

# Package ‘R6’

August 19, 2014

**Title** Classes with reference semantics

**Version** 2.0

**Description** The R6 package allows the creation of classes with reference semantics, similar to R's built-in reference classes. Compared to reference classes, R6 classes are simpler and lighter-weight, and they are not built on S4 classes so they do not require the methods package. These classes allow public and private members, and they support inheritance, even when the classes are defined in different packages.

**Depends** R (>= 3.0)

**Suggests** knitr, pryr, testthat,

**License** MIT + file LICENSE

**URL** <https://github.com/wch/R6/>

**LazyData** true

**VignetteBuilder** knitr

**Author** ``Winston Chang" [aut, cre]

**Maintainer** ``Winston Chang" <winston@stdout.org>

**NeedsCompilation** no

**Repository** CRAN

**Date/Publication** 2014-08-19 07:54:18

## R topics documented:

R6Class	2
Index	8

R6Class

*Create an R6 reference object generator***Description**

R6 objects are essentially environments, structured in a way that makes them look like an object in a more typical object-oriented language than R. They support public and private members, as well as inheritance across different packages.

**Usage**

```
R6Class(classname = NULL, public = list(), private = NULL,
        active = NULL, inherit = NULL, lock = TRUE, class = TRUE,
        portable = TRUE, parent_env = parent.frame())
```

**Arguments**

classname	Name of the class. The class name is useful primarily for S3 method dispatch.
public	A list of public members, which can be functions (methods) and non-functions (fields).
private	An optional list of private members, which can be functions and non-functions.
active	An optional list of active binding functions.
inherit	A R6ClassGenerator object to inherit from; in other words, a superclass.
portable	If TRUE (the default), this class will work with inheritance across different packages. Note that when this is enabled, fields and members must be accessed with <code>self\$x</code> or <code>private\$x</code> ; they can't be accessed with just <code>x</code> .
parent_env	An environment to use as the parent of newly-created objects.
class	Should a class attribute be added to the object? Default is TRUE. If FALSE, the objects will simply look like environments, which is what they are.
lock	Should the environments of the generated objects be locked? If locked, new members can't be added to the objects.

**Details**

An R6 object consists of a public environment, and may also contain a private environment, as well as environments for superclasses. In one sense, the object and the public environment are the same; a reference to the object is identical to a reference to the public environment. But in another sense, the object also consists of the fields, methods, private environment and so on.

The `active` argument is a list of active binding functions. These functions take one argument. They look like regular variables, but when accessed, a function is called with an optional argument. For example, if `obj$x2` is an active binding, then when accessed as `obj$x2`, it calls the `x2()` function that was in the `active` list, with no arguments. However, if a value is assigned to it, as in `obj$x2 <- 50`, then the function is called with the right-side value as its argument, as in `x2(50)`. See [makeActiveBinding](#) for more information.

If the public or private lists contain any items that have reference semantics (for example, an environment), those items will be shared across all instances of the class. To avoid this, add an entry for that item with a NULL initial value, and then in the `initialize` method, instantiate the object and assign it.

### The print method

R6 object generators and R6 objects have a default `print` method to show them on the screen: they simply list the members and parameters (e.g. `lock`, `portable`, etc., see above) of the object.

The default `print` method of R6 objects can be redefined, by supplying a public `print` method. (print members that are not functions are ignored.) This method is automatically called whenever the object is printed, e.g. when the object's name is typed at the command prompt, or when `print(obj)` is called. It can also be called directly via `obj$print()`. All extra arguments from a `print(obj, ...)` call are passed on to the `obj$print(...)` method.

### Portable and non-portable classes

When R6 classes are portable (the default), they can be inherited across packages without complication. However, when in portable mode, members must be accessed with `self` and `private`, as in `self$x` and `private$y`.

When used in non-portable mode, R6 classes behave more like reference classes: inheritance across packages will not work well, and `self` and `private` are not necessary for accessing fields.

### S3 details

Normally the public environment will have two classes: the one supplied in the `classname` argument, and "R6". It is possible to get the public environment with no classes, by using `class=FALSE`. This will result in faster access speeds by avoiding class-based dispatch of `$`. The benefit is negligible in most cases.

If a class is a subclass of another, the object will have as its classes the `classname`, the superclass's `classname`, and "R6"

The primary difference in behavior when `class=FALSE` is that, without a class attribute, it won't be possible to use S3 methods with the objects. So, for example, pretty printing (with `print.R6Class`) won't be used.

### Examples

```
# A queue -----
Queue <- R6Class("Queue",
  public = list(
    initialize = function(...) {
      for (item in list(...)) {
        self$add(item)
      }
    },
    add = function(x) {
      private$queue <- c(private$queue, list(x))
      invisible(self)
    },
```

```

    remove = function() {
      if (private$length() == 0) return(NULL)
      # Can use private$queue for explicit access
      head <- private$queue[[1]]
      private$queue <- private$queue[-1]
      head
    }
  ),
  private = list(
    queue = list(),
    length = function() base::length(private$queue)
  )
)

q <- Queue$new(5, 6, "foo")

# Add and remove items
q$add("something")
q$add("another thing")
q$add(17)
q$remove()
#> [1] 5
q$remove()
#> [1] 6

# Private members can't be accessed directly
q$queue
#> NULL
# q$length()
#> Error: attempt to apply non-function

# add() returns self, so it can be chained
q$add(10)$add(11)$add(12)

# remove() returns the value removed, so it's not chainable
q$remove()
#> [1] "foo"
q$remove()
#> [1] "something"
q$remove()
#> [1] "another thing"
q$remove()
#> [1] 17

# Active bindings -----
Numbers <- R6Class("Numbers",
  public = list(
    x = 100
  ),
  active = list(
    x2 = function(value) {
      if (missing(value)) return(self$x * 2)

```

```

        else self$x <- value/2
      },
      rand = function() rnorm(1)
    )
  )

n <- Numbers$new()
n$x
#> [1] 100
n$x2
#> [1] 200
n$x2 <- 1000
n$x
#> [1] 500

# If the function takes no arguments, it's not possible to use it with <-:
n$rand
#> [1] 0.2648
n$rand
#> [1] 2.171
# n$rand <- 3
#> Error: unused argument (quote(3))

# Inheritance -----
# Note that this isn't very efficient - it's just for illustrating inheritance.
HistoryQueue <- R6Class("HistoryQueue",
  inherit = Queue,
  public = list(
    show = function() {
      cat("Next item is at index", private$head_idx + 1, "\n")
      for (i in seq_along(private$queue)) {
        cat(i, ": ", private$queue[[i]], "\n", sep = "")
      }
    },
    remove = function() {
      if (private$length() - private$head_idx == 0) return(NULL)
      private$head_idx <<- private$head_idx + 1
      private$queue[[private$head_idx]]
    }
  ),
  private = list(
    head_idx = 0
  )
)

hq <- HistoryQueue$new(5, 6, "foo")
hq$show()
#> Next item is at index 1
#> 1: 5
#> 2: 6
#> 3: foo
hq$remove()

```

```

#> [1] 5
hq$show()
#> Next item is at index 2
#> 1: 5
#> 2: 6
#> 3: foo
hq$remove()
#> [1] 6

# Calling superclass methods with super$ -----
CountingQueue <- R6Class("CountingQueue",
  inherit = Queue,
  public = list(
    add = function(x) {
      private$total <<- private$total + 1
      super$add(x)
    },
    get_total = function() private$total
  ),
  private = list(
    total = 0
  )
)

cq <- CountingQueue$new("x", "y")
cq$get_total()
#> [1] 2
cq$add("z")
cq$remove()
#> [1] "x"
cq$remove()
#> [1] "y"
cq$get_total()
#> [1] 3

# Non-portable classes -----
# By default, R6 classes are portable, which means they can be inherited
# across different packages. Portable classes require using self$ and
# private$ to access members.
# When used in non-portable mode, members can be accessed without self$,
# and assignments can be made with <<-.

NP <- R6Class("NP",
  portable = FALSE,
  public = list(
    x = NA,
    getx = function() x,
    setx = function(value) x <<- value
  )
)

```

```
np <- NP$new()
np$setx(10)
np$getx()
#> [1] 10

# Setting new values -----
# It is possible to add new members to the class after it has been created,
# by using the $set() method on the generator.

Simple <- R6Class("Simple",
  public = list(
    x = 1,
    getx = function() x
  )
)

Simple$set("public", "getx2", function() self$x*2)

# Use overwrite = TRUE to overwrite existing values
Simple$set("public", "x", 10, overwrite = TRUE)

s <- Simple$new()
s$x
s$getx2()
```

# Index

`makeActiveBinding`, [2](#)

`R6 (R6Class)`, [2](#)

`R6Class`, [2](#)