Object oriented programming

Hadley Wickham

Assistant Professor / Dobelman Family Junior Chair
Department of Statistics / Rice University

June 2011

1. Motivation
2. S3
3. S4
4. R5 (reference classes)
5. Other OO styles

Motivation

```
mean(1:10)
mean(mtcars)

# What does mean do?
mean
```

```
sd <- function (x, na.rm = FALSE) {
    if (is.matrix(x))
        apply(x, 2, sd, na.rm = na.rm)
    else if (is.vector(x))
        sqrt(var(x, na.rm = na.rm))
    else if (is.data.frame(x))
        sapply(x, sd, na.rm = na.rm)
    else sqrt(var(as.vector(x), na.rm = na.rm))
}

# What if you want to create an object where
# sd is created in a different way?
```

Motivation

- Understanding more code
- Extensibility
- Programming “in the large”

- Focus on S3, then differences to S4. Overview of R5. Summary of contributed OO approaches.

S3

Key points

Generic function style of OO.

No formal class definition: no definition of what fields or class hierarchy. Class attribute determines class of object.

Naming convention + `UseMethod()` used to find appropriate methods.

Super simple, but ad hoc, and many inconsistencies. Most common OO in R.

Challenge

Develop a class for numeric vectors that remembers its range (like factors do)

Will extend a numeric vector to add to attributes: `min` and `max`

```
# Structure function takes vector and adds attributes
# class attribute determines S3 class
structure(1:10, min = 0, max = 10,
  class = ”minmax”)

# Customary to create convenience function to create
# objects of specific class
minmax <- function(x, minx = min(x), maxx = max(x)) {
  stopifnot(is.numeric(x))

  structure(x, min = minx, max = maxx,
    class = ”minmax”)
}
minmax(1:10)
```

```
# Also customary to create function to test if
# an object is of that class:
is.minmax <- function(x) {
  inherits(x, ”minmax”)
}
is.minmax(minmax(1:10))
```

```
# First method is usually a print method. Always
# look at the generic first so that you can match
# the arguments correctly.

print
# Can tell it’s a generic method because it uses
# UseMethod

# Methods follow simple naming scheme
print.minmax <- function(x, ...) {
  print.default(as.numeric(x))
  cat(”Range: [”, attr(x, ”min”), ”, ”,
    attr(x, ”max”), ”]\n”, sep = “”)
}
minmax(1:10)
# Only time it’s ok to call a method directly
```

Generic functions

Methods are associated with functions, not classes.

Methods are associated with functions, not classes.

Methods are associated with functions, not classes.

```
# No checks for object correctness, so easy to abuse

mod <- glm(log(mpg) ~ log(disp), data = mtcars)
class(mod)
class(mod) <- ”lm”
mod

class(mod) <- ”table”
mod

# But surprisingly, this doesn’t cause that
# many problems - instead of the language enforcing
# certain properties you need to do it yourself
```

Your turn

What's wrong with the following code?

```
minmax(1:10, max = 5)
```

Modify `minmax` to prevent it from occurring.

```
minmax <- function(x, minx = min(x), maxx = max(x)) {
  stopifnot(is.numeric(x))
  stopifnot(all(minx <= x))
  stopifnot(all(maxx >= x))

  structure(x, min = minx, max = maxx,
    class = ”minmax”)
}
minmax(1:10, max = 5)
```

```
a <- minmax(1:10, max = 20)

max(a)
min(a)
range(a)

# Need to add methods for these generic functions

max
min
range

# How do you know if a function is generic?
#  * includes UseMethod (like print)
#  * is primitive or internal and listed in:
#    * ?S3groupGeneric
#    * ?InternalMethods
```

```
max.minmax <- function(..., na.rm = FALSE) {
  parts <- list(...)
  if (length(parts) == 1) {
    attr(parts[[1]], ”max”)
  } else {
    stop(”Maximum of more than one minmax not”,
      ”implemented”)
  }
}
```

Your turn

Add method for `min`. Does `range` work as expected? If not, fix it.

Extend the function to work with any number of inputs.

```
max.minmax <- function(..., na.rm = FALSE) {
  parts <- list(...)
  if (length(parts) == 1) {
    attr(parts[[1]], ”max”)
  } else {
    max(vapply(parts, ”min”, numeric(1)))
  }
}
min.minmax <- function(..., na.rm = FALSE) {
  parts <- list(...)
  if (length(parts) == 1) {
    attr(parts[[1]], ”min”)
  } else {
    min(vapply(parts, ”min”, numeric(1)))
  }
}
range.minmax <- function(..., na.rm = FALSE) {
  c(min(..., na.rm = TRUE), max(..., na.rm = TRUE))
}
```

```
a <- minmax(1:10, max = 20)
a[1:5]

# Always need to locate the generic so you can
# figure out what the arguments are.  This is
# sometimes hard!

match.fun(”[”)
?”[”

# In this case we can punt, and allow the parent
# method to do the hard work
”[.minmax” <- function(x, ...) {
  minmax(NextMethod(), minx = attr(x, ”min”),
    maxx = attr(x, ”max”))
}
```

```
# Storing S3 objects in a data frame requires a
# method for as.data.frame.

df <- data.frame(a = a)

as.data.frame.minmax <- function(x, ...) {
  structure(list(x),
    row.names = seq_along(x),
    class = ”data.frame”)
}
df <- data.frame(a = a)
df[1:5, ”a”]
```

```
a <- minmax(1:10)
b <- minmax(1:5, max = 20)
a + b
a + 3
3 + a

match.fun(”+”)
”+.minmax” <- function(e1, e2) {
  minmax(NextMethod(), min = min(e1) + min(e2),
    max = max(e1) + max(e2))
}
a + b
a + 3
3 + a
```

Inheritance

`NextMethod()` strips the first element off the class vector and then re-calls the generic with the same arguments.

Confusing here because it looks like there is only one element in the class vector. But: `class(unclass(minmax(1:10)))`

```
# Creating your own generics
mean2 <- function (x, ...) {
  UseMethod(”mean2”, x)
}

# Methods follow a simple naming convention
mean2.numeric <- function(x, ...) sum(x) / length(x)
mean2.data.frame <- function(x, ...)
  sapply(x, mean, ...)
mean2.matrix <- function(x, ...) apply(x, 2, mean)

# Bad practice to call methods directly
# Finds all methods for the mean2 generic:
# mean2.*
methods(”mean2”)

# Find all methods associated with matrix class
# *.matrix
methods(class = ”matrix”)
```

Namespacing

In Java/C#/Ruby/Python etc., often have many small methods, even if only used by one class.

This is not useful in R – only useful to define methods that are used by multiple classes.

Use namespaces (tomorrow) for the equivalent encapsulation.

S4

Key points

Same basic style as S3, but formal and rigorous (and verbose).

`setClass()` defines classes.

`setGeneric()` defines generic functions.

`setMethod()` defines methods.

Your turn

Read through `3-S4.r`. Compare and contrast S3 to S4.

S3	S4
UseMethod	setGeneric / standardGeneric
NextMethod	callNextMethod
methods	findMethods

Tips

S4 supports multiple inheritance and multiple dispatch - but don’t use both. Method dispatch becomes extremely complex.

See example in ?"?" for getting help on S3 methods

Keep it simple!

Learning more

?setClass ?setMethod

http://www.ci.tuwien.ac.at/Conferences/useR-2004/Keynotes/Leisch.pdf

http://www.bioconductor.org/help/course-materials/2011/AdvancedRFeb2011Seattle/

Chapter 9 in “Software for Data Analysis”, by John Chambers

R5

Key points

Class-based (message passing) OO.
Much closer to Java/C#/Python/Ruby etc.

Have mutable state.

Still under active development.

Currently all methods/fields are public.

```
Range <- setRefClass("Range", fields = "range",
  methods = list(
    initialize = function() {
      initFields(range = NULL)
    },
    reset = function() range <<- NULL
  )
)

ContinuousRange <- setRefClass(
  "Continuous", contains = "Range",
  methods = list(
    train = function(x) range <<- train_continuous(x, range)
  )
)

DiscreteRange <- setRefClass(
  "DiscreteRange", contains = "Range",
  methods = list(
    train = function(x, drop) range <<- train_discrete(x, range, drop)
  )
)
```

```
library(scales)

r1 <- ContinuousRange$new()
r1$train(1:10)
r1$range
r1$train(100)
r1$range
r1$reset()
r1$range
```

Key points

- Works much like a list of functions. Use `$` to access fields and methods
- In methods, use `<<-` to modify fields.

Tips

Use R5 classes only for components that really need mutable state. Use S3/S4 for everything else.

Others

Packages

- proto
- mutatr
- R.oo
- OOP
- ofp, s3x