R Cheat Sheet: Environments, Frames and the Call Stack

Environments Code example explained

1) R uses environments to store the name-The above dictionary function returns the
object pairing between variable name and list at the end of the function. That list
the R object assigned to that variable and the listed callable functions exist in
(assign creates pair: <-, <<-, assign()) the environment created when the dictionary
2) They are implemented with hash tables. function was called. This use of functions
3) Like functions, environments are "first and lexical scoping is a poor man's OOP-
class objects" in R: They can be class-like mechanism. The function also
created, passed as parameters and creates an environment (e), which it uses
manipulated like any other R object. for its hash table properties to save and
4) Environments are hierarchically retrieve key-value pairs.
organised (each env. has a parent).
5) When a function is called, R creates a Lexical and dynamic scoping
new environment and the function R is a lexically scoped language. Variables
operates in that new environment. All are resolved in terms of the function in
local variables to the function are which they were written, then the function
found in that environment (aka frame). in which that function was written, all
they way back to the top-level global/
Code example: package environment where the program was
dictionary <- function() { written. Variables are not resolved in
# *private* ... effectively hidden terms of the functions that called them
e <- new.env(parent=emptyenv()) when the program is running (dynamic
# use emptyenv() to stop chained lookup scoping). Interrogating the function call
keyCheck <- function(key) # sanity chk stack allows R to simulate dynamic scoping.
stopifnot(is.character(key) &&
length(key) == 1) Frames and environments
# *public* ... made public by list below A frame is an environment plus a system
hasKey <-function(key) { reference to a calling frame. R creates
keyCheck(key) each frame to operate within (starting with
exists(key, where=e, the global environment, then a new frame
inherits=FALSE) with each function call). All frames have
} associated environments, but you can create
rmKey <- function(key) { environments that are not associated with
stopifnot(!missing(key)) the call stack (like we did with e above).
keyCheck(key)
rm(list=key, pos=e) The call stack
} As a function calls a new function, a stack
putObj <- function(key, obj=key) { of calling frames is built up. This call
stopifnot(!missing(key)) stack can be interrogated dynamically.
keyCheck(key) # some call stack functions ...
if(is.null(obj)) return(rmObj(key)) sys.frame() # the current frame
assign(key, obj, envir=e) parent.frame() # get the frame for the
} # calling function (an env)
getObj <- function(key) { parent.frame(1) # same as above
stopifnot(!missing(key)) parent.frame(2) # get the grandparent
keyCheck(key) # function's frame
if(!hasKey(key)) return(NULL) # parent.frame(n) is the same as ...
e[[key]] # also $ indexing possible # sys.frame(sys.parent(n))
} sys.nframe() # the current frame number
allKeys <- function() # (global environment = 0)
ls(e, all.names=TRUE) # on the call stack
allObjs <- function() sys.call() # returns the call (which
eapply(e, getObj, all.names=TRUE) # is language expression)
list(hasKey=hasKey, allKeys=allKeys, sys.call(-1) # parent function's call
rmKey=rmKey, getObj=getObj, sys.call(1) # the first function call
putObj=putObj, allObjs= allObjs) # on the call stack down
} # from the global env.
d <- dictionary(); # create deparse(sys.call())[[1]] # string name
sapply(LETTERS, d$putObj) # populate # of this function
d$hasKey('A'); d$allKeys() # inspect # potential confusions ...
d$allObjs() # inspect parent.env(sys.frame()) # lexical scoping
d$getObj('A') # retrieve Sys.getenv() #Operating System environment
d$rmKey('A'); d$hasKey('A') # remove Sys.setenv() #as above – not an R env.