
c1
```

```
An object of class "newclass1"
Slot "a":
[1] TRUE
```

```
c2
```

```
An object of class "newclass2"
Slot "b":
[1] 5

Slot "a":
[1] TRUE
```

Note that the slots for the two classes “mix” together:

```
slotNames(c2)
```

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

```
c2@a
```

```
[1] TRUE
```

```
c2@b
```

```
[1] 5
```

Slot names cannot be overwritten in the superclass:

```
newclass3 <- setClass("newclass3",
                      contains = "newclass2",
                      slots = c(a = "numeric"))
```

```
Error in reconcilePropertiesAndPrototype(name, slots, prototype, superClasses, : Definition 
```