Chapter 14: R6 (part 2)

Advanced R Book Group, Cohort 3

Megan Stodel (@MeganStodel, www.meganstodel.com)

7 December, 2020

Controlling Access

Privacy

- In R6, you can define private fields/methods, which can only be accessed from within the class.
- Works similarly to the public argument - provide it with a named list of methods (functions) and fields.
- Available within methods using `private$` instead of `self$`.

Example

```
library(R6)

Person <- R6Class("Person",
  public = list(
    initialize = function(name, location = NULL) {
      private$name <- name
      private$location <- location
    },
    print = function(...) {
      cat("Person: \n")
      cat("  Name: ", private$name, "\n", sep = "")
      cat("  Location:  ", private$location, "\n", sep = "")
    }
  ),
  private = list(
    name = NULL,
    location = NULL
  )
)
```

Using it

You can't call `$age` or `$name` from outside the class.

```
megan <- Person$new(name = "Megan", location = "London")
megan
```

```
## Person:
##   Name: Megan
##   Location:  London
```

```
megan$name
```

```
## NULL
```

Why would you use this?

- Gives you control over what people can access.
- Easier to refactor in future because others won't be relying on it.
- (More important in other languages, because object hierarchies in R tend to be simpler)

Active Fields

- You can define components that look like fields from the outside, but are defined with functions.
- Each active binding is a function that takes the argument `value` - if missing, the value is being retrieved, otherwise it's being modified.

Example

Active field `random` returns a different value every time you access it.

```
Rando <- R6::R6Class("Rando", active = list(
  random = function(value) {
    if (missing(value)) {
      runif(1)
    } else {
      stop("Can't set `$random`", call. = FALSE)
    }
  }
))

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

```
## [1] 0.9629635
```

```
x$random
```

```
## [1] 0.5774945
```

Why would you use this?

Can provide additional checks in conjunction with private fields.

```
Person <- R6Class("Person",
  private = list(
    .name = NULL
  ),
  active = list(
    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) {
      private$.name <- name
    }
  )
```

Which looks like...

```
megan <- Person$new(name = "Megan")
megan$name
```

```
## [1] "Megan"
```

```
megan$name <- 10
```

```
## Error in (function (value) : is.character(value) is not TRUE
```

```
megan$name <- c("Megan", "Stodel")
```

```
## Error in (function (value) : length(value) == 1 is not TRUE
```

EXCERCISES

1. Create a bank account class that prevents you from directly setting the account balance, but you can still withdraw from and deposit to. Throw an error if you attempt to go into overdraft.

2. Create a class with a write-only $password field. It should have $check_password(password) method that returns TRUE or FALSE, but there should be no way to view the complete password.

3. Extend the Rando class with another active binding that allows you to access the previous random value. Ensure that active binding is the only way to access the value.

4. Can subclasses access private fields/methods from their parent? Perform an experiment to find out. [RAN OUT OF TIME FOR THIS]

Reference Semantics

What?

Objects are NOT copied when modified

```
y1 <- Accumulator$new()
y2 <- y1
y1$add(10)
c(y1 = y1$sum, y2 = y2$sum)
```

```
## y1 y2
## 10 10
```

If you want a copy, explicitly `$clone()` the objects.

```
y1 <- Accumulator$new()
y2 <- y1$clone()
y1$add(10)
c(y1 = y1$sum, y2 = y2$sum)
```

```
## y1 y2
## 10  0
```

Reasoning

Reference semantics makes it hard to reason about code.

```
x <- list(a = 1)
y <- list(b = 2)
z <- f(x, y)
```

For the vast majority of functions, you know that the final line only modifies z.

```
x <- List$new(a = 1)
y <- List$new(b = 2)
z <- f(x, y)
```

The final line is much harder to reason about: if `f()` calls methods of x or y, it might modify them as well as z.

This is the biggest potential downside of R6 and you should take care to avoid it by writing functions that either return a value, or modify their R6 inputs, but not both.

Finalizer

R6 objects are only deleted once, when finalized.

Think of `$finalize()` as a complement to `initialize()` - cleans up any resources created by the initializer.

This method will be run when it is deleted or when R exits.

Example

```
TemporaryFile <- R6Class("TemporaryFile", list(
  path = NULL,
  initialize = function() {
    self$path <- tempfile()
  },
  finalize = function() {
    message("Cleaning up ", self$path)
    unlink(self$path)
  }
))
```

R6 fields

If you use an R6 class as the default value of a field, it will be shared across all instances of the object. Therefore put it in `$initialize()` if you want it called fresh each time.

```
TemporaryDatabase <- R6Class("TemporaryDatabase", list(
  con = NULL,
  file = NULL,
  initialize = function() {
    self$file <- TemporaryFile$new()
    self$con <- DBI::dbConnect(RSQLite::SQLite(), path = file$path)
  },
  finalize = function() {
    DBI::dbDisconnect(self$con)
  }
))

db_a <- TemporaryDatabase$new()
db_b <- TemporaryDatabase$new()

db_a$file$path == db_b$file$path
```

```
## [1] FALSE
```

EXCERCISE

1. Create a class that allows you to write a line to a specified file. You should open a connection to the file in `$initialize()`, append a line using `cat()` in `$append_line()`, and close the connection in `$finalize()`.

R6 vs RC (Reference Classes)

- R6 is very similar to a built-in OO system called reference classes, or RC for short. I prefer R6 to RC because:
- R6 is much simpler. Both R6 and RC are built on top of environments, but while R6 uses S3, RC uses S4. This means to fully understand RC, you need to understand how the more complicated S4 works.
- R6 has comprehensive online documentation at https://r6.r-lib.org.
- R6 has a simpler mechanism for cross-package subclassing, which just works without you having to think about it. For RC, read the details in the "External Methods; Inter-Package Superclasses" section of ?setRefClass.
- RC mingles variables and fields in the same stack of environments so that you get (field) and set (field <<- value) fields like regular values. R6 puts fields in a separate environment so you get (self$field) and set (self$field <- value) with a prefix. The R6 approach is more verbose but I like it because it is more explicit.

Even more reasons

- R6 is much faster than RC. Generally, the speed of method dispatch is not important outside of microbenchmarks. However, RC is quite slow, and switching from RC to R6 led to a substantial performance improvement in the shiny package. For more details, see vignette("Performance", "R6").
- RC is tied to R. That means if any bugs are fixed, you can only take advantage of the fixes by requiring a newer version of R. This makes it difficult for packages (like those in the tidyverse) that need to work across many R versions.
- Finally, because the ideas that underlie R6 and RC are similar, it will only require a small amount of additional effort to learn RC if you need to.