PharmaSUG China 2022 - Paper AT-135

Developing R code using S3 and R6 Object Oriented approach

Shuang Gao, BeiGene (Beijing) Co., Ltd

ABSTRACT

R, at its heart, is a functional programming (FP) language. Functional style means decomposing a big problem into smaller pieces, then solve each piece with a function or combination of functions. R support also exists for programming in an OOP style. Object oriented programming is a programming paradigm in which your program using objects to represent things you are programming about. Object Oriented Programming in R is a superb tool to manage complexity in larger programs. There are multiple OOP systems to choose. S3, R6 and S4 are the commonly used methods. This paper can help beginner with R programmers on their journey from getting to know S3 and R6 methods to developing R code using S3 and R6 Object Oriented approach.

INTRUDUCTION

Object Oriented Programming (OOP) is a popular programming paradigm. Using its concepts, we can construct modular pieces of code that can be used to build blocks for large systems. R is a functional language. The support also exists for programming in an OOP style. Object Oriented Programming in R is a superb tool to manage complexity in larger programs. There are multiple OOP systems to choose. S3, R6 and S4 are the commonly used methods.

In this paper, we will focus on S3 and R6 methods. S4 is used to make good effect in Bioconductor community. We provide detailed instructions on how to develop R code using S3 and R6 methods through examples, discussions and a live demo.

OBJECT ORIENTED PROGRAMMING BASIC CONCEPTS

- Class
  - Class is a template for defining objects. Describe object that serve as blueprints for creating objects, and their attributes and methods.
- Method
  - Describe what the object of class can do. Methods are functions that are defined inside a class that describes the behaviors of an object.
- Fields (Attributes)
  - Defined by the class. This is the data possessed by all instances of that class.
- Object
  - In short, an object is a data structure having some fields and methods which act on its attributes. An object is a member or an “instance” of a class.
- Method dispatch
  - The process of finding the correct method for a given class.
- Polymorphism
  - Use the same function form for different types of input.
- Inheritance
  - Inheritance is the mechanism of basing an object or class upon another object. An inherited class is called a subclass of its parent class or super class.
- Encapsulation

– The user doesn’t need to worry about an object’s details because they are ‘encapsulated’ behind a standard interface.

WHAT IS S3

S3 refers to a class system built into R. The system governs how R handles objects of different classes. Certain R functions will look up an object’s S3 class and then behave differently in response.

- R’s the first and simplest OO system.
- Base R uses it, if possible, to solve simple problems. It is the most commonly used system in CRAN packages.
- Know how to use S3 is a fundamental R skill. S3 allows your functions to return rich results with user-friendly display and programmer-friendly internals.

HOW TO MAKE YOUR OWN S3 METHOD

DEFINING A S3 OBJECT

What do you need for an S3 object? S3 object = base type + class attribute.

S3 class is unlike any other object-oriented programming in other languages. S3 has no formal definition of a class: to make an object an instance of a class, we simply set the class attribute. For example, Base type of factor is the integer vector, it has a class attribute of “factor”, and a level attribute that stores the possible levels. You can do that during creation with structure () or after the fact with class ():

```
x <- c("apple", "orange", "orange")
# structure() Method
structure(x, class="myclass")
#> [1] "apple"  "orange" "pear"
#> attr(,"class")
#> [1] "myclass"

# class() Method
class(x) <- "myclass"
```

GENERIC FUNCTIONS AND METHODS

To interact with an S3 object, you have to use functions. There are two types of functions involved: generic functions and methods.

1.Generic functions:

Method dispatch starts with a generic function that decides which specific method to dispatch to, as shown in Figure 1. Depending on the class of an argument to the generic, it uses a different implementation. All generic functions have the same form: a call to UseMethod that specifies the generic name and the object to be dispatched on. Functions such as print, plot, and summary adapt their action according to different types of objects. They are known as generic functions.

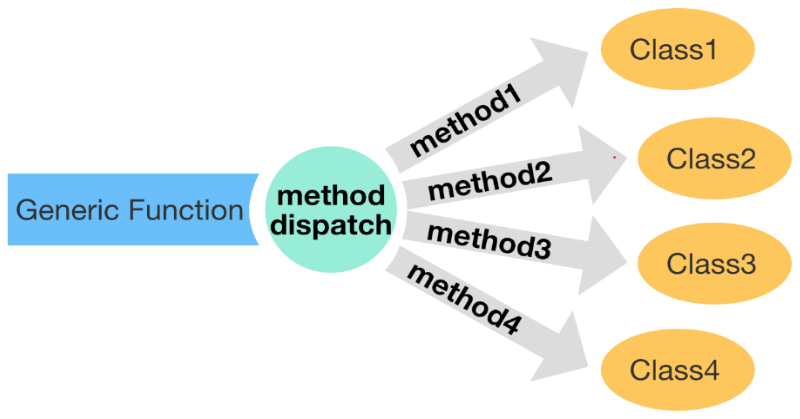


Figure 1. Method dispatch

Example of a generic function print:

```
# Use sloop::s3_methods_generic to see the methods associated with a generic
function.
sloop::s3_methods_generic("print")
# A tibble: 216 x 4
#>  generic class                    visible source
#>   <chr>   <chr>                    <lgl>   <chr>
#>1 print   acf                      FALSE   registered S3method
#>2 print   anova                    FALSE   registered S3method
#>3 print   aov                      FALSE   registered S3method
#>4 print   aovlist                  FALSE   registered S3method
# ...
```

Creating your own generic method:

```
my_new_generic <- function(x) {
  UseMethod("my_new_generic")
}
```

2. Methods and Methods dispatch:

Methods are written to be used with a specific class. To find out which classes a generic function has methods for, you can use the methods function. Remember, in R, that methods are associated with functions (not objects). When you call print, print calls a special function. We can use print () with vectors, matrix, data frames, factors etc. and they get printed differently according to the class they belong to.

```
g <- factor(c("apple", "orange", "pear"))
s3_dispatch(print(g))
#>=> print.factor
#> * print.default
```

WRITING YOUR OWN METHODS

If you want to write your own method, there are 2 cases:

1. There is a pre-existing generic function: Create a new method generic.class.
2. There is not a pre-existing generic function: Create a new generic and new methods.

```
print_fruit <- function(x) { # Generic Function
  UseMethod("print_fruit")
}
print_fruit.default <- function(x, ...){ # Default Method
```

```
  print("No specific method defined")
}
print_fruit.myclass <- function(x, ...){ # Method for Class = myclass
  paste0(x,sep="/")
}

print_fruit(x)
#>   paste0(x,sep="/")
# [1] "apple/"  "orange/" "pear/"
```

SUMMARY

S3 programming requires three steps:

- Define a S3 object that sets the class and the attributes.
- Define new generics with UseMethod().
- Implement methods for new and existing generics.

S3 has no formal definition of a class. It is essential to perform more computationally expensive checks to ensure that the object has correct values.

WHY S3

Simple:

- S3 objects are very easy to deal with. You can also define multiple classes to an object.
- As Hadley Wickham states: “Overall, when picking an OO system, I recommend that you default to S3. S3 is simple, and widely used throughout base R and CRAN. While it’s far from perfect, its idiosyncrasies are well understood and there are known approaches to overcome most shortcomings.”

Easier to maintain:

- It can make your codes much more understandable and organized, especially if you are working on a package. Functions can be assembled and organized into R packages. S3 methods in the R programming language are ways of writing functions in R that do different things for objects of different classes.
- It avoids a large sequence of if-else () statements. It can only consider the classes we currently know about. Straightforward code is always easier to maintain.

WHEN SHOULD I USE R’S S3 OBJECT ORIENTED APPROACH

An R function is a set of R statements organized together for a specific task and can have arguments. An R package is a collection of R functions that enables users to expand R functionality. Developing R packages is typically divided into eight steps: Planning, Requirements, Design, Build, Document, Test, Deploy and Maintain. At the design phase, collecting all requirements needed to generate an R package for functions, and list functions to be developed.

For each function, consider using the S3 Object-Oriented approach when functions need to adapt their actions according to different types of objects. A good function or package offers flexibility and readiness for implementation of change requests and should be easy to understand. S3 is enough flexible and is easily extendible.

WHAT IS R6

The R6 package provides an implementation of encapsulated object-oriented programming for R. Unlike the functional programming style, R6 encapsulates methods and fields in classes that can be instantiated into objects. R6 classes are like R’s reference classes but are more efficient and do not depend on S4 classes and the methods package. It is powerful for abstracting complex objects with lots of self-contained components that you might want to update.

- A common need for R6 is to model data that comes from a web API and where changes come from inside or outside of R.
- R6 is a powerful OOP framework method. R6 is a profoundly different OO system from S3 and S4 because it is built on encapsulated objects, rather than generic functions. In encapsulated OOP, like R6, methods belong to objects. The R6 system, by contrast, is an encapsulated object-oriented system akin to those in Java or C++, where objects contain methods in addition to data, and those methods can modify objects directly.

HOW TO MAKE YOUR OWN R6

CREATE R6 CLASSES

R6 classes are created by calling R6:R6Class() and passing a list of methods and fields. The following is a simple example of defining an R6 class:

```
Person <- R6Class(
  classname = "Person",
  public = list(
    name = NULL,
    age = NA,
    country =NULL,
    initialize = function(name=NA, age = NA ,country=NA ) {
      self$name <- name
      self$age <- age
      self$country <- country
    },
    greet = function() {
      cat("Person: ")
      cat("\nName: ", self$name, sep = "")
      cat("\nAge: ", self$age,sep = "")
      cat("\nCountry: ", self$country, sep = "")

    }
  )
)
```

- The first argument classname by convention uses UpperCamelCase.
- The second argument, public, supplies a list of methods (greet) and fields (name, age, country) that make up the public interface of the object.
- $initialize() overrides the default behavior when creating a new object.

```
Person
#><Person> object generator
#> Public:
#>   name: NULL
#>   age: NA
#>   country: NULL
#>    initialize: function (name = NA, age = NA, country = NA)
#>    greet: function ()
#>    clone: function (deep = FALSE)
#>  Parent env: <environment: R_GlobalEnv>
#>  Locked objects: TRUE
#> Locked class: FALSE
#> Portable: TRUE
```

CREATE NEW OBJECT

Create a new instance of a class with the $new() method. Access fields and methods with $.

```
person_1<-Person$new(name="Ann",age=24,country ="China")

person_1$age
#> 24
person_1$greet()
#> Person:
#> Name: Ann
#> Age: 24
#> Country: China
```

WHY R6

- R6 objects take less memory and are significantly faster than R’s reference class objects and they also have some options that provide for even much higher speed. R6 implements encapsulated OOP like R’s reference class objects but resolve some important issues.
- Inheritance (superclasses) which works across packages.

WHEN USE R6

This article mainly focuses on how to use R6 in the Shiny app. Shiny is an R package that makes it easy to build interactive web apps straight from R. Shiny’s straightforward implementation of UI/Server interface makes it an ideal tool to create applications that can produce great efficiency for the repetitive tasks that a statistical programmer often encounters.

Sharing an R6 object makes it simpler to create data and methods that are shared across modules without the complexity generated by reactive objects, and the instability of using global variables. Or when working with shiny modules that are interdependent, R6 is a good choice.

SHARING DATA AND METHOD ACROSS MODULES

Here’s an example of using R6 as data storage mentioned in the book Engineering Production-Grade Shiny Apps (https://engineering-shiny.org/common-app-caveats.html?q=r6#using-r6-as-data-storage).

Create R6 class - MyDataProcessing

```
MyDataProcessing <- R6::R6Class(
  "MyDataProcessing",
  # Defining our public methods, that will be the dataset container, plot fun
ction and data clean function
  public = list(
    data = NULL,
    initialize = function(data){
      self$data <- data
    },
    plot = function(){
      ....
    }

    clean = function(arg1,arg2){
      ....
    }
  )
)
```

Create modules UI and Server

Module server functions take as argument, an R6 object. Modules interact directly with the R6.

```r
data_cleaning_ui <- function(id){
  ns <- NS(id)
  tagList(
    # Defining the UI for your first module
    # [...]
  )
}

mod_data_cleaning_server <- function(id, r6){
  moduleServer( id, function(input, output, session){
    ns <- session$ns
    observeEvent( input$launch_cleaning , {
      # Once the launch_cleaning input is triggered, we
      # use the internal method from our r6 object
      r6$clean(arg1 = input$a, arg2 = input$b)
      # Triggering the plot
      trigger("plot")
    })
  })
}

plotting_ui <- function(id){
  ns <- NS(id)
  tagList(
    # Defining the UI for your second module
    # [...]
  )
}

mod_plotting_server <- function(id, r6){
  moduleServer( id, function(input, output, session){
    ns <- session$ns
    # Rendering, inside this second module, the plot based on the
    # cleaning done in the other module
    output$plot <- renderPlot({
      # We use the trigger/watch pattern from before
      watch("plot")
      # Calling the plot() method from our R6 object
      r6$plot()
    })
  })
}
```

Call app.R

Create an instance of the R6 class in server.R, and call the modules with it.

```r
ui <- function(){
  tagList(
    # Putting our two module UIs here
    data_cleaning_ui("data_cleaning_ui"),
    plotting_ui("plotting_ui")
  )
}
```

```
server <- function(
  input,
  output,
  session
){
  # We start by creating a new instance
  r6 <- MyDataProcessing$new()
  # Passing this object to the two server functions
  mod_data_cleaning_server("data_cleaning_ui_1", r6)
  mod_plotting_server("plotting_ui_1", r6)

}

shinyApp(ui, server)
```

SUMMARY

Using R6 class for sharing data and methods that need to be passed between the modules. The author recommends the following these steps:

- Creating a class named like “Myclass” which contains reactive methods (reactive()), non-reactive methods (clean(), plot()) and field (data), as shown in Figure 2.
- Module server functions take the R6 object as an argument. Modules interact directly with the R6 object.
- Create an instance of the R6 class in server.R, and call the modules with it, as shown in Figure 3.

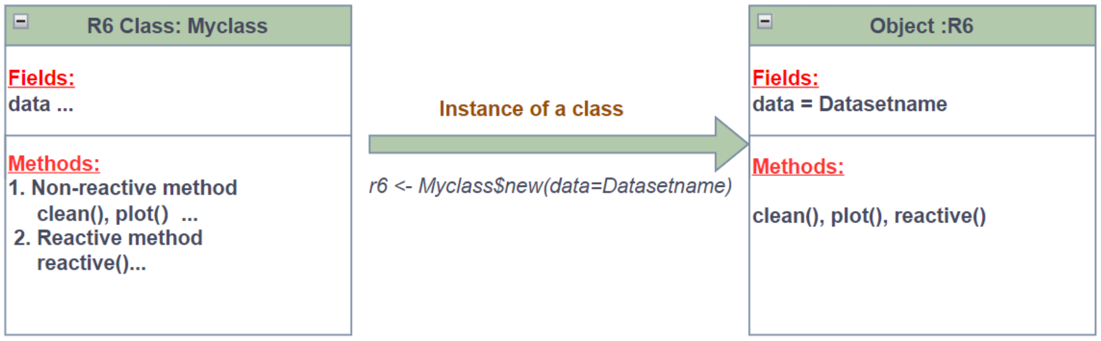
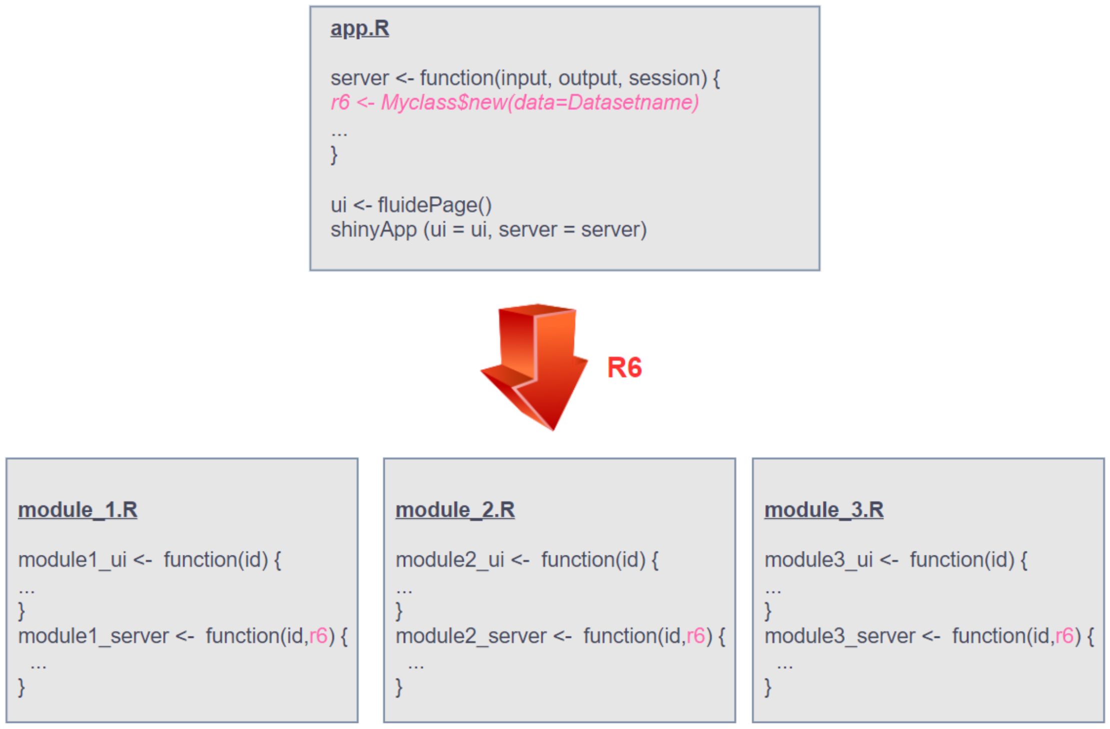


Figure 2. Create R6 class and object



```
app.R

server <- function(input, output, session) {
r6 <- Myclass$new(data=Datasetname)
...
}

ui <- fluidePage()
shinyApp (ui = ui, server = server)
```

```
module_1.R

module1_ui <-  function(id) {
...
}
module1_server <-  function(id,r6) {
  ...
}
```

```
module_2.R

module2_ui <-  function(id) {
...
}
module2_server <-  function(id,r6) {
  ...
}
```

```
module_3.R

module3_ui <-  function(id) {
...
}
module3_server <-  function(id,r6) {
  ...
}
```

Figure 3. Shiny App

DISCUSSION AND CONCLUSION

OOP has become the most popular paradigm for organizing computer software. Some of the most well-known object-oriented languages are Perl, Java, Python, and C++. Object-oriented programming can help us grasp and analyze the whole system from a macroscopic perspective. In this paper, we introduce the concepts of the S3 and R6 methods and their examples. The author hopes that this paper will provide you with some simple guidance and ideas on how to implement the S3 and R6 methods.

Until recently, Object Oriented Programming has not widely used in R comparing with other programming languages (Perl, Java, Python, and C++). We cannot deny that OOP has significant benefits in Web API and complex design tasks. OOP language allows you to break the program into bit-sized problems that can be solved easily. The author hopes that OOP systems will be widely and effectively used in R.

REFERENCES

Wickham, H. (n.d.). Advanced R. Retrieved July 22, 2022, from https://adv-r.hadley.nz/oo.html

Dealing with S3 methods in R with a simple example. (2017, July 11). R-Bloggers. https://www.r-bloggers.com/2017/07/dealing-with-s3-methods-in-r-with-a-simple-example/

Dubel, M. (2019, October 15). Super solutions for Shiny Apps #4 of 5: Using R6 classes. R-Bloggers. https://www.r-bloggers.com/2019/10/super-solutions-for-shiny-apps-4-of-5-using-r6-classes/

Fay, C., Rochette, S., Guyader, V., & Girard, C. (n.d.). Chapter 15 common application caveats. Engineering-shiny.org. Retrieved July 22, 2022, from https://engineering-shiny.org/common-app-caveats.html?q=R6

Gunuganti, A., & Inc, P. (n.d.). Application development framework for R/shiny. Lexjansen.com. Retrieved July 22, 2022, from https://www.lexjansen.com/pharmasug/2018/AD/PharmaSUG-2018-AD24.pdf

R4DS advanced R bookclub. (n.d.). Bookclub-Advanced_R. Retrieved July 22, 2022, from https://r4ds.github.io/bookclub-Advanced_R/

R6: Encapsulated object-oriented programming for R. (n.d.). Retrieved July 22, 2022, from https://github.com/r-lib/R6

Radečić, D. (2022, June 9). Object-Oriented Programming (OOP) in R with R6 – The Complete Guide. R-Bloggers. https://www.r-bloggers.com/2022/06/object-oriented-programming-oop-in-r-with-r6-the-complete-guide/

Smith, D. (2017, July 27). The R6 class system. R-Bloggers. https://www.r-bloggers.com/2017/07/the-r6-class-system/

Yang, A., Zhu, Y., & Zhang, Y. (n.d.). PharmaSUG 2021 -paper SI-074 A strategy to develop specification for R functions in regulated clinical trial environments. Lexjansen.com. Retrieved July 22, 2022, from https://www.lexjansen.com/pharmasug/2021/SI/PharmaSUG-2021-SI-074.pdf

AKNOWLEDGEMENT

Thanks to my manager Wei Wei for his great support. Also, many thanks to Daniel Huang for his careful reviewing and inputs.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Shuang Gao

BeiGene (Beijing) Co., Ltd

Shuang.Gao@BeiGene.com