R6 Object Oriented System

Stats 102A

Miles Chen based on Hadley Wickham's Advanced R

Department of Statistics

Week 5 Wednesday

UCLA

Section 1

R6 Object Oriented System

R6 Object Oriented System

R6 uses the encapsulated OOP paradigm.

This means methods belong to objects (vs. generic functions) and you call them using `object$method()`

R6 objects are **mutable** meaning they are modified in place and have reference semantics.

library(R6)

You will need to install and load the R6 package to use R6.

```
# install.packages("R6")
library(R6)
```

Section 2

Classes and Methods

Creating an R6 object

Creation of a new object class in R6 is done with the function `R6Class()`

This is the only function from the R6 package that you will use.

The function will need at least two arguments:

- `classname`: This is the name of class you are creating. By convention R6 class names have `UpperCamelCase`
- `public`: This is a list of methods and fields that belong to the object. The methods and fields use `snake_case`

You should always assign the result of R6Class() into a variable with the same name as the class. The result will be an R6 object that defines the class.

Simple R6 class Example

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

This creates a very simple class called "Accumulator". It has one public field, `sum` which has an initial value of 0. It has one public method `add()`, which adds `x` to the `sum`.

Simple R6 class Example

```
Accumulator
```

```
## <Accumulator> object generator
##   Public:
##     sum: 0
##     add: function (x = 1)
##     clone: function (deep = FALSE)
##   Parent env: <environment: R_GlobalEnv>
##   Locked objects: TRUE
##   Locked class: FALSE
##   Portable: TRUE
```

Using the new class

You can create new objects of the class by calling the `$new()` method.

```
x <- Accumulator$new()
x
```

```
## <Accumulator>
##   Public:
##     add: function (x = 1)
##     clone: function (deep = FALSE)
##     sum: 0
```

You can access values and methods with thr `$`.

```
x$sum
```

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

Using the class

```
x$sum
```

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

```
x$add() # default value of 1
x$sum
```

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

```
x$add(3)
x$sum
```

```
## [1] 4
```

return invisible(self)

Let's revisit our definition of the `Accumulator` class. Note that the method `$add()` returns `invisible(self)`.

```
Accumulator <- R6Class(
  ...
    add = function(x = 1) {
      self$sum <- self$sum + x
      invisible(self)
    }
  ...
```

This is a recommended practice for R6 methods. The object itself is returned, and allows methods to be chained together.

The `invisible()` means that even though the object is returned, it is not printed out.

Method Chaining

```
x$sum
```

```
## [1] 4
```

```
x$add(10)$add(10)$sum
```

```
## [1] 24
```

In the above code, we call the `$add()` method in `x$add(10)`.

Because the method returns `invisible(self)`, this call returns the object itself after running the `$add()` method. This means we can call the `$add()` method again on the resulting object, which we do.

Finally, we ask for the field `$sum`, and we see the value that sum has.

Method Chaining - Readability

For readability, it is sometimes recommended that chained methods be put on their own line.

```
x$sum
```

```
## [1] 24
```

```
x$
  add(10)$
  add(10)$
  sum
```

```
## [1] 44
```

The $initialize() method

The $initialize() method will override the default behavior of $new(). It allows you define required fields when creating a new object and to perform some checks to make sure the inputs are valid.

Here is an example from Hadley Wickham where he defines a new R6 class person. The class has two fields, $name and $age. The $initialize() method checks to make sure name and age are length 1 vectors with values of the appropriate type.

```
Person <- R6Class(
  classname = "Person",
  public = list(
    name = NULL,
    age = NA,
    initialize = function(name, age = NA) {
      stopifnot(is.character(name), length(name) == 1)
      stopifnot(is.numeric(age), length(age) == 1)
      self$name <- name
      self$age <- age
    }
))
```

$initialize() overrides $new()

```
hadley <- Person$new("Hadley", age = "thirty-eight")
```

```
## Error in .subset2(public_bind_env, "initialize")(...): is.numeric(age) is n
```

```
hadley <- Person$new("Hadley", age = 38)
```

The $print() method

The default way R6 objects are printed may not be desireable. When you create an R6 object and type its name into the console, R prints out the values according to the default R6 method.

```
hadley
```

```
## <Person>
##   Public:
##     age: 38
##     clone: function (deep = FALSE)
##     initialize: function (name, age = NA)
##     name: Hadley
```

You can define a `$print()` method in the class definition to change this behavior.

The $print() method

```
Person <- R6Class(
  classname = "Person",
  public = list(
    name = NULL,
    age = NA,
    initialize = function(name, age = NA) {
      stopifnot(is.character(name), length(name) == 1)
      stopifnot(is.numeric(age), length(age) == 1)
      self$name <- name
      self$age <- age
    },
    print = function(...) {
      cat("Person: \n")
      cat("  Name:", self$name, "\n")
      cat("  Age:", self$age, "\n")
      invisible(self)
    }
))
```

The $print() method

```
hadley2 <- Person$new("Hadley", 38)
hadley2
```

```
## Person:
##   Name: Hadley
##   Age: 38
```

Methods are bound to the R6 objects

The first instance of `hadley` was created with an R6 class that **did not** have the `$print()` method defined. `hadley2` was created with an R6 class that **did** have the `$print()` method defined. The methods are bound to the objects created, so the old `hadley` object does not get the new `$print()` method.

```
hadley2
```

```
## Person:
##   Name: Hadley
##   Age: 38
```

```
hadley
```

```
## <Person>
##   Public:
##     age: 38
##     clone: function (deep = FALSE)
##     initialize: function (name, age = NA)
##     name: Hadley
```

Methods are bound to the R6 objects

There is no relationship between `hadley` and `hadley2`. They just happen to share the same class name.

As you are working in R to develop a class definition, having multiple objects that share the same class name but have different class defintions can become confusing. Be sure to delete objects with the old class definition and redefine them with the new class definition.

Adding methods after creation

It is possible to modify the fields and methods of an existing class. You can use the `$set()` method to achieve this.

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

As before, new methods and fields are only available to new objects. They are not retrospectively added to existing objects.

Inheritance

You can define new classes that inherit the fields and methods of another class.

When defining the new class, you specify the superclass from which the new class inherits with `inherit = superclass`

```
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
```

Inheritance - a closer look

In our definition of `AccumulatorChatty`, we define a method `$add()`. This overrides the `$add()` method in the superclass `Accumulator`.

You'll notice that in the definition of this method, we add a line with `cat()` to announce what value we are adding. The final line, however makes a call to `super$add(x = x)`. This calls the original `$add()` method in the original `Accumulator` class (which is the `superclass` relative to `AccumulatorChatty`. (That function added `x` to `self$sum` and returned `invisible(self)`).

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

Introspection

When you call `class()` on an R6 object, it will return the vector of classes that it inherits from.

```
class(hadley)
```

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

```
class(hadley2)
```

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

```
class(x2)
```

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

Keep in mind the `Person` class definition of `hadley` is different from the `Person` class definition of `hadley2`. Again, best practice is to delete old objects if you redefine the class definition.

Introspection

You can also see the methods and fields available for an object with `names()`

```
names(hadley) # old Person class does not have $print()
```

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

```
names(hadley2)
```

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

```
names(x2)
```

```
## [1] ".__enclos_env__" "sum"             "clone"           "add"
```

Section 3

Reference Semantics

Most of R uses copy-on-modify

In most of R, objects are copied when they are modified. In the following code:

- We bind the name a to a value.
- We then bind the name b to the same object as a.
- When we modify b, R creates a copy of the object and modifies it.
- Modifying b does not affect a.

```
a <- c(1, 2)
b <- a
b[1] <- b[1] + 10
a
```

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

```
b
```

```
## [1] 11  2
```

Reference Semantics

R6 objects have reference semantics. Objects are not copied when we modify them.

- We bind the name `x` to an R6 `Accumulator` object.
- We then bind the name `y` to the same object as `x`.
- When we modify `y` by calling `$add()`, we modify the R6 object, not a copy.
- Names `x` and `y` are bound to the same object, so `x$sum` and `y$sum` show the same value.

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

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

```
y$sum
```

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

Cloning

If you want a copy of an object, you must explicity create a copy using `$clone()`

- We bind the name `x` to an R6 `Accumulator` object.
- We then bind the name `y` to a clone of the object `x`.
- When we call `y$add()`, we modify `y` which is a clone of `x`.
- Modifying `y` changes `y` only and not `x`.

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

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

```
y$sum
```

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

Reference Semantics can be tricky

Reference Semantics can be tricky and should be handled with care.

For example, look at this simple example using base R objects:

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

We know that the only object that will be modified by function `f()` is the object `z`

Reference Semantics can be tricky

Now look at this example with an imaginary R6 class `List`.

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

The function `f()` uses R6 objects `x` and `y` as inputs. Within the function, there could be calls to methods that modify the objects `x` and `y` directly. The function also seems to return a value that will be assigned to the name `z`.

Exactly what gets modified in this example is less clear.

Tips

When using R6, one recommendation is that your functions that interact with R6 objects should either:

- return a value but not modify the input R6 object
- modify the input R6 object but not return a value

This will generally make your code easier to read and understand.

Example

```
x <- Accumulator$new()
add5tosum <- function(accum){ # modifies x
  accum$add(5)
}
add5tosum(x) # x is modified in place. There is no assignment operator
x$sum
```

```
## [1] 5
```

```
sumtimes10 <- function(accum){ # returns a value but does not modify
  accum$sum * 10
}
y <- sumtimes10(x) # returned values must be assigned somewhere to be used
y
```

```
## [1] 50
```

```
x$sum
```

```
## [1] 5
```

Section 4

Public, Private, Active

Public, Private, Active

The following content will not be tested in the midterm exam.

It is also not necessary to know to complete the homework assignment.

More can be learned at https://adv-r.hadley.nz/r6.html and at https://r6.r-lib.org/articles/Introduction.html

Public vs Private

When you define a class with `R6Class()`, you generally provide a list of fields and methods into the argument `public`.

You can also create a list of fields and methods in `private`. These values are available only inside the object, and not from the outside. Please note that private does not mean secret. It simply means that these values cannot be directly accessed or modified from outside the object.

Revisit the Person definition

When we defined the R6 class Person, we placed the $name and $age fields into public. This allows the values of name and age directly accessible from the outside.

```
Person <- R6Class(
  classname = "Person",
  public = list(
    name = NULL,
    age = NA,
 ...
```

```
hadley2$age # we can see the public fields directly
```

```
## [1] 38
```

```
hadley2$age <- 25 # public fields can be modified directly
hadley2$age
```

```
## [1] 25
```

A private person

This time, we keep $age and $name fields in private. $initialize() and print() are still in public

We access these private values with private$ instead of self$

```
Person <- R6Class("Person",
  public = list(
    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
  )
)
```

A private person

```
hadley3 <- Person$new("Hadley", age = 38)
hadley3$age # private fields are not directly visible from the outside
```

```
## NULL
```

```
hadley3$age <- 25 # private fields cannot be modified directly
```

```
## Error in hadley3$age <- 25: cannot add bindings to a locked environment
```

```
hadley3 # the values do exist and made visible in the $print() method
```

```
## Person:
##   Name: Hadley
##   Age:  38
```

Active fields

Active fields allow you to define something that looks like a public field, but is actually defined by a function.

The active field is created by defining a function that takes a single argument `value`.

- If the `value` is `missing()`, then the value is being retrieved.
- If the value is not `missing()`, then it is being modified.

Active Field Example

We define a class `Rando` with a single active field `random`. The value of `random` is generated by a function that calls `runif(1)`

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

Active Field Example

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

```
## [1] 0.3584381
```

```
x$random
```

```
## [1] 0.5111816
```

```
x$random
```

```
## [1] 0.3925567
```

Active Fields in Practice

Active fields are particularly 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.

For example, we can use them to make a read-only age field and to ensure that name is a length 1 character vector.

Active Field example

```
Person <- R6Class(
  classname = "Person",
  public = list(
    initialize = function(name, age = NA) {
      private$.name <- name
      private$.age <- age
    }
  ),
  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
      }
    }
  )
)
```

A few notes

We define `$initialize()` in public, so that when we call `$new()` we will produce new objects as specified in initialize.

The values for age and name are kept in `private`. They are named `$.age` and `$.name`. This is to distinguish them from the values of `$age` and `$name` that are available in the list of active fields.

Inside `active`, we define active fields `$age` and `$name`.

The field `$age` is a function with argument `value`. If no value is supplied, the function returns the value of `$.age` which is stored in `private`. If avalue‘ is supplied, the function returns an error saying that age is read only.

The field `$name` is a function with argument `value`. If no value is supplied, the function returns the value of `$.name` which is stored in `private`. If a `value` is supplied, the function checks to see if it is valid. If it is not valid, there is an error. If it is valid, it modifies the value of `private$.name` with the supplied value.

Active Fields in Practice

Note that it looks like `hadley4$name` is a public field that we can access and modify directly. Each time we ask for `hadley4$name`, we are running the function in `active` (with `missing(value)`) which looks up the `private$.name` value. When we assign "Wickham" to `hadley4$name`, we run the function in `active` with `value = Wickham`, which performs some checks to make sure the entry is valid.

```
hadley4 <- Person$new("Hadley", age = 38)
hadley4$name
```

```
## [1] "Hadley"
```

```
hadley4$name <- "Wickham" # "Wickham" is the 'value' supplied to the function
hadley4$name
```

```
## [1] "Wickham"
```

Active Fields in Practice

```
hadley4$name <- 10 # 10 causes an error in the function
```

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

```
hadley4$name
```

```
## [1] "Wickham"
```

```
hadley4$age
```

```
## [1] 38
```

```
hadley4$age <- 20
```

```
## Error: `$age` is read only
```

A note about S4 and RC

Two other Object Oriented Programming system exists in R: **S4** and **RC**

S4 is similar to S3 in that it uses generic functions. S4 formalizes class definitions. You are no longer allowed to simply create an object and change its class on the fly. Object classes have formal definitions, and trying to create an object that doesn't fit the definition will produce an error.

RC is similar to R6 in that it creates an encapsulated OOP system. It is built on top of S4 objects and thus requires an understanding of S4 before you can use it.