

Programación OO

July 2019

Alejandro Reyes

@areyesq89

alejandro.reyes@gmail.com

alejandroleyes.org

Traducida de *OO programming* de Charlotte Wickham
Adapted from *Tidy Tools* by Hadley Wickham



¿Qué es S3?

¿Qué hace S3?

S3 permite un comportamiento dependiente del contexto

```
x <- 1:5  
y <- factor(letters[1:5])
```

```
summary(x)
```

#	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
#	1	2	3	3	4	5

```
summary(y)
```

#	a	b	c	d	e
#	1	1	1	1	1

Un resumen de 6 números

Una tabla de categorías

summary() es un genérico S3

```
sloop::ftype(summary)
```

```
# [1] "S3"           "generic"
```

```
# summary() buscará methods dependiendo de la  
# clase del objeto
```

```
sloop::s3_class(y)
```

```
# [1] “factor”
```

```
sloop::s3_dispatch(summary(y))
```

```
# => summary.factor      => este método es usado
```

```
# * summary.default      * este método existe pero no es usado
```

Su turno

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

¿Cuál es la **clase** de mod?

¿Qué método es enviado por la llamada a
summary()?

¿Pueden encontrar el código del método?

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

```
s3_class(mod)
# [1] "lm"
```

```
s3_dispatch(summary(mod))
# => summary.lm
# * summary.default
```

```
summary.lm
```

```
# no siempre funciona
# usen `s3_get_method()` para encontrar los métodos no
standard
s3_get_method(summary.lm)
```

Motivación

¿Porqué nos debe de importar S3?



¡Ya están utilizando S3 en sus
análisis!

Objetos S3 importantes en R base

`data.frame()`

`factor()`

`Sys.Date()`

`Sys.time()`

`table()`



Las funciones complejas necesitan
regresar múltiples salidas

Esto es obviamente importante en modelos lineales

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

```
# Pero también su "summary"  
sum <- summary(mod)  
str(sum)
```

A close-up, low-angle shot of the iconic undulating facade of the Walt Disney Concert Hall. The facade is composed of numerous thin, light-colored metal panels that curve and overlap, creating a dynamic, wave-like pattern. The lighting highlights the metallic texture and the shadows between the panels.

La forma sigue
a la función

Un ejemplo son los modelos lineales

```
sum
#> Call:
#> lm(formula = mpg ~ wt, data = mtcars)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -4.5432 -2.3647 -0.1252  1.4096  6.8727
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 37.2851    1.8776 19.858 < 2e-16 ***
#> wt          -5.3445    0.5591 -9.559 1.29e-10 ***
#> ---
#> Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
#>
#> Residual standard error: 3.046 on 30 degrees of freedom
#> Multiple R-squared:  0.7528, Adjusted R-squared:  0.7446
#> F-statistic: 91.38 on 1 and 30 DF,  p-value: 1.294e-10
```

Otro ejemplo son los objetos “tibble”

Tamaño total

```
# A tibble: 53,940 x 10
```

	carat	cut	color	clarity	depth	table	price	x	y	z
1	<dbl>	<ord>	<ord>	<ord>	<dbl>	<dbl>	<int>	<dbl>	<dbl>	<dbl>
2	0.230	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
3	0.210	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
4	0.230	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
5	0.290	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
6	0.310	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75
7	0.240	"Very Good"	J	VVS2	62.8	57.0	336	3.94	3.96	2.48
8	0.240	"Very Good"	I	VVS1	62.3	57.0	336	3.95	3.98	2.47
9	0.260	"Very Good"	H	SI1	61.9	55.0	337	4.07	4.11	2.53
10	0.220	Fair	E	VS2	65.1	61.0	337	3.87	3.78	2.49
	0.230	"Very Good"	H	VS1	59.4	61.0	338	4.00	4.05	2.39
# ... with 53,930 more rows										

Tipo de variable

Solo muestra las primeras 10 filas

S3 hace los paquetes extendibles

Métodos nuevos

te permiten expandir otros paquetes

Genéricos nuevos

te permiten escribir paquetes de
manera que otros lo puedan
expandir fácilmente

Clases “Scalar”

objetos simple pero complejos

Principio:

Definir métodos “structure” y
“print” consistentes para
funciones con salidas
complejas

Cambia al proyecto:
[safely]

Reto: ¿Cómo podemos mejorar las salidas?

```
library(purrr)
safe_log <- safely(log)

safe_log("a")
#> $result
#> NULL
#>
#> $error
#> <simpleError in log(...):
#>   non-numeric argument to
#>   mathematical function>

safe_log(10)
#> $result
#> [1] 2.302585
#>
#> $error
#> NULL
```

Creando una nueva clase S3

1. Definir el nombre *safely*
2. Definir las propiedades
3. Codificar el constructor
4. Codificar los métodos

Su turno

¿Qué cosas son inmutables de la función safely?

```
safe_log <- purrr::safely(log)
```

```
safe_log(x)
```

```
# ¿Qué sabemos del resultado de safe_log(x)  
que
```

```
# siempre es verdad?
```

Inmutables

Regresa una lista en la que

- hay dos componentes: result y error
- results siempre viene primero que error
- uno de los componentes siempre es NULL

Ahora escribimos el constructor

```
new_safely <- function(result = NULL, error = NULL) {  
  if (!is.null(result) && !is.null(error)) {  
    stop(  
      "One of `result` and `error` must be NULL",  
      call. = FALSE  
    )  
  }  
}
```

Checa la
entrada

```
structure(  
  list(  
    result = result,  
    error = error  
  class = "safely"  
)  
}
```

Implementa la
clase

Definición de la función safely

```
safely <- function(.f) {  
  stopifnot(is.function(.f))  
  
  function(...) {  
    tryCatch({  
      list(result = .f(...), error = NULL)  
    }, error = function(e) {  
      list(result = NULL, error = e)  
    })  
  }  
}
```

Ahora utilizamos el constructor

```
safely <- function(.f) {  
  stopifnot(is.function(.f))  
  
  function(...) {  
    tryCatch({  
      new_safely(result = .f(...))  
    }, error = function(e) {  
      new_safely(error = e)  
    })  
  }  
}
```

Abreviación

Prueba

`expect_null()`

Revisa si es NULL

`expect_type()`
`expect_s3_class()`
`expect_s4_class()`

Revisa que el output sea la clase esperada, ya sean estructuras base, clases S3, clases S4, etc

`expect_true()`
`expect_false()`

Revisa todas las expectativas que no son cubiertas por otras funciones

Su turno

Escriban pruebas unitarias para asegurarse de que la función `new_safely()` regrese la salida esperada independientemente si un error ocurre o no. (i.e. expresen los inmutables de la clase como pruebas unitarias)

```
# In tests/testthat/test-safely.R
context("test-safely.R")

test_that("can only supply error or result", {
  expect_error(new_safely(1, 2), "must be NULL")
})

test_that("it's ok for both to be null", {
  expect_error(new_safely(NULL, NULL), NA)
})

test_that("result and error are captured", {
  s1 <- new_safely(result = 1)
  s2 <- new_safely(error = 1)

  expect_s3_class(s1, "safely")
  expect_equal(s1$result, 1)
  expect_equal(s1$error, NULL)

  expect_s3_class(s2, "safely")
  expect_equal(s2$result, NULL)
  expect_equal(s2$error, 1)
})
```

Expect no error

Ahora mejoraremos el método print

```
safe_log(10)
#> <safely: ok>
#> [1] 2.302585
```

Es buena práctica incluir
el tipo de output en <>

```
safe_log("a")
#> <safely: error>
#> Error: non-numeric argument to
#> mathematical function
```

Todos los métodos S3 tienen la misma estructura básica

generic

Same arguments as generic

```
print.safely <- function(x, ...) {
```

class

```
}
```

Los métodos
pertenecen a
genéricos, no a la
clase

`print`

`mean`

`sum`

Date

POSIXct integer

Date

POSIXct integer

print

mean

sum

Su turno: completen los espacios vacíos

```
# En R/safely.R
print.safely <- function(x, ...) {
}

# Un “helper” útil de utils.R
cat_line <- function(...) {
  cat(..., "\n", sep = "")
}
# Véase https://github.com/r-lib/cli si
# buscan más “helpers”.
```

Algunas pruebas

```
f <- function() stop("message")
```

```
g <- function() 1
```

```
safe_f <- safely(f)
```

```
safe_g <- safely(g)
```

```
safe_f()
```

```
safe_g()
```

El método print

```
print.safely <- function(x, ...) {  
  if (!is.null(x$error)) {  
    cat_line("<safely: error>")  
    cat_line("Error: ", x$error$message)  
  } else {  
    cat_line("<safely: ok>")  
    print(x$result)  
  }  
}
```

```
invisible(x)
```

```
}
```

Invisible sirve para
evitar efectos
secundarios

Un poco de color puede hacerlo más legible

```
print.safely <- function(x, ...) {  
  if (!is.null(x$error)) {  
    cat_line("<safely: ", crayon::bold(crayon::red("error")), ">")  
    cat_line(crayon::red("Error: "), x$error$message)  
  } else {  
    cat_line("<safely: ", crayon::green("ok"), ">")  
    print(x$result)  
  }  
  
  invisible(x)  
}
```

Genéricos nuevos

Cambia al proyecto:
[bizarro]

Objetivo: create una función bizarro

```
bizarro("abc")
```

```
#> [1] "cba"
```

```
bizarro(1)
```

```
#> [1] -1
```

```
bizarro(c(TRUE, FALSE))
```

```
#> [1] FALSE TRUE
```

Podríamos usar “if” y “else”

```
str_reverse <- function(x) {  
  purrr::map_chr(stringr::str_split(x, ""),  
    ~ stringr::str_flatten(rev(.x)))  
}  
  
bizarro <- function(x) {  
  if (is.character(x)) {  
    str_reverse(x)  
  } else if (is.numeric(x)) {  
    -x  
  } else if (is.logical(x)) {  
    !x  
  } else {  
    stop(  
      "Don't know how to make bizzaro <", class(x)[[1]], ">",  
      call. = FALSE)  
  }  
}
```

En cambio, definiremos un nuevo genérico

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

Mágicamente pasa los argumentos correctos a la función

generic.class

```
bizarro.character <- function(x) {  
  str_reverse(x)  
}
```

```
bizarro("abc")  
#> [1] cba
```

Permite expandir a cualquiera

Su turno

Implementen:

1. Un método para objetos numéricos que multiplique por -1
2. Un método para objetos lógicos que invierta TRUE/FALSE
3. Un método para objetos data frame que haga la función bizarro a los nombres de las columnas y a las columnas también.

(i.e. get tests passing)

```
bizarro.numeric <- function(x) {  
  -x  
}
```

```
bizarro.logical <- function(x) {  
  !x  
}
```

```
bizarro.data.frame <- function(x) {  
  names(x) <- bizarro(names(x))  
  x[] <- purrr::map(x, bizarro)  
  x  
}
```

Técnica útil: Si el método es para objetos complejos, usen la función en cada componente

¿Qué pasa cuando un método no existe?

```
bizarro(factor(letters))
#> Error in UseMethod("bizarro") :
#>   no applicable method for 'bizarro'
#>   applied to an object of class "factor"

# ¿Podríamos mejorar?
# Podemos implementar un método default
```

```
bizarro.default <- function(x) {  
  stop(  
    "Don't know how to make bizzaro < ",  
    class(x)[[1]], ">",  
    call. = FALSE  
)  
}
```

```
bizarro(factor(letters))  
#> Error: Don't know how to make  
#> bizzaro <factor>
```

Su turno

¿Cuál debería ser la salida de bizzaro.factor("abc"))?

Decidan, implementen el método bizarro.factor() y escriban pruebas unitarias.

Una idea: aplicar bizarro en los “levels”

```
# En tests/testthat/test-bizarro.R
test_that("bizarro factors have levels reversed", {
  f1 <- factor(c("abc", "def", "abc"))
  f2 <- factor(c("cba", "fed", "cba"))

  expect_equal(bizarro(f1), f2)
  expect_equal(bizarro(f2), f1)
})
```

```
# En R/bizarro.R  
bizarro.factor <- function(x) {  
  levels(x) <- bizarro(levels(x))  
  x  
}
```

Más material

R avanzado (edición II) tiene cuatro capítulos

S3: <https://adv-r.hadley.nz/s3.html>

S4: <https://adv-r.hadley.nz/s4.html>

R6: <https://adv-r.hadley.nz/r6.html>

Trade-offs: <https://adv-r.hadley.nz/oo-tradeoffs.html>

Adapted from *Tidy Tools* by Hadley Wickham

This work is licensed as
Creative Commons
Attribution-ShareAlike 4.0
International

To view a copy of this license, visit
[https://creativecommons.org/
licenses/by-sa/4.0/](https://creativecommons.org/licenses/by-sa/4.0/)