Class and Objects in R: S3 Style

Abhishek Tiwari

Citation: *A. Tiwari*, "Class and Objects in R: S3 Style", Abhishek Tiwari, 2014. doi:10.59350/3by2m-e2w34

Published on: March 06, 2014

When it comes to R Programming, I guess you can always consider me as a beginner. Although I have known basic R for quite some time now, I never tried object-oriented programming in R. Recently I started to explore some object-oriented stuff in R. To my surprise, R has three object-oriented systems: S3, S4 and R5. Creator of ggplot, Hadley Wickham maintains a brief description of all three types of system S3, S4 and R5.

Unfortunately there is not much documentation or guideline about implementing S3 style class and objects in R. Most of tutorials and book chapters cover how to use relatively new S4 style system as S4 has some nice features such as method dispatch on multiple arguments, consistency check, control over inheritance patterns. Despite the goodness of S4 system, as R programer you have to learn S3 because most of core R packages make heavy use of S3 and also S4 systems has it's own problems.

Rather than going into detailed comparison of S3 vs S4, I would like to introduce my workflow for S3 system (Please consider it as documentation and don't forget to drop a comment if you find something wrong in this post). Here is what I usually do,

1. Creating objects
2. Defining methods

Creating objects

A S3 object is nothing but a R object with an additional 'class' attribute, a character vector giving the names of the classes.

Simple way

So given a R object (vector, factor, matrix, array, list , data.frame or anything else) you can easily convert your R object in object of certain class type by simply attaching the 'class' attribute. If object belongs to one type of class `cname` then

```
attr(your.r.object, "class") <- "cname"

class(your.r.object) <- "cname"

your.class.object <- structure(your.r.object, class = "cname")
```

else if object belongs to many class types `cname1`, `cname2` and so on then,

```
attr(your.r.object, "class") <- c("cname1", "cname1", ..)

class(your.r.object) <- c("cname1", "cname1", ..)
```

```
your.class.object <- structure(your.r.object, class = c("cname1", "cname1"
    , ..))
```

So for example,

```
# Output on R console
> my.r.obj <- c(1:10)
> class(my.r.obj)
[1] "integer"
> # That was in-built type, 'integer'
> class(my.r.obj) <- c("real", "positive")
class(my.r.obj)
[1] "real"     "positive"
# Object belongs to class type both 'real' and 'positive'
```

Constructor function

Otherwise you can use a class type cunstructor function to create new object of certain class type,

```
# Constructor function for the class
cname <- function(x, ...)  {
  args <- list(...)
  # Do something here with x, args and put in something
  object <- list(attribute.name = something, ..)
  class(object) <- "cname"
  return (object)
}

# Creating object of class type
my.object <- cname(x, ...)

# To access object attributes,  my.object$attribute.name
my.object$attribute.name
```

As matter fact S3 System can be used to generalize class constructor function and definition itself, more about that later. ##Defining methods When dealing with S3 system we are interested in creating methods of generic functions. In S3 system, generic functions work by naming convention. So for instance, 'print' is a generic function with alternative definitions for different class types. You can see all existing definitions for a generic function using `methods()`. Format looks like a list of `gname.cname` (gname is generic function name and cname is class name), so for print there will be print.default, print.ts, print.table and so on.

So when you define a new S3 class constructor function `cname`, you need to define corresponding methods such as `print.cname`, `plot.cname`, `ggplot.cname` etc. Now generic function dispatches the appropriate matching S3 method with `UseMethod()` based on the type of the first argument (S4 system uses multiple arguments). If no method is defined, as fall back the default method

.default will be used like `print.default`, `plot.default` etc.

```
methods(gname)

methods(print)
# more than 150 definitions for print
```

There are generic functions such as print, plot etc those are S3 ready, which mean they already include `UseMethod()`. For instance,

```
# Output on R console
> print
function (x, ...)
UseMethod("print")
<environment: namespace:base>
> plot
function (x, y, ...)
{
    if (is.function(x) && is.null(attr(x, "class"))) {
        if (missing(y))
            y <- NULL
        hasylab <- function(...) !all(is.na(pmatch(names(list(...)),
            "ylab")))
        if (hasylab(...))
            plot.function(x, y, ...)
        else plot.function(x, y, ylab = paste(deparse(substitute(x)),
            "(x)"), ...)
    }
    else UseMethod("plot")
}
<environment: namespace:graphics>
```

For others you may need to define your own generic function with `UseMethod()`. For example in package ggplot function `ggplot` is generic (it offers only two methods `ggplot.data.frame` and `ggplot.default`) but not S3 ready. Similarly `qplot` is not generic. So if you want to extend `ggplot` and `qplot` for a new class type you may need to define a custom generic function. How you will do that? So here is stepwise instruction,

1. Check if your generic function is S3 ready (whether or not generic function includes `UseMethod()`), else
2. Create a function named gname . In the function body, call `UseMethod("gname")`.
3. For each custom class type, create a function called . with first argument as an object of class classname.

```
# Step 2
gname <- function (x, ...)  {
  if(is.null(attr(x, "class"))){
    args <- list(...)
```

```
    #do something here for shake
  }
  else  UseMethod("gname", x) # x is object
}
# Step 3
gname.default <- gname(x)

gname.cname1 <- function(x, ...)  {
  # First argument is an object of class
  args <- list(...)
  # do something here
}

gname.cname2 <- function(x, ...)  {
  if(some condition is true) {
  # Do something here
  }
  # Only if 'class' attribute of an S3 object is a vector
  else  NextMethod()
}

gname.cname3 <- function(x, ...)  {
  # First argument is an object of class
  args <- list(...)
  # do something here
}
```

`NextMethod()` function provides a simple inheritance mechanism by invoking the next method only if 'class' attribute of an S3 object is a vector. `NextMethod()`, should be used in a method called by `UseMethod()`. From Hadley's notes on S3

> `NextMethod()` works like `UseMethod()` but instead of dispatching on the first element of the class vector, it will dispatch based on the second (or subsequent) element.

A simple example,

```
# Constructor function for class
animal <- function(x)  {
  object <- list(name = x)
  class(object) <- x
  return (object)
}

# Generic methods
milk <- function (x)  {
  if(is.null(attr(x, "class"))){
    print("no animal no milk")
  }
  else  UseMethod("milk", x)
}
```

```
milk.default <- milk(x)

milk.cow <- function(x)  {
  # Only if class of an S3 object is a vector
  NextMethod()
}

milk.goat <- function(x)  {
  # First argument is an object of class classname
  print(x$name)
  print("Yummy milk mix")
}

milk.donkey <- function(x)  {
  # First argument is an object of class classname
  print(x$name)
  print("Milk mix, not again")
}

# Creating object of class type
> a <- animal(c("cow","goat"))
> milk(a)
[1] "cow"  "goat"
[1] "Yummy milk mix"
> b <- animal(c("cow","donkey"))
> milk(b)
[1] "cow"    "donkey"
[1] "Milk mix, not again"
> c <- animal(c("cow"))
> milk(c)
Error in NextMethod() : no method to invoke
> milk("me")
[1] "no animal no milk"
```

Extending the Internal generics

Many primitive (written in C) and internal functions in R are generic and allow methods to be written for new class types. For example, `[`, `[[`, `$`, `length`, `c` , `unlist` , `unlist`, `cbind`, `rbind` can be extended for a new class type. To see a list of primitives which are internal check `.S3PrimitiveGenerics` and for all internal generic function `?InternalMethods`. Note internal dispatch only occurs on objects, that is those for which is.object returns true. Again from Hadley's notes on S3,

> Internal generic have a slightly different dispatch mechanism to other generic functions: before trying the default method, they will also try dispatching on the mode of an object, i.e. `mode(x)`.

An example of internal generic method adopted from Stackoverflow,

```
# Constructor function for class
myclass <- function(x, n)  {
  structure(x, class = "myclass", n = n)
}

# Generic methods to extend the 'c' and '['
c.myclass <- function(..., recursive = FALSE)  {
  dots <- list(...)
  ns <- sapply(dots, attr, which = "n")
  classes <- rep("myclass", length(dots))
  res <- structure(unlist(dots, recursive = FALSE), class = classes)
  attr(res, "n") <- ns
  return (res)
}

`[.myclass` <- function (x, i)  {
  y <- unclass(x)[i]
  ns <- attr(x, "n")[i]
  class(y) <- "myclass"
  attr(y, "n") <- ns
  return (y)
}

# Creating object of class type
> x1 <- myclass(1, 5)
> x2 <- myclass(2, 6)
> c(x1, x2)
[1] 1 2
attr(,"class")
[1] "myclass" "myclass"
attr(,"n")
[1] 5 6
> c(x1, x2)[2]
[1] 2
```

Generalizing the constructor functions

Now coming back to class constructor functions, for a given class constructor function can be generalized so that it can accommodate different type of forms for the class.

```
# Generalised Constructor function for class geom
geom <- function(x, ...) UseMethod("geom")

geom.default <- function(x, ...)  {
  object <- list(area = x^2)
  class(object) <- "geom"
  return (object)
```

```
}

geom.rectangle <- function(x, ...)  {
  args <- list(...)
  object <- list(area = x*as.numeric(args[1]))
  class(object) <- "geom"
  return (object)
}

geom.triangle <- function(x,...)  {
  args <- list(...)
  object <- list(area = (x*as.numeric(args[1])*as.numeric(args[2]))/2)
  class(object) <- "geom"
  return (object)
}

> my.object1 <- geom(3)
> my.object1
$area
[1] 9

attr(,"class")
[1] "geom"
> my.object2 <- geom.rectangle(3,4)
> my.object2
$area
[1] 12

attr(,"class")
[1] "geom"
> my.object3 <- geom.triangle(3,4,5)
> my.object3
$area
[1] 30

attr(,"class")
[1] "geom"
```