Advanced R - Hadley Wickham
Chapter 12 and 13
OO programming: Base types and S3 objects

Alejandra Hernandez

1/09/2020

Welcome!

- This book club is a joint effort between RLadies Nijmegen, Rotterdam, 's-Hertogenbosch (Den Bosch), Amsterdam and Utrecht
- We meet every 2 weeks to go through a chapter
- Use the HackMD to present yourself, ask questions and see your breakout room
- We split in breakout rooms after the presentation, and we return to the main jitsi link after 20 min
- There are still possibilities to present a chapter :) Sign up at rladiesnl.github.io/book_club

Object-Oriented Programming

- Multiple OOP systems!
  - Today we focus on S3: important for base R!
  - Others: R6, S4, RC, R.oo, proto...
- Why use OOP?
  - Making functions more versatile (same function for different input)
  - Easier to contribute with new *methods*

```
x_vector <- c(1:15)
print(x_vector)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

```
x_matrix <- matrix(1:15, nrow = 5)
print(x_matrix)
```

```
     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8   13
[4,]    4    9   14
[5,]    5   10   15
```

Important vocabulary for OOP

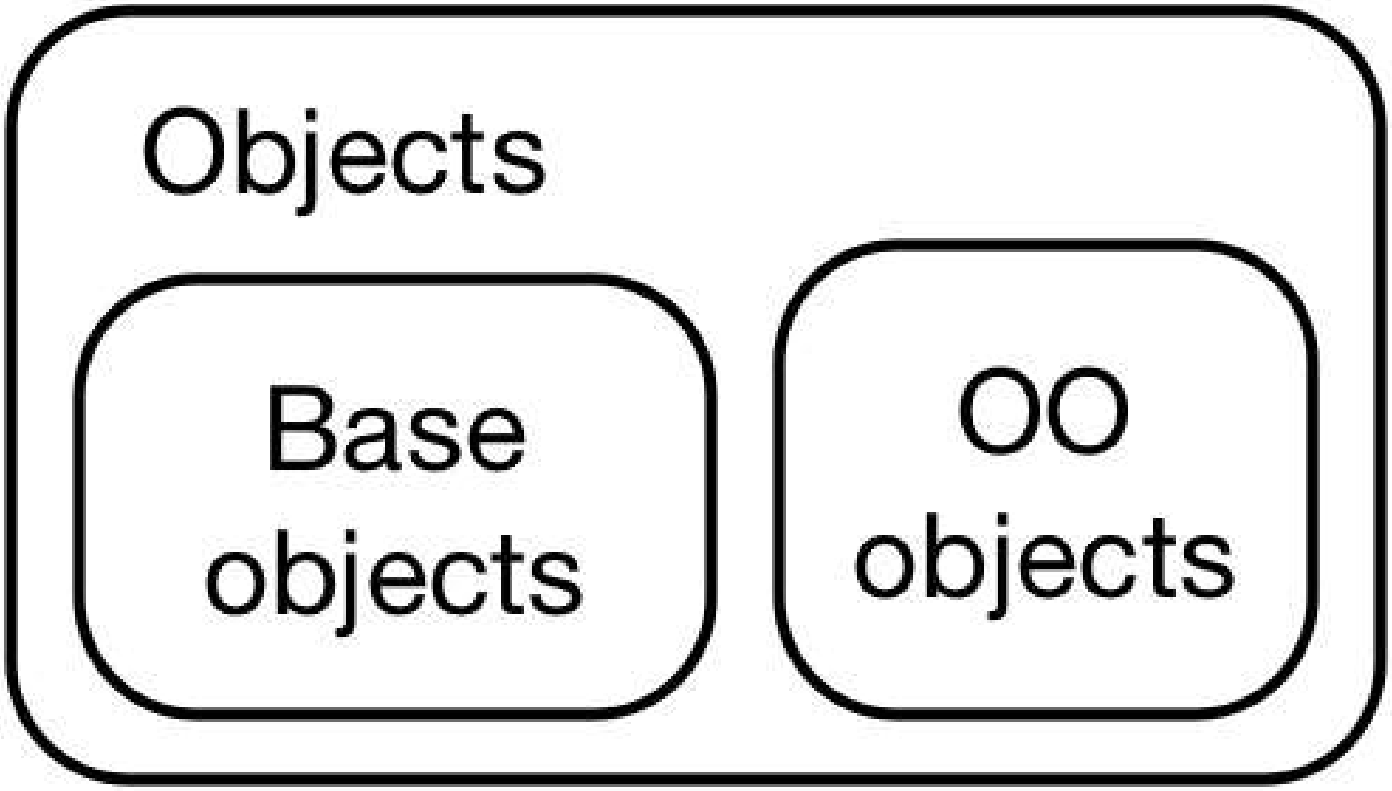
- **Types of OOP**
  - **Encapsulated OOP:** methods belong to objects or classes, and method calls typically look like object.method(arg1, arg2)
  - **Functional OOP:** methods belong to generic functions, and method calls look like ordinary function calls: generic(object, arg2, arg3) **<- S3**
- **Generic Function:** the function you actually call
- **Method:** the implementation of a generic function for the specific class of your object
- **Method dispatch:** process of finding the correct method given a class

Let's start!!

Make sure you have installed and loaded `library(sloop)`

Chapter 12: Base Types

Objects in R



How to find out if an object is Base or OO?

Base object:

```
sloop::otype(1:10)
```

```
[1] "base"
```

```
attr(1:10, "class")
```

```
NULL
```

OO object

```
sloop::otype(mtcars)
```

```
[1] "S3"
```

```
attr(mtcars, "class")
```

```
[1] "data.frame"
```

Base Types

Every object has a base type:

```
typeof(1:10)
```

```
[1] "integer"
```

```
typeof(mtcars)
```

```
[1] "list"
```

> Base types do not form an OOP system because functions that behave differently for different base types are primarily written in C code that uses switch statements. This means that only R-core can create new types

What are the base types? (1/2)

- Vectors:
  - NULL (NILSXP), logical (LGLSXP), integer (INTSXP), double (REALSXP)**, complex (CPLXSXP), character (STRSXP), list (VECSXP), and raw (RAWSXP).
- Functions:
  - Closure (regular R functions, CLOSXP), special (internal functions, SPECIALSXP), and builtin (primitive functions, BUILTINSXP).
- Environments have type environment (ENVSXP).

** Careful with the *"numeric"* type! It can mean many things in R!

What are the base types? (2/2)

- S4 type (S4SXP) is used for S4 classes that don't inherit from an existing base type
- Language components
  - Symbol (aka name, SYMSXP), language (usually called calls, LANGSXP), and pairlist (used for function arguments, LISTSXP) types.
- Expression (EXPRSXP) is a special purpose type that's only returned by parse() and expression(). Expressions are generally not needed in user code.
- Others
  - Externalptr (EXTPTRSXP), weakref (WEAKREFSXP), bytecode (BCODESXP), promise (PROMSXP), ... (DOTSXP), and any (ANYSXP).

Chapter 13: S3

S3 objects

- Most commonly used system in CRAN packages
- Very flexible! (good and bad)
- At least a class attribute

```
f <- factor(c("a", "b", "c"))

typeof(f)
```

```
[1] "integer"
```

```
attributes(f)
```

```
$levels
[1] "a" "b" "c"

$class
[1] "factor"
```

How does the S3 object 'f' actually look like?

You can remove the "special" behavior of a class:

```
f
```

```
[1] a b c
Levels: a b c
```

```
unclass(f)
```

```
[1] 1 2 3
attr(,"levels")
[1] "a" "b" "c"
```

Classes

Not as formal as in other languages

```
# Create and assign class in one step
x <- structure(list(), class = "my_class")

# Create, then set class
x <- list()
class(x) <- "my_class"
```

Recommendations:

- The class name can be any string, but better to use only letters and _
- When using a class in a package, good to include the package name in the class name

S3 has no checks for correctness which means you can change the class of existing objects!

```
y <- matrix(1:10)
class(y)
```

```
[1] "matrix" "array"
```

```
class(y) <- "my_class"
class(y)
```

```
[1] "my_class"
```

Flexibility can be problematic

```
# Create a linear model
mod <- lm(log(mpg) ~ log(disp), data = mtcars)
class(mod)
```

```
[1] "lm"
```

```
print(mod)
```

```

Call:
lm(formula = log(mpg) ~ log(disp), data = mtcars)

Coefficients:
(Intercept)    log(disp)
     5.3810      -0.4586
```

```
# Turn it into a date
class(mod) <- "Date"
print(mod)
```

```
Error in as.POSIXlt.Date(x): 'list' object cannot be coerced to type 'double'
```

Recommendations when creating a class:

Provide three functions:

1. A constructor, `new_myclass()` <- MOST IMPORTANT!
2. A validator, `validate_myclass()`
3. A user-friendly helper, `myclass()`

Constructor

The constructor should follow three principles:

- Be called new_myclass()
- Have one argument for the base object, and one for each attribute
- Check the type of the base object and the types of each attribute

```
new_factor <- function(x = integer(), levels = character()) {
  stopifnot(is.integer(x))
  stopifnot(is.character(levels))

  structure(
    x,
    levels = levels,
    class = "factor"
  )
}

new_factor(x = 1:2, levels = c("a", "b", "c"))
```

```
[1] a b
Levels: a b c
```

The constructor is a developer function: it's OK to trade a little safety in return for performance.

Validators (1/2)

```
validate_factor <- function(x) {
  values <- unclass(x)
  levels <- attr(x, "levels")

  if (!all(!is.na(values) & values > 0)) {
    stop(
      "All `x` values must be non-missing and greater than zero",
      call. = FALSE
    )
  }

  if (length(levels) < max(values)) {
    stop(
      "There must be at least as many `levels` as possible values in `x`",
      call. = FALSE
    )
  }

  x
}
```

Validators (2/2)

```
# Without validator
new_factor(1:5, "a")
```

```
Error in as.character.factor(x): malformed factor
```

```
#With validator
validate_factor(new_factor(1:5, "a"))
```

```
Error: There must be at least as many `levels` as possible values in `x`
```

Helpers

- Created mainly for users
- A helper should always:
  - Have the same name as the class, e.g. myclass()
  - Finish by calling the constructor, and the validator, if it exists
  - Create carefully crafted error messages tailored towards an end-user
  - Have a thoughtfully crafted user interface with carefully chosen default values and useful conversions
  - Coerce inputs to the desired type

```
factor <- function(x = character(), levels = unique(x)) {
  ind <- match(x, levels)
  validate_factor(new_factor(ind, levels))
}

factor(x = c("a", "a", "b"))
```

```
[1] a a b
Levels: a b
```

Generics and methods

A **generic function** is an interface that uses **method dispatch** to find the right implementation (**method**) depending on the class of the main argument.

Every generic calls `UseMethod()` to do *method dispatch*.

```
# In this case, the generic function is 'print'
print
```

```
function (x, ...)
UseMethod("print")
<bytecode: 0x0000000014e98910>
<environment: namespace:base>
```

Methods are named: `generic.class()`

```
s3_dispatch(print(time))
```

```
=> print.function
 * print.default
```

Creating your own generic is similarly simple:

```
my_new_generic <- function(x) {
    UseMethod("my_new_generic")
}
```

WARNING! It is not always the case! (think about `t.test`)

Method dispatch

How does `UseMethod()` work?

```
x <- Sys.Date()
s3_dispatch(print(x))
```

```
=> print.Date
 * print.default
```

```
x <- matrix(1:10, nrow = 2)
class(x)
```

```
[1] "matrix" "array"
```

```
s3_dispatch(print(x))
```

```
   print.matrix
   print.integer
   print.numeric
=> print.default
```

Find Methods...

...for a generic ...for a class

```
s3_methods_generic("mean")
```

```
# A tibble: 6 x 4
  generic class    visible source
  <chr>   <chr>    <lgl>   <chr>
1 mean    Date     TRUE    base
2 mean    default  TRUE    base
3 mean    difftime TRUE    base
4 mean    POSIXct  TRUE    base
5 mean    POSIXlt  TRUE    base
6 mean    quosure  FALSE   registered S3method
```

Other Object Styles (1/3)

- Record style objects: list of equal-length vectors

```
x <- as.POSIXlt(ISOdatetime(2020, 1, 1, 0, 0, 1:3))
x
```

```
[1] "2020-01-01 00:00:01 CET" "2020-01-01 00:00:02 CET"
[3] "2020-01-01 00:00:03 CET"
```

```
length(x)
```

```
[1] 3
```

```
length(unclass(x))
```

```
[1] 11
```

```
head(unclass(x), 2)
```

```
$sec
[1] 1 2 3

$min
[1] 0 0 0
```

Other Object Styles (2/3)

- Data frames: lists of equal length vectors that are conceptually 2-D. Individual components are exposed to the user

```
x <- data.frame(x = 1:100, y = 1:100)
nrow(x)
```

```
[1] 100
```

```
length(x)
```

```
[1] 2
```

```
unclass(x)
```

```
$x
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100

$y
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
```

Other Object Styles (3/3)

- Scalar objects: use a list to represent a single thing

```
mod <- lm(mpg ~ wt, data = mtcars)
mod
```

```
Call:
lm(formula = mpg ~ wt, data = mtcars)

Coefficients:
(Intercept)           wt
     37.285       -5.344
```

```
length(mod)
```

```
[1] 12
```

```
#unclass(mod)
```

Inheritance

If a method is not found for the class in the first element of the vector, R looks for a method for the second class (and so on):

```
s3_dispatch(print(ordered("x")))
```

```
   print.ordered
=> print.factor
 * print.default
```

```
s3_dispatch(print(Sys.time()))
```

```
=> print.POSIXct
   print.POSIXt
 * print.default
```

A method can delegate work by calling NextMethod().

```
s3_dispatch(ordered("x")[1])
```

```
   [.ordered
=> [.factor
   [.default
-> [ (internal)
```

Other ways of dispatching: Internal generics

Functions that don't call `UseMethod()` but instead call the C functions `DispatchGroup()` or `DispatchOrEval()`. Examples: `[`, `sum()` and `cbind()`

```
s3_dispatch(Sys.time()[1])
```

```
=> [.POSIXct
   [.POSIXt
   [.default
-> [ (internal)
```

Other ways of dispatching: Group generics

Like internal generics, they only exist in base R, and you cannot define your own group generic. Four group generics:

- Math: abs(), sign(), sqrt(), floor(), cos(), sin(), log(), and more
- Ops: +, -, *, /, ^, %%, %/%, &, |, !, ==, !=, <, <=, >=, and >
- Summary: all(), any(), sum(), prod(), min(), max(), and range()
- Complex: Arg(), Conj(), Im(), Mod(), Re()

Methods for group generics are looked for only if the methods for the specific generic do not exist:

```
s3_dispatch(sum(Sys.time()))
```

```
   sum.POSIXct
   sum.POSIXt
   sum.default
=> Summary.POSIXct
   Summary.POSIXt
   Summary.default
-> sum (internal)
```

Double dispatch

Generics in the Ops group, which includes the two-argument arithmetic and Boolean operators like - and &, implement a special type of method dispatch. They dispatch on the type of both of the arguments.

```
date <- as.Date("2017-01-01")
integer <- 1L

date + integer
```

```
[1] "2017-01-02"
```

Three possible outcomes of this lookup:

1. The methods are the same, so it doesn't matter which method is used.
2. The methods are different, and R falls back to the internal method with a warning.
3. One method is internal, in which case R calls the other method.

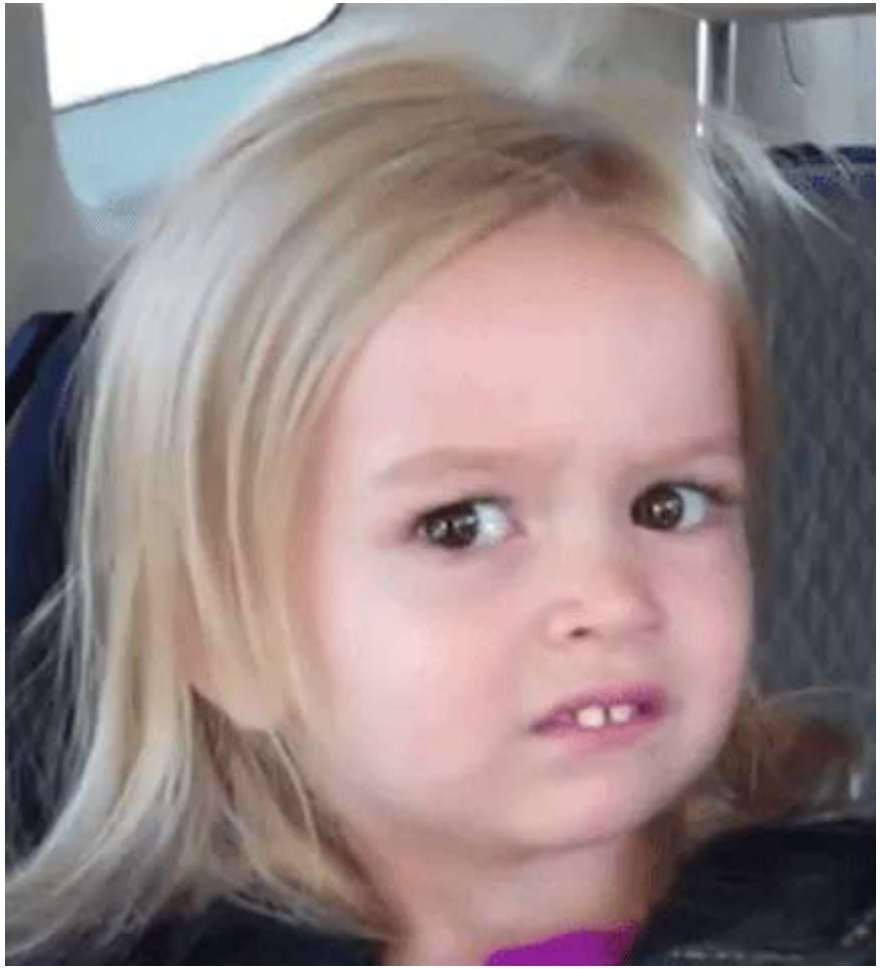
```
s3_dispatch(date + integer)
```

```
=> +.Date
   +.default
 * Ops.Date
   Ops.default
 * + (internal)
```

```
s3_dispatch(integer + date)
```

```
   +.integer
   +.numeric
   +.default
   Ops.integer
   Ops.numeric
   Ops.default
=> + (internal)
```

You got it all, right?



Exercises

Exercise 1

Question Answer

Describe the difference between `t.test()` and `t.data.frame()`? When is each function called?

Exercise 2

Question Answer

What class of object does the following code return? What base type is it built on? What attributes does it use?

```
x <- table(rpois(100, 5))
x
```

```
 1  2  3  4  5  6  7  8  9
 2  9 13 17 16 13 26  2  2
```

Exercise 3

Question Answer-Constructor Answer-Validator Answer-Helper

Read the documentation for `utils::as.roman()`. How would you write a constructor for this class? Does it need a validator? What would a helper look like?

Exercise 4

Question Answer

Carefully read the documentation for `UseMethod()` and explain why the following code returns the results that it does. What two usual rules of function evaluation does `UseMethod()` violate?

```
g <- function(x) {
  x <- 10
  y <- 10
  UseMethod("g")
}
g.default <- function(x) c(x = x, y = y)

x <- 1
y <- 1
g(x)
```

```
x  y
1 10
```

Exercise 5

Question Answer

What do you expect this code to return? What does it actually return? Why?

```
generic2 <- function(x) UseMethod("generic2")
generic2.a1 <- function(x) "a1"
generic2.a2 <- function(x) "a2"
generic2.b <- function(x) {
  class(x) <- "a1"
  NextMethod()
}

generic2(structure(list(), class = c("b", "a2")))
```

```
[1] "a2"
```


PALACE

Thank you!

Do you want to present next?

Or just follow the book club until the end!!