R6

Daniel Heimgartner

12/5/2021

Introduction

Official Documentation

Two special properties:

- Encapsulated OOP paradigm (methods belong to objects, not generics) `object$method()`
- Reference semantics -> objects are mutable (which means they are modified in place; not copied-on-modify)

Further remarks:

- R6 is very similar to RC
- `R6::R6Class()` only method needed from `library(R6)`
- Initializer and finalizer
- Private and active fields
- Fields and methods as opposed to variables and functions

Classes and methods

`R6Class()` two most important arguments:

- Classname (UpperCamelCase)
- Public -> list of methods (funcitons) and fields (anything else) -> public interface of the object
  - snake_case
  - access methods and fields of current object via `self$`

Further remarks:

- Call methods and access fields with `$`

Method chaining

Side-effect R6 methods (modifying fields) should always **return self invisibly** which returns the current object and hence allows method chaining. Chaining is deeply related to pipe.

```r
library(R6)

Accumulator <- R6Class("Accumulator", list(
  sum = 0,
  add = function(x = 1) {
    self$sum <- self$sum + x
    invisible(self)
  }
))

x <- Accumulator$new()
x$
  add(10)$
  add(20)$
  sum
```

```
## [1] 30
```

Important methods

- `$initialize()` overwrites the default behaviour of `$new()`
  - If you have more expensive validation requirements, implement them in a separate `$validate()`
- `$print()` overwrites the default printing behaviour -> should return `invisible(self)`
- Because methods are boudn to individual object, previously created *person* objects won't get this new print method… -> So if something does not behave as planned it might be a good idea to re-construct the objects with the new class!

```
Person <- R6Class("Person", list(
  name = NULL,
  age = NA_real_,
  initialize = function(name, age = NA_real_) {
    self$name <- name
    self$age <- age
  },
  print = function(...) {
    cat("Person: \n")
    cat(" Name: ", self$name, "\n", sep = "")
    cat(" AGe:  ", self$age, "\n", sep = "")
    invisible(self)
  }
))

dani <- Person$new("Daniel", 29)
dani
```

```
## Person:
##  Name: Daniel
##  AGe:  29
```

Adding methods after creation

```
Person$set("public", "greet", function() {
  cat("Hello from ", self$name, "\n", sep = "")
})

# dani$greet()    # won't work (see above)
dani2 <- Person$new("Daniel", 29)
dani2$greet()
```

```
## Hello from Daniel
```

Inheritance

- Use the `inherit` argument
- Use `super$` in order to delegate to the **superclass**

- Any method which are not overridden will use the impolementation of the parent class.

```
AccumulatorChatty <- R6Class("AccumulatorChatty",
                             inherit = Accumulator,
                             public = list(
                               add = function(x = 1) {
                                 cat("Adding ", x, "\n", sep = "")
                                 super$add(x = x)
                               }
                             ))

x2 <- AccumulatorChatty$new()
x2$add(10)$add(1)$sum
```

```
## Adding 10
## Adding 1
```

```
## [1] 11
```

Introspection

- Use `class()` to deterime the class and all classes it inherits from
- Use `names()` to list all methods and fields

```
class(dani2)
```

```
## [1] "Person" "R6"
```

```
names(dani2)
```

```
## [1] ".__enclos_env__" "age"             "name"            "greet"
## [5] "clone"           "print"           "initialize"
```

```
class(x2)    # inherits from Accumulator
```

```
## [1] "AccumulatorChatty" "Accumulator"       "R6"
```

Controlling access

R6 has two other arguments (fields) that work similarly to public

- private -> only available within the class
  - Named list of methods (functions) and fields (everything else). -> as with public…
  - Use `privat$` instead of `self$`
  - Anything that's private can be more easily refactored because you knwo others aren't relying on it (because they can't access it)
- active -> define dynamic, or active, fields.
  - Look like fields from the outside, but are defined with functions, like methods
  - Uses **active bindings**
  - In particular useful in conjunction with private fields, because they make it possible to implement components that look like fields from the outside but provide additional checks
    - f.ex *read-only* or *ensure that* (see example below *)

```
# With this class definition of Person we can only set $age and $name during object creation
# -> we can not access their values from outside of the class!
Person <- R6Class("Person",
                  public = list(
```

```r
                    initialize = function(name, age = NA) {
                      private$name <- name
                      private$age <- age
                    },
                    print = function(...) {
                      cat("Person: \n")
                      cat("  Name: ", private$name, "\n", sep = "")
                      cat("  Age:  ", private$age, "\n", sep = "")
                    }
                  ),
                  private = list(
                    age = NA,
                    name = NULL
                  )
)

dani3 <- Person$new("Daniel")
dani3
```

```
## Person:
##   Name: Daniel
##   Age:  NA
```

```r
dani3$name    # no access!!
```

```
## NULL
```

```r
Rando <- R6Class("Rando", active = list(
  random = function(value) {
    if (missing(value)) {
      runif(1)
    } else {
      stop("Can't set `$random`", call. = FALSE)
    }
  }
))

# Better than this because error when trying to set the active field (not visible from
# outside that this is not a regular, i.e. public field...)

# Rando <- R6Class("Rando", active = list(
#   random = function() {
#       runif(1)
#   }
# ))

x <- Rando$new()
x$random
```

```
## [1] 0.2590919
```

```r
# x$random <- 5
```

*** Active fields in conjunction**

```
Person <- R6Class("Person",
  private = list(
    .age = NA,
    .name = NULL
  ),
  active = list(
    age = function(value) {
      if (missing(value)) {
        private$.age
      } else {
        stop("`$age` is read only", call. = FALSE)
      }
    },
    name = function(value) {
      if (missing(value)) {
        private$.name
      } else {
        stopifnot(is.character(value), length(value) == 1)
        private$.name <- value
        self
      }
    }
  ),
  public = list(
    initialize = function(name, age = NA) {
      private$.name <- name
      private$.age <- age
    }
  )
)

dani4 <- Person$new("Dani", 29)
dani4$name <- "Daniel"
# dani4$name <- 6    # error checking
# dani4$age <- 30    # read-only
```

Reference semantics

- Objects are not copied when modified
- If you want a copy -> use `$clone()`
  - Does not recursively clone nested R6 objects -> if you want, use `$clone(deep = TRUE)`
- It makes sense to think about when an R6 object is deleted, and you can write a `$finalize()` method to complement the `$initialize()`
- If one of the fileds is and R6 object, you must create it inside `$initialize()` not `R6Class()`!!

```
y1 <- Accumulator$new()
y2 <- y1

y1$add(10)
y2$sum
```

```
## [1] 10
```

```r
# But if you use clone()
y1 <- Accumulator$new()
y2 <- y1$clone()
y1$add(10)
y2$sum
```

```
## [1] 0
```

```r
# Finalize
TempFile <- R6Class("TempFile", list(
  path = NULL,
  initialize = function() {
    self$path <- tempfile()
  },
  finalize = function() {
    message("Cleaning up ", self$path)    # sends to stderr() connection
    unlink(self$path)
  }
))

tf <- TempFile$new()
tf$path
```

```
## [1] "/var/folders/01/c4pf64yn0d76mpp8bwgt_6m80000gn/T//Rtmp7rQv8y/file15c3d3581e1"
```

```r
rm(tf)
```