Programming with objects in R

Alexandre Courtiol

Leibniz Institute for Zoo and Wildlife Research

21 March 2018

Content

Definitions

What is an object?
An instance of a class.

What is a class?
A data structure for which specific methods can be defined.

What is a method?
A function that is designed to work with all objects of a given class.

R is an object-based system

Rule #1:
Everything that exists in R is an object

(John M. Chambers)

Example

```
> res <- 1
```

```
> res
## [1] 1
> class(res)
## [1] "numeric"
> methods(class = "numeric")
## [1] all.equal     as.data.frame as.Date       as.POSIXct    as.POSIXlt
## [6] as.raster     coerce        Ops
## see '?methods' for accessing help and source code
```

```
> `<-`
## .Primitive("<-")
> class(`<-`)
## [1] "function"
> methods(class = "function")
## [1] as.list  coerce   coerce<- head     plot     print    tail
```

What is the point of objects?

Using objects allows abstraction!

```
> foo1 <- 1:10
>
> head(foo1)

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

```
> foo2 <- matrix(foo1, ncol = 1)
>
> head(foo2)

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

```
> foo3 <- as.data.frame(foo2)
>
> head(foo3)

##   V1
## 1  1
## 2  2
## 3  3
## 4  4
## 5  5
## 6  6
```

Abstraction is made possible via methods

Methods behind the generic function `head`:

```
> methods(head)

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

Methods behind the generic function `residuals`:

```
> methods(residuals)

## [1] residuals.default*       residuals.glm            residuals.HoltWinters*
## [4] residuals.isoreg*        residuals.lm             residuals.nls*
## [7] residuals.smooth.spline* residuals.tukeyline*
## see '?methods' for accessing help and source code
```

Objects are accessed or modified using references

Reference = a name + an environment:

```
> ls()
## [1] "foo1" "foo2" "foo3" "res"
> environmentName(pryr::where("res"))
## [1] "R_GlobalEnv"
```

Objects are accessed or modified using references

Behind one reference there is one memory address:

```
> pryr::address(res)

## [1] "0x55a860b25608"
```

Behind one memory address there can be several references:

```
> res2 <- res
> pryr::address(res2)

## [1] "0x55a860b25608"
```

Note: as we will see, the address behind a reference can change during computation.

Why environments and not just names?

Environments are used to define the scope of the objects
(i.e. from where they can be seen and accessed)

To solve ambiguities:

```
> i <- 20
> foo <- function() {i <- 1; return(i)}
> foo()

## [1] 1

> i

## [1] 20
```

To keep things tidy:

```
> ls(".GlobalEnv")

## [1] "foo"   "foo1" "foo2" "foo3" "i"
## [6] "res"   "res2"

> head(ls("package:base"))

## [1] "-"          "-.Date"    "-.POSIXt"
## [4] ":"          "::"        ":::"

> length(ls("package:base"))

## [1] 1217
```

Note: environments are objects!

There are different systems for defining and using objects

Native class systems:

- S3 (legacy from S version 3, in *base*)
- S4 (legacy from S version 4, in core package *methods*)
- Reference Class (sometimes referred to as R5, in *methods*)

There are different systems for defining and using objects

Native class systems:

- S3 (legacy from S version 3, in *base*)
- S4 (legacy from S version 4, in core package *methods*)
- Reference Class (sometimes referred to as R5, in *methods*)

Additional class systems:

- R6 (in the package *R6*, one of the most downloaded package on CRAN!)
- ggproto (in the package *ggplot2*)
- others (proto, ...)

Note: the objects created with one system can contain objects created with another.

You can create your own classes and methods

Example; we want to improve the following outputs:

```
> res
## [1] 1
> AIC(res)
## Error in UseMethod("logLik"):  no applicable method for 'logLik' applied to an object of class
"c('double', 'numeric')"
```

You can create your own classes and methods

Example using S3:

```
> resS3 <- 1
> class(resS3) <- "alexS3"
> print.alexS3 <- function(x, ...) print(paste("The object of class 'alexS3' is equal to", x), ...)
> AIC.alexS3   <- function(x) return(NA)
```

```
> resS3

## [1] "The object of class 'alexS3' is equal to 1"

> AIC(resS3)

## [1] NA
```

You can create your own classes and methods

Example using S4:

```
> setClass("alexS4", slots = list(value = "numeric"))
> setMethod("show", signature = "alexS4", definition = function(object)
+   print(paste("The object of class 'alexS4' is equal to", object@value)))

## [1] "show"

> setMethod("AIC", signature = "alexS4", definition = function(object) return(NA))

## [1] "AIC"
```

```
> resS4 <- new("alexS4", value = 1)
> resS4

## [1] "The object of class 'alexS4' is equal to 1"

> AIC(resS4)

## [1] NA
```

You can create your own classes and methods

Example using RC:

```
> alexRC <- setRefClass(Class   = "alexRC",
+                       fields  = list(value = "numeric"),
+                       methods = list(
+                       show   = function()
+                         print(paste("The object of class 'alexRC' is equal to", value)),
+                       AIC    = function() return(NA)))
```

```
> resRC <- alexRC$new(value = 1)
> resRC

## [1] "The object of class 'alexRC' is equal to 1"

> resRC$AIC()

## [1] NA
```

You can create your own classes and methods

Example using R6:

```
> library(R6)
> alexR6 <- R6Class(
+   public  = list(
+     value = NA,
+     initialize = function(value) self$value <- value,
+     print = function()
+         print(paste("The object of class 'alexR6' is equal to", self$value)),
+     AIC = function() return(NA)
+   )
+ )
```

```
> resR6 <- alexR6$new(value = 1)
> resR6

## [1] "The object of class 'alexR6' is equal to 1"

> resR6$AIC()

## [1] NA
```

Content

The 2 main object-based programming paradigms

Functional (Object Oriented) Programming

- suitable for analytical workflows
- S3, S4
- methods defined outside the objects
- objects are not mutable
- $a \xrightarrow{fn} b \xrightarrow{fn} c \xrightarrow{fn} \dots$

The 2 main object-based programming paradigms

Functional (Object Oriented) Programming

- suitable for analytical workflows
- S3, S4
- methods defined outside the objects
- objects are not mutable
- a $\xrightarrow{\text{fn}}$ b $\xrightarrow{\text{fn}}$ c $\xrightarrow{\text{fn}} \dots$

Encapsulated Object Oriented Programming (aka OOP)

- suitable for data that evolve over time (modularity and reusability)
- RC, R6
- methods defined inside the objects
- objects are mutable
- a $\xrightarrow{\text{fn}}$ b & a $\xrightarrow{\text{fn}} \varnothing$

Note: actual programming can borrow from multiple paradigms (pure form is difficult).

Functional Programming: simple example

```
> res_fp <- 1
> addone <- function(x) x + 1
> res_fp <- addone(res_fp)
> res_fp

## [1] 2
```

```
> replicate_fp1 <- function(n, obj, fn){
+   for (i in 1:n) obj <- fn(obj)  ## not truly functional
+   return(obj)
+ }
```

```
> replicate_fp2 <- function(n, obj, fn, i = 1){
+   res <- ifelse(n > 0, replicate_fp2(n = n - 1, obj = fn(obj), fn = fn), obj)
+   return(res)
+ }
```

```
> res_fp <- 1
> res_fp <- replicate_fp2(n = 10, obj = res_fp, fn = addone)
> res_fp

## [1] 11
```

Encapsulated OOP: simple example

```
> resObj <- R6Class(
+   public  = list(
+     value = NA,
+     initialize = function(value) self$value <- value,
+     addone = function() self$value <- self$value + 1
+   )
+ )
```

```
> res_oop <- resObj$new(value = 1)
> res_oop$addone()
> res_oop$value

## [1] 2
```

```
> res_oop$value <- 1
> replicate(10, res_oop$addone())

##  [1]  2  3  4  5  6  7  8  9 10 11

> res_oop$value

## [1] 11
```

Encapsulated OOP requires mutability

S3 objects are (generally) not mutable (same for S4):

```
> a <- 1:3
> a
## [1] 1 2 3
> pryr::address(a)
## [1] "0x55a86132b890"
```

```
> a[2] <- 10
> a
## [1]  1 10  3
> pryr::address(a)
## [1] "0x55a861351600"
```

```
> b <- a
> b
## [1]  1 10  3
> pryr::address(b)
## [1] "0x55a861351600"
```

```
> b[2] <- 11
> pryr::address(b)
## [1] "0x55a86188bfe8"
> pryr::address(a)
## [1] "0x55a861351600"
```

```
> a
## [1]  1 10  3
```

Encapsulated OOP requires mutability

R6 objects are mutable (same for RC):

```
> resR6$value
## [1] 1
> pryr::address(resR6)
## [1] "0x55a8610b25c8"
```

```
> resR6$value <- 3
> resR6$value
## [1] 3
> pryr::address(resR6)
## [1] "0x55a8610b25c8"
```

```
> resR6_bis <- resR6
> pryr::address(resR6_bis)
## [1] "0x55a8610b25c8"
```

```
> resR6_bis$value
## [1] 3
> resR6_bis$value <- 4
> resR6_bis$value
## [1] 4
```

```
> resR6$value
## [1] 4
```

Encapsulated OOP requires mutability

What can be mutable does not have to be!

```
> pryr::address(resR6)

## [1] "0x55a8610b25c8"

> resR6_ter <- resR6$clone()
> pryr::address(resR6_ter)

## [1] "0x55a860cc5db0"
```

```
> resR6_ter$value <- 5
> resR6_ter

## [1] "The object of class 'alexR6' is equal to 5"

> resR6

## [1] "The object of class 'alexR6' is equal to 4"
```

Note: the cloning creates a physical copy of the original object and thus the 2 clones have different addresses (confusing: 2 clones and more or less similar than 2 copies depending on how you think about it. . .)

Pros and cons of mutability

Pros

- less verbose
- save memory

Cons

- side effects

Note: most R operations perform copy-on-write (also called copy-on-modify) to avoid side effects.

Encapsulated OOP in S3 by messing with the scoping

```
> res <- 1
> replicate(10, res <- addone(res))

## [1] 2 2 2 2 2 2 2 2 2 2

> res

## [1] 1
```

```
> res <- 1
> replicate(10, res <<- addone(res))

## [1]  2  3  4  5  6  7  8  9 10 11

> res

## [1] 11
```

```
> res <- 1
> e <- environment()
> replicate(10,
+           eval(substitute(res <- addone(res)), envir = e))

## [1]  2  3  4  5  6  7  8  9 10 11

> res

## [1] 11
```

Note: but the addresses do change (`res` is copied in memory many times).

Note: this is also what the functions ‘[‘ rely on.

Content

Our Individual Based Models (IBM)

Setup:

- two age classes: children (0–14 yrs), adults (15+ yrs)
- children die at an average rate of 15 deaths per 1000 per year
- adults die at an average rate of 5 deaths per 1000 per year
- children do not reproduce
- adults reproduce at an average rate of 10 births per 1000 per year

Question:

Starting with 1000 individuals (with age following a uniform distribution between 0 and 40 yrs), what is the number of children and adults after 50 years?

Functional way using S3: functions

```
> death <- function(pop) {
+   death_children <- rbinom(n = length(pop), size = 1, prob = 15/1000)
+   death_adults   <- rbinom(n = length(pop), size = 1, prob = 5/1000)
+   alive <- rep(1, length(pop))
+   alive[pop < 15] <- 1 - death_children[pop < 15]
+   alive[pop > 14] <- 1 - death_adults[pop > 14]
+   pop <- pop[alive == 1]
+   return(pop)
+ }
>
> birth <- function(pop) {
+   adults <- pop[pop > 14]
+   babies_nb <- sum(rbinom(n = length(adults), size = 1, prob = 10/1000))
+   babies <- rep(0, babies_nb)
+   pop <- c(pop, babies)
+   return(pop)
+ }
>
> age <- function(pop) pop <- pop + 1
```

Functional way using S3: run

```
> pop <- round(runif(1000, min = 0, max = 40))
>
> for (i in 1:50) {
+   pop <- birth(pop)
+   pop <- death(pop)
+   pop <- age(pop)
+ }
> table(pop > 14)

##
## FALSE  TRUE
##   112   979
```

OOP way using R6: definition of the class `individual`

```
> individual <- R6Class(
+   public  = list(
+     age = NA,
+     alive = 1,
+     initialize = function(age = 0) {self$age <- age},
+     die = function() {
+       if (self$age < 15 & self$alive) self$alive <- 1 - rbinom(n = 1, size = 1, prob = 15/1000)
+       if (self$age > 14 & self$alive) self$alive <- 1 - rbinom(n = 1, size = 1, prob =  5/1000)
+     },
+     reproduce = function() {
+       ifelse(self$alive == 1 & self$age > 14, rbinom(n = 1, size = 1,  prob = 10/1000), FALSE)
+     },
+     aging = function() {
+       if (self$alive == 1) {self$age <- self$age + 1; self$die()}
+     }
+   )
+ )
```

OOP way using R6: test

You can test things before creating the population!

```
> alex <- individual$new()
> alex

## <R6>
##   Public:
##     age: 0
##     aging: function ()
##     alive: 1
##     clone: function (deep = FALSE)
##     die: function ()
##     initialize: function (age = 0)
##     reproduce: function ()
```

```
> for (i in 1:200) alex$aging()
> alex

## <R6>
##   Public:
##     age: 46
##     aging: function ()
##     alive: 0
##     clone: function (deep = FALSE)
##     die: function ()
##     initialize: function (age = 0)
##     reproduce: function ()
```

OOP way using R6: definition of the class population

```
> population <- R6Class(
+   public  = list(
+     individuals = list(),
+     initialize = function(N = 1000) {
+         for (i in 1:N)
+           self$individuals[[i]] <- individual$new(age = round(runif(n = 1, min = 0, max = 40)))
+     },
+     repro = function() {
+       for (i in 1:length(self$individuals))
+         if (self$individuals[[i]]$reproduce() == TRUE)
+           self$individuals[[length(self$individuals) + 1]] <- individual$new()
+       },
+     death = function(){
+       alive <- sapply(self$individuals, function(i) i[["alive"]])
+       self$individuals[!alive] <- NULL
+       },
+     aging = function(){
+       for (i in 1:length(self$individuals)) self$individuals[[i]]$aging()
+       },
+     year = function() {self$repro(); self$death(); self$aging()},
+     count = function() table(sapply(self$individuals, function(i) i[["age"]]) > 14)
+   )
+ )
```

OOP way using R6: run

```
> pop <- population$new()
> for (i in 1:50) pop$year()
> pop$count()
##
## FALSE  TRUE
##    98   967
```

Note: I did not handle the possible population crash, so it may crash :-/

Our Individual Based Models (IBM): UPDATE

Setup:

- two age classes: children (0–14 yrs), adults (15+ yrs)
- children die at an average rate of 15 deaths per 1000 per year
- adults die at an average rate of 5 deaths per 1000 per year
- children do not reproduce
- adults reproduce at an average rate of 10 births per 1000 per year
- two sexes, females do not reproduce after 45 yrs (males do not reproduce)

Question:

Starting with 1000 individuals (with age following a uniform distribution between 0 and 40 yrs), what is the number of children and adults after 50 years?

OOP way using R6: re-definition of the class individual

```
> individual <- R6Class(
+   public  = list(
+     age = NA,
+     alive = 1,
+     sex = NA,                                                              ##new
+     initialize = function(age = 0) {
+       self$age <- age
+       self$sex <- ifelse(runif(1) < 0.5, "male", "female")                ##new
+     },
+     die = function() {
+       if (self$age < 15 & self$alive) self$alive <- 1 - rbinom(n = 1, size = 1, prob = 15/1000)
+       if (self$age > 14 & self$alive) self$alive <- 1 - rbinom(n = 1, size = 1, prob =  5/1000)
+     },
+     reproduce = function() {
+       ifelse(self$alive == 1 &
+                (self$age > 14 & self$age < 44 & self$sex == "female"),      ##new
+              rbinom(n = 1, size = 1,  prob = 10/1000), FALSE)
+     },
+     aging = function() {
+       if (self$alive == 1) {self$age <- self$age + 1; self$die()}
+     }
+   )
+ )
```

Note: there is no need to redefine the class `population`!

OOP way using R6: re-run

```
> pop <- population$new()
> for (i in 1:50) pop$year()
> pop$count()

##
## FALSE  TRUE
##     2   861
```

Functional way using S3: re-defining functions

We would have to recode everything . . .

Pros and cons of R6 for IBM

Pros

- clearer structure (see butterfly example)
- easier to modify once in existing
- easier to share classes between projects, packages...
- easier to translate to C++

Cons

- initially difficult for those knowing mostly S3
- much slower (cost can somewhat be reduced with some tweaks)
- additional issues to take care (e.g. side effects)

Content

Remember

- everything that exists in R is an object
- several class systems can be used to create objects
- S3 and S4 are more suitable to functional programming
- RC and R6 are more suitable to encapsulated OOP
- most existing methods are writen in S3
- the focus of R on a functional paradigm based on objects is quite unusual
- which programming paradigm and which class system to rely on depends on your task and tastes[1]

[1] it is a vast and complex topic, I barely scratched the surface...

Did you sort your legos?

Note: did you have a lot of them?