**Classes and OOP in R**

One way we can define our own classes in R is via a function that returns a list of members and/or methods. An instance of our class is the list itself.  
  
Here is a silly little guessing game class. Copy the following code and execute it in R.

Guess <- function(max) {

secret.number <- sample(1:max, 1)

total.guesses <- 1

finished <- FALSE

message(sprintf("I'm thinking of a number from 1 to %d.", max))

self.list <- list(

get.info = function() {

message(

sprintf(

"You %s guessed the number!",

if (finished) "have already"

else "still haven't"

)

)

message(sprintf("Guess number: %d", total.guesses))

message(sprintf("The number is taken from 1 to %d.", max))

},

get.clues = function(guess) {

if (finished) message("You've already guessed my number!")

else {

if (guess == secret.number) {

message("Congratulations!")

message(sprintf("You got it in %d guesses!", total.guesses))

finished <<- TRUE

} else {

if (secret.number < guess) message("Too high!")

else message("Too low!")

total.guesses <<- total.guesses + 1

}

}

}

)

class(self.list) <- "Guess"

self.list

}

To play the game, create an instance of Guess and interact with it via its methods get.info() and get.clues(guess). For example:

> game <- Guess(100)

I'm thinking of a number from 1 to 100.

> class(game)

[1] "Guess"

> game$get.info()

You still haven't guessed the number!

Guess number: 1

The number is taken from 1 to 100.

> game$get.clues(50)

Too high!

> game$get.clues(25)

Too high!

> game$get.clues(12)

Too low!

> game$get.clues(17)

Too low!

> game$get.clues(21)

Too high!

> game$get.clues(19)

Too high!

> game$get.clues(18)

Congratulations!

You got it in 7 guesses!

Here are the main things we should notice.

* The function Guess returns a list that has, in this example, arbitrarily been labelled as class "Guess".
* The variable game is an instance of Guess, which just means that it is a list - in this case, a list containing two functions.
* The methods get.clues and get.info have access to the variables that exist in the body of Guess. For example, secret.number, total.guesses and finished can be accessed. We can think of these latter variables as "private" members of the class Guess. (Whatever appears in the list that is actually returned is, of course, "public".)
* Notice that these members are accessed using the <<- operator. This is because these members don't exist within the context of the respective function, so <- would actually create a new variable with the same name (that would only exist within that context), which is not what we want. The <<- operator, then, tells R *not* to create a new variable but to look for the variable in some parent context.
* Since game is simply a list, we access the methods of the object game in the same way that we access members of a list, i.e., using the $ operator. (We could also do such things as game[["get.info"]]().)

**Assignment 2**

So, to get back to assignment 2, the code that is given to us is:

makeVector <- function(x = numeric()) {

m <- NULL

set <- function(y) {

x <<- y

m <<- NULL

}

get <- function() x

setmean <- function(mean) m <<- mean

getmean <- function() m

list(set = set, get = get,

setmean = setmean,

getmean = getmean)

}

We can think of makeVector as a class definition. An instance of makeVector would have "private" members x and m and "public" methods set, get, setmean and getmean.

[+ Comment](javascript:void(0);)

##### [Richard Ambler](https://class.coursera.org/rprog-005/forum/profile?user_id=1875527)Community TA· [6 days ago](https://class.coursera.org/rprog-005/forum/thread?thread_id=139" \l "post-2294)

Remember that a full understanding of what is going on with closures is *not* required for assignment 2. In fact, you can copy-and-paste the code given in the assignment and then just change a couple of lines to perform the new task.  
  
In any case, I've dug up some related examples that I created for a previous iteration of this course. Sometimes it helps to see more than one example. I've clumped it all together so that you can just copy, paste in your console and enter to see the results. This is only if you'd like to understand a few of the inner details. If nothing else, look at it as some silly entertainment! :)

# Functions can take functions as arguments and

# functions can return functions in R.

# An example of a function that takes a function as an argument

hof <- function(f, x) cat(

sprintf("The value of y at x = %s is %s.\n", x, f(x))

)

# We can pass functions that already exist:

hof(sin, pi / 4)

# We can pass our own functions:

myFunction <- function(x) 100 \* x

hof(myFunction, -5)

# We can pass functions anonymously:

hof(function(x) 2 \* x, 10)

# An example of a function that returns a function

nTimesTedious <- function(n) {

f <- function(x) {

n \* x

}

f

}

# Which can be written more succinctly:

nTimes <- function(n) function(x) n \* x

threeTimes <- nTimes(3) # threeTimes is now a function of x that returns 3 \* x

fourTimes <- nTimes(4) # fourTimes returns 4 \* x

fiveTimes <- nTimes(5) # etc.

threeTimes(5)

fourTimes(5)

fiveTimes(5)

# We can even write expressions like the following in R:

nTimes(10)(5) # nTimes(10) returns a function, then we pass 5 to that function

# Lists of functions are often useful for grouping functions in a

# single object, and can be used to creating objects analagous to

# those in some other languages.

funcList = list(threeTimes, fourTimes, fiveTimes)

funcList[[1]](5)

funcList[[2]](5)

funcList[[3]](5)

# (Though it's probably easier to read if we name the list members...)

funcListBetter = list(

threeTimes = threeTimes,

fourTimes = fourTimes,

fiveTimes = fiveTimes

)

funcListBetter$threeTimes(5)

funcListBetter$fourTimes(5)

funcListBetter$fiveTimes(5)

# How is this analagous to a class?

# The elements in the list can be seen as class methods.

# Let's create a very simple "class" to represent a dog.

Dog <- function(name = "Fido", barkType = "Woof!") {

# Let's set up a "property" that contains a random number of

# times the dog will bark when asked to...

noOfBarks <- numeric(1)

updateBarkNumber <- function() noOfBarks <<- sample(1:10, 1)

updateBarkNumber()

# Let's create a "method" that may change the property noOfBarks...

f <- function() {

cat(c("YIP!\n", "Grr!\n", "YELP!\n", "Ouch!")[sample(1:4, 1)])

updateBarkNumber()

}

# And a "method" to get the dog to bark...

g <- function() cat(sprintf("%s", rep(barkType, noOfBarks)))

# The "properties" and "methods" that we'd like to be "public"

# are simply elements of a list that the function Dog returns; let's

# create the list and give the elements appropriate names...

dogObject <- list(name = name, swatWithNewspaper = f, bark = g)

class(dogObject) <- "Dog"

dogObject

}

# Now let's create some "instances" of Dog...

fido <- Dog()

snookums <- Dog(name = "Snookums", barkType = "Yap!")

fido$name

fido$bark()

snookums$name

snookums$bark()

snookums$bark()

snookums$swatWithNewspaper()

snookums$bark()

snookums$bark()

snookums$swatWithNewspaper()

snookums$bark()

fido$swatWithNewspaper()

fido$bark()

# We can even create "classes" within "classes" as in

# the following example.

# Let's create a "class" that represents a pack

# of cards form which we can draw random cards:

PackOfCards <- function() {

# A Card "class" within a PackOfCards "class"

Card <- function(x) {

# An "instance" of a Card is really just a list containing

# an integer x (named rawInteger, that we don't actually use

# in this example) as a "property" and a function

# that maps x to an interpretation of a card (named showCard)

# as a "method"...

list(rawInteger = x, showCard = function() {

face <- c(

"Ace", "Two", "Three", "Four", "Five", "Six", "Seven",

"Eight", "Nine", "Ten", "Jack", "Queen", "King"

)[x %% 13 + 1]

suit <- c(

"Hearts", "Diamonds",

"Clubs", "Spades"

)[x %/% 13 %% 4 + 1]

cat(sprintf("%s of %s\n", face, suit))

})

}

# Let's keep track of which cards are still in the pack.

cardInPack <- rep(1, 52)

# A PackOfCards "instance" is then simply a list containing

# several "methods".

list(

cardsRemaining = function() sum(cardInPack),

drawCard = function() {

if (sum(cardInPack) == 0) message("No more cards in the pack!")

else {

c <- numeric(1)

repeat {

c <- sample(1:52, 1)

if (cardInPack[c] == 1) break

}

cardInPack[c] <<- 0

# Create an "instance" of a Card and call its showCard "method"

card = Card(c)

card$showCard()

}

},

showRemainingCards = function() for (i in 1:52) if (cardInPack[i]) Card(i)$showCard()

)

}

pack1 <- PackOfCards()

pack1$cardsRemaining()

# Let's draw 40 cards from the deck

for (i in 1:40) pack1$drawCard()

pack1$cardsRemaining()

pack1$showRemainingCards()

<http://adv-r.had.co.nz/OO-essentials.html> : This chapter is a field guide for recognising and working with R’s objects in the wild. R has three object oriented systems (plus the base types), so it can be a bit intimidating. The goal of this guide is not to make you an expert in all four systems, but to help you identify which system you’re working with and to help you use it effectively.