OO programming

January 2017

Hadley Wickham
@hadleywickham
Chief Scientist, **RStudio**

Warmup

Your turn

What is an attribute? What types of objects can have attributes?

How do you *get* the value of an attribute?
How do you *set* the value an attribute?

What are the three most important attributes?

Attributes add arbitrary metadata to any object

```r
x <- 1:6
attr(x, "max") <- 5
attr(x, "max")
attributes(x)

# structure returns a modified object with attrs
structure(1:10, max = 5)

# Most important attributes are dim, class and
# names.  Should always use dim(), class() and
# names() respectively to get and set their
# values.
```

Your turn

Every S3 class is built on a base type (e.g. a vector). The two most important S3 classes are factor and data frame.

What are factors built on top of?
What attributes do they use?

What are data frames built on top of?
What attributes do they use?

```r
f <- factor(c("a", "b", "c"))
typeof(f)     # Built on top of integer
attributes(f) # Use levels and class attributes

d <- data.frame(f)
typeof(d)     # Built on top of list
attributes(d) # names, row.names and class
```

Your turn

What is a generic function?

What makes a generic function generic?
How is it different from a regular function?

You could use a nested if statement

```r
mean <- function(x, ...) {
  if (is.Date(x)) {
    ...
  } else if (is.difftime(x)) {
    ...
  } else if (is.POSIXct(x)) {

  } else if (is.POSIXlt(x)) {
    ...
  } else {
    ...
  }
}
```

But a generic function lets anyone extend

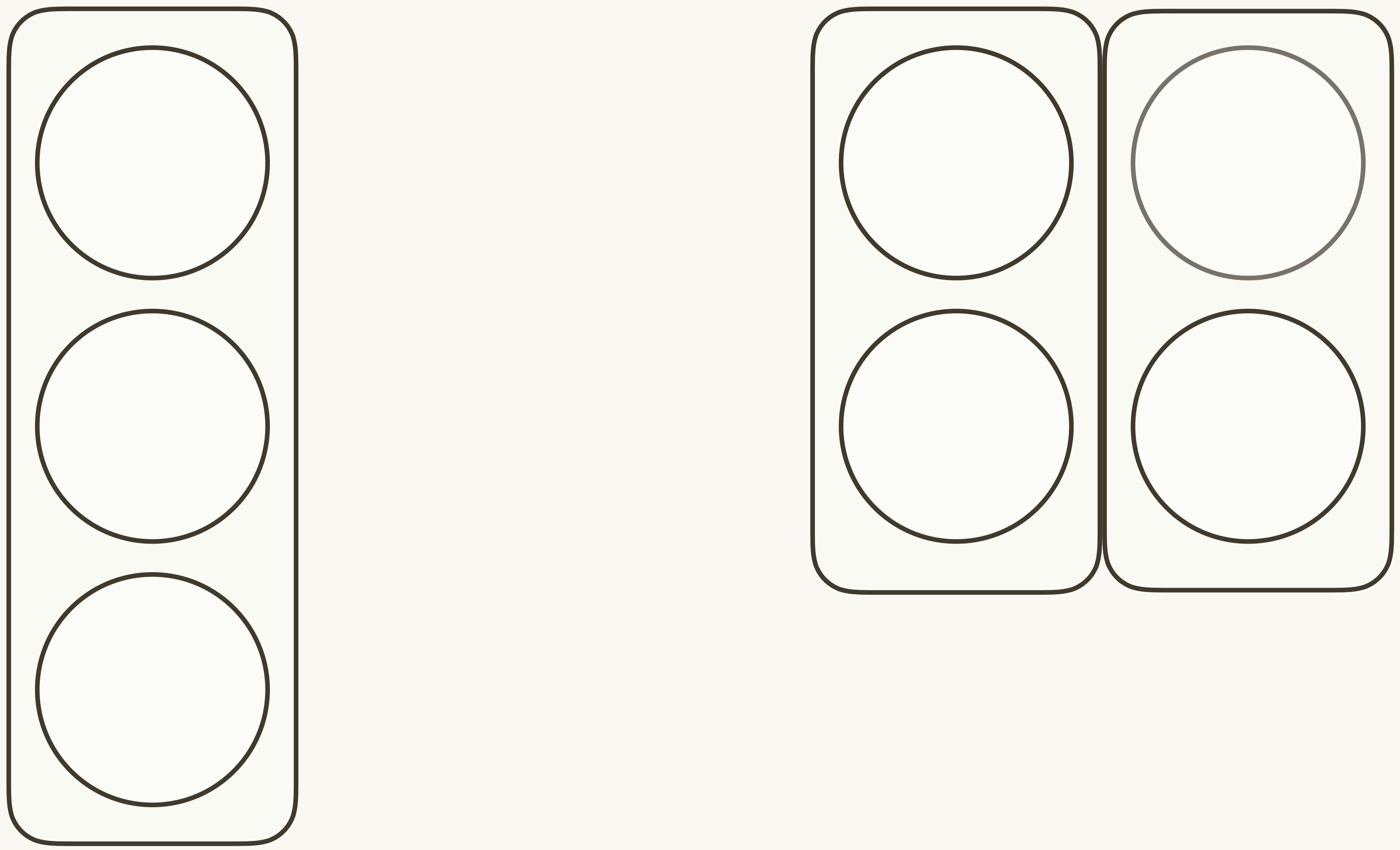
```
mean <- function(x, ...) {
  UseMethod("mean")
}

mean.Date <- function(x, ...) ...
mean.difftime <- function(x, ...) ...
mean.POSIXct <- function(x, ...) ...
mean.POSIXlt <- function(x, ...) ...

mean.default <- function(x, ...) ...
```

```
mean <- function(x, ...) {
  fun <- paste(”mean”, class(x), sep = ”.”)
  match.fun(fun)(x, ...)
}

mean.Date <- function(x, ...) ...
mean.difftime <- function(x, ...) ...
mean.POSIXct <- function(x, ...) ...
mean.POSIXlt <- function(x, ...) ...

mean.default <- function(x, ...) ...
```

Motivation

Why should you care about S3?

Complex functions need to return multiple things



S3 lets you control how your objects are printed

```
    carat       cut color clarity depth table price    x    y    z
1    0.23     Ideal     E     SI2  61.5  55.0   326 3.95 3.98 2.43
2    0.21   Premium     E     SI1  59.8  61.0   326 3.89 3.84 2.31
3    0.23      Good     E     VS1  56.9  65.0   327 4.05 4.07 2.31
4    0.29   Premium     I     VS2  62.4  58.0   334 4.20 4.23 2.63
5    0.31      Good     J     SI2  63.3  58.0   335 4.34 4.35 2.75
6    0.24 Very Good     J    VVS2  62.8  57.0   336 3.94 3.96 2.48
7    0.24 Very Good     I    VVS1  62.3  57.0   336 3.95 3.98 2.47
8    0.26 Very Good     H     SI1  61.9  55.0   337 4.07 4.11 2.53
9    0.22      Fair     E     VS2  65.1  61.0   337 3.87 3.78 2.49
10   0.23 Very Good     H     VS1  59.4  61.0   338 4.00 4.05 2.39
11   0.30      Good     J     SI1  64.0  55.0   339 4.25 4.28 2.73
12   0.23     Ideal     J     VS1  62.8  56.0   340 3.93 3.90 2.46
13   0.22   Premium     F     SI1  60.4  61.0   342 3.88 3.84 2.33
14   0.31     Ideal     J     SI2  62.2  54.0   344 4.35 4.37 2.71
15   0.20   Premium     E     SI2  60.2  62.0   345 3.79 3.75 2.27
16   0.32   Premium     E      I1  60.9  58.0   345 4.38 4.42 2.68
17   0.30     Ideal     I     SI2  62.0  54.0   348 4.31 4.34 2.68
18   0.30      Good     J     SI1  63.4  54.0   351 4.23 4.29 2.70
19   0.30      Good     J     SI1  63.8  56.0   351 4.23 4.26 2.71
20   0.30 Very Good     J     SI1  62.7  59.0   351 4.21 4.27 2.66
21   0.30      Good     I     SI2  63.3  56.0   351 4.26 4.30 2.71
22   0.23 Very Good     E     VS2  63.8  55.0   352 3.85 3.92 2.48
23   0.23 Very Good     H     VS1  61.0  57.0   353 3.94 3.96 2.41
24   0.31 Very Good     J     SI1  59.4  62.0   353 4.39 4.43 2.62
25   0.31 Very Good     J     SI1  58.1  62.0   353 4.44 4.47 2.59
26   0.23 Very Good     G    VVS2  60.4  58.0   354 3.97 4.01 2.41
27   0.24   Premium     I     VS1  62.5  57.0   355 3.97 3.94 2.47
28   0.30 Very Good     J     VS2  62.2  57.0   357 4.28 4.30 2.67
29   0.23 Very Good     D     VS2  60.5  61.0   357 3.96 3.97 2.40
30   0.23 Very Good     F     VS1  60.9  57.0   357 3.96 3.99 2.42
```

Plus 970 more rows

A good print method is very valuable

Total size

```
Source: local data frame [53,940 x 10]

   carat       cut  color clarity depth table price     x     y     z
   (dbl)    (fctr) (fctr)  (fctr) (dbl) (dbl) (int) (dbl) (dbl) (dbl)
1   0.23     Ideal      E     SI2  61.5    55   326  3.95  3.98  2.43
2   0.21   Premium      E     SI1  59.8    61   326  3.
3   0.23      Good      E     VS1  56.9    65   327  4.0
4   0.29   Premium      I     VS2  62.4    58   334  4.20  4.23  2.63
5   0.31      Good      J     SI2  63.3    58   335  4.34  4.35  2.75
6   0.24 Very Good      J    VVS2  62.8    57   336  3.94  3.96  2.48
7   0.24 Very Good      I    VVS1  62.3    57   336  3.95  3.98  2.47
8   0.26 Very Good      H     SI1  61.9    55   337  4.07  4.11  2.53
9   0.22      Fair      E     VS2  65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good      H     VS1  59.4    61   338  4.00  4.05  2.39
..   ...       ...    ...     ...   ...   ...   ...   ...   ...   ...
```

Variable type

Only shows first 10 rows

Over 100 lines of code!

A constructor ensures that you always return the same thing

Powers dr_github()
and dr_devtools()

```
doctor <- function(name, messages) {
  structure(
    length(messages) == 0,
    doctor = paste0("DR_", toupper(name)),
    messages = messages,
    class = "doctor"
  )
}


print.doctor <- function(x, ...) {
  if (x) {
    message(attr(x, "doctor"), " SAYS YOU LOOK HEALTHY")
    return()
  }

  warning(attr(x, "doctor"), " FOUND PROBLEMS", call. = FALSE, immediate. = TRUE)
  messages <- strwrap(attr(x, "messages"), exdent = 2)
  message(paste(messages, collapse = "\n"))
}
```

S3 makes packages extensible

New methods

Lets you extend other packages

New generics

Write packages in way that others can easily extend.

Complex return values

Your turn

How is a date represented? How is a date time represented? (what do the values *mean*? unclass(x))

How is a linear model represented?

```
x1 <- as.Date("1970-01-02")
x2 <- as.POSIXct(x1)
x3 <- lm(mpg ~ wt, data = mtcars)
```

Dates and date times are built on top of vectors

```
x <- as.Date("1970-01-02")
typeof(x)
attributes(x)
unclass(x)
# Number of days since Jan 1, 1970

y <- as.POSIXct(x)
typeof(y)
attributes(y)
unclass(y)
# Number of seconds since Jan 1, 1970
```

Linear models are built on top of lists

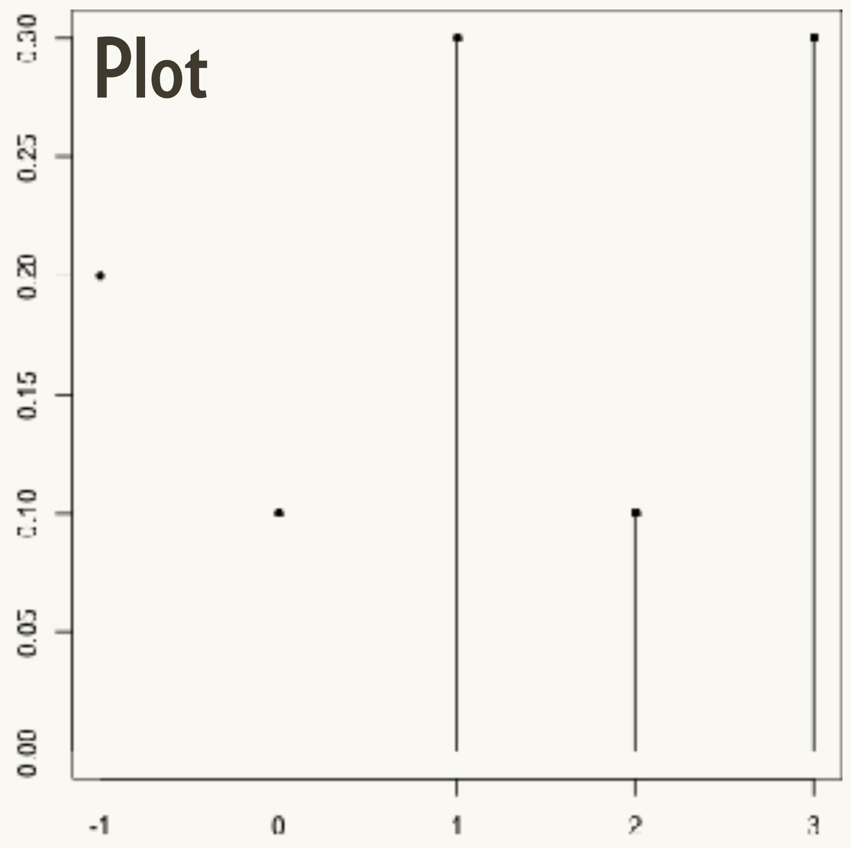
```
mod <- lm(mpg ~ wt, data = mtcars)
typeof(mod)
attributes(mod)

names(mod)
unclass(mod)
```

Goal: model random variables in R

x	-1	0	1	2	3
P(x)	0.2	0.1	0.3	0.1	0.3

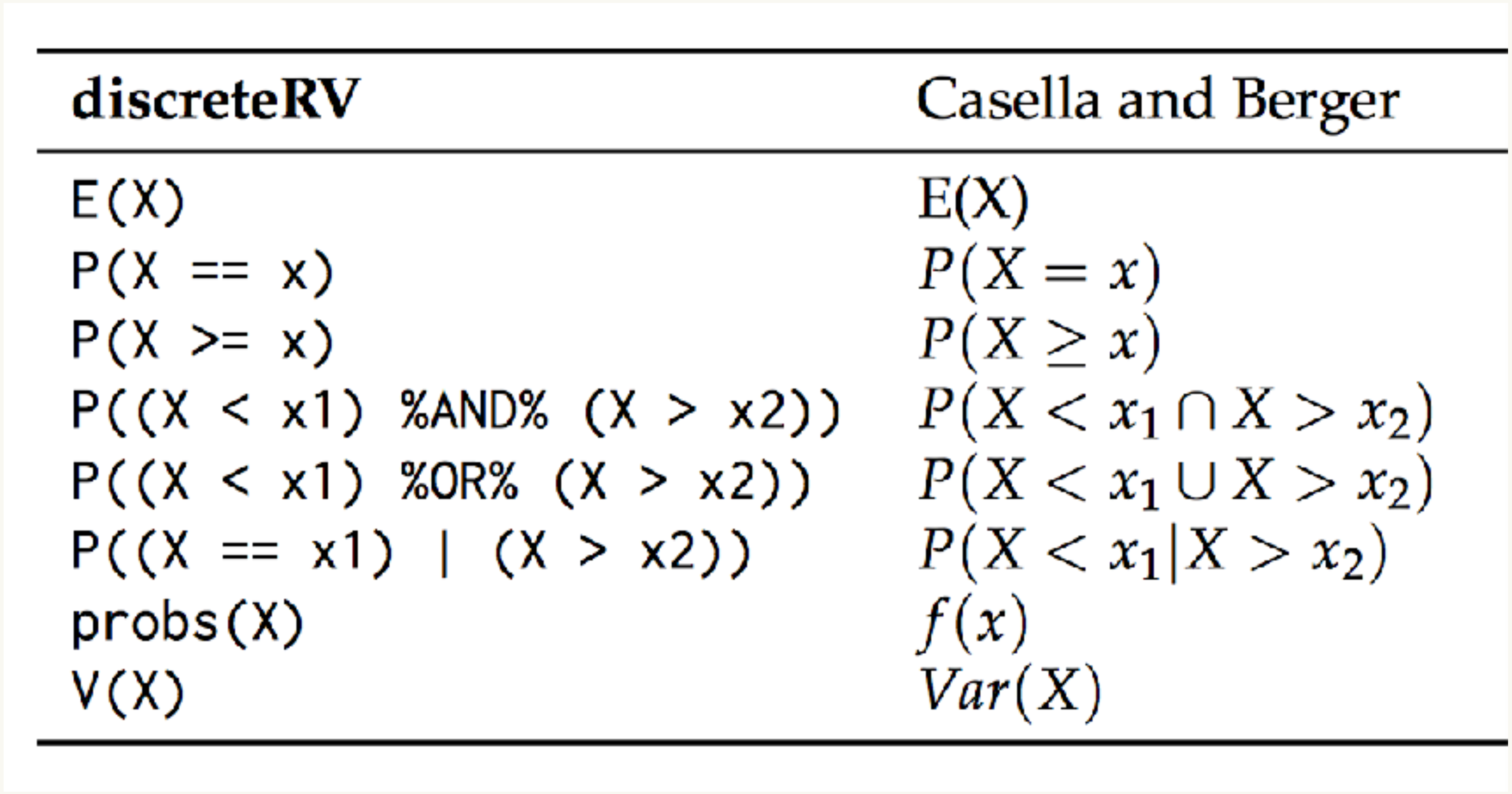
1.9

Mean



```r
source(”rv.r”)

dice <- rv(1:6)
mean(dice)
min(dice)
max(dice)
range(dice)

P(dice > 3)
plot(dice + dice + dice)
```

discreteRV explores these ideas in more depth



discreteRV	Casella and Berger
`E(X)`	E(X)
`P(X == x)`	$P(X = x)$
`P(X >= x)`	$P(X \geq x)$
`P((X < x1) %AND% (X > x2))`	$P(X < x_1 \cap X > x_2)$
`P((X < x1) %OR% (X > x2))`	$P(X < x_1 \cup X > x_2)$
`P((X == x1) | (X > x2))`	$P(X < x_1 \| X > x_2)$
`probs(X)`	$f(x)$
`V(X)`	$Var(X)$

http://journal.r-project.org/archive/2015-1/hare-buja-hofmann.pdf

Representing a random variable

x	-1	0	1	2	3
P(x)	0.2	0.1	0.3	0.1	0.3

```
x <- c(-1, 0, 1, 2, 3)
p <- c(0.2, 0.1, 0.3, 0.1, 0.3)
```

How might we represent this random variable as a single R object?

(There are at least five ways)

x	-1	0	1	2	3
P(x)	0.2	0.1	0.3	0.1	0.3

```r
x <- c(-1, 0, 1, 2, 3)
p <- c(0.2, 0.1, 0.3, 0.1, 0.3)

# Ways to store
structure(x, prob = p)
structure(p, val = x)
list(x = x, p = p)
data.frame(x, p)

# Use whichever makes your life easier
```

S3 is just a convention

Simplest OO system that might possibly work

No formal definition!

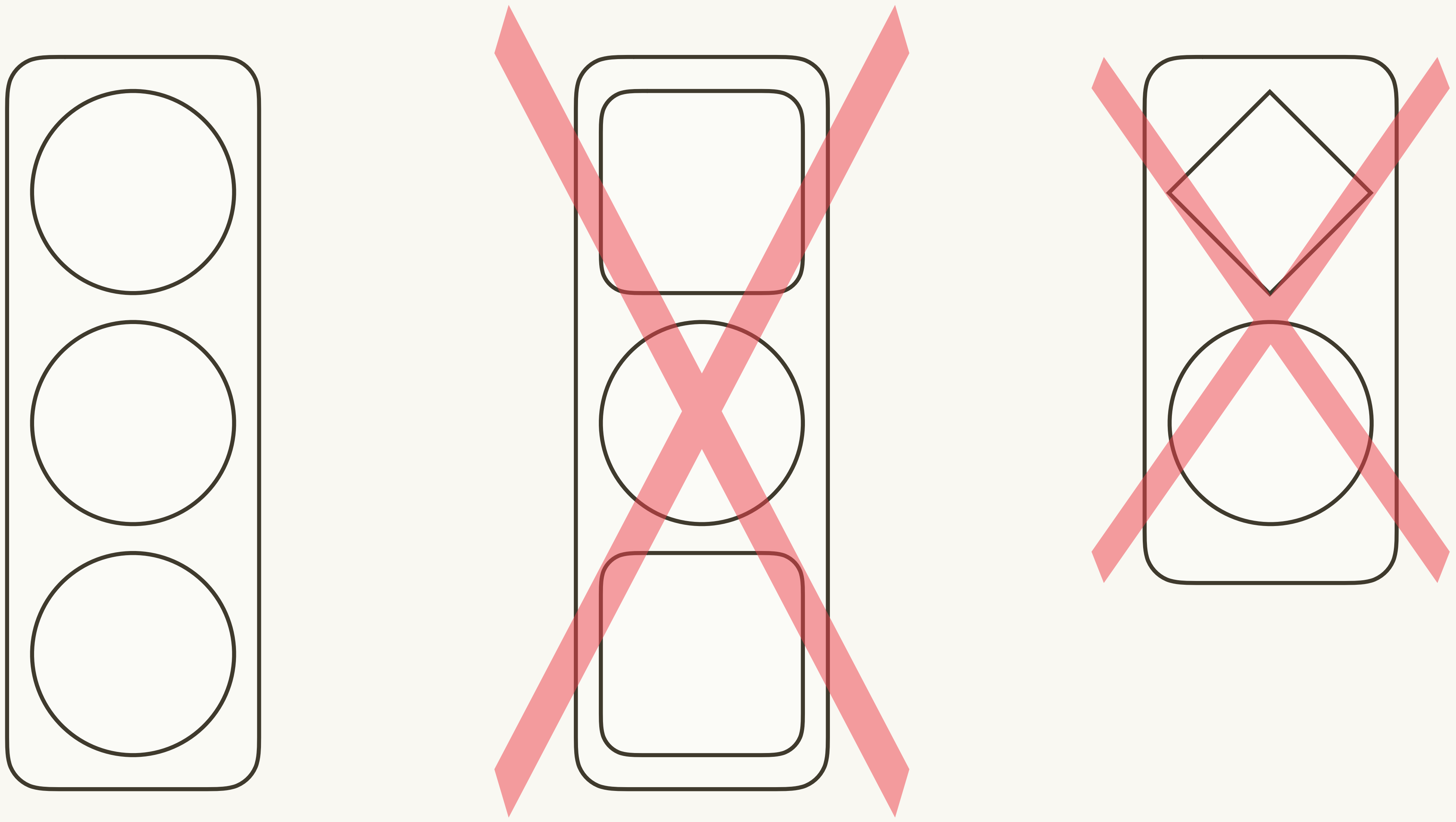
```
structure(x, prob = p, class = "rv")

# Or
rv <- structure(x, prob = p)
class(rv) <- "rv"
```

S3 is easy to abuse!

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

class(mod) <- "data.frame"
mod

# Surprisingly, this doesn't cause many problems!
```

Constructors

A constructor ensures that you’re consistent



Constructor functions can be helpful

```r
rv <- function(x, probs = NULL) {
  if (is.null(probs)) {
    probs <- rep(1, length(x)) / length(x)
  }
  structure(x, probs = probs, class = "rv")
}

# We'll also write a helper to extract
# the probabilities
probs <- function(x) attr(x, "probs")
```

Also useful to add an is.*() function

```r
is.rv <- function(x) {
  # equivalent to "rv" %in% class(x)
  inherits(x, "rv")
}
```

What’s wrong with each of these rvs?

1

x	-1	0	1
P(x)	0.5	0.5	0.5

2

x	-1	0	1
P(x)	“a”	FALSE	😀

3

x	-1	0	1
P(x)	-0.25	1	0.25

4

x	-1	-1	-1
P(x)	0.5	0.4	0.1

5

x	-1	0	1
P(x)	0.33	0.67	

6

x	-1	0	1
P(x)	NA	0.5	0.5

How could you write code to detect the problem?

A constructor should also check its inputs

```
check_probs <- function(x) {
  if (!is.numeric(x)) {
    stop("`prob` must be numeric.")
  }
  if (any(is.na(x))) {
    stop("`prob` must not contain any NA")
  }
  if (any(x < 0)) {
    stop("All `prob` must be >= 0")
  }
  if (sum(x) != 1) {
    stop("`sum(prob)` must equal 1")
  }
}
x <- rep(1/49, 49)
check_probs(x)
```

Beware the perils of floating point!

```
check_probs <- function(x) {
  if (!is.numeric(x)) {
    stop("`prob` must be numeric.")
  }
  if (any(is.na(x))) {
    stop("`prob` must not contain any NA")
  }
  if (any(x < 0)) {
    stop("All `prob` must be >= 0")
  }
  if (abs(sum(x) - 1) > 1e-6) {
    stop("`sum(prob)` must equal 1")
  }
}
x <- rep(1/49, 49)
check_probs(x)
```

Need to balance strictness and helpfulness

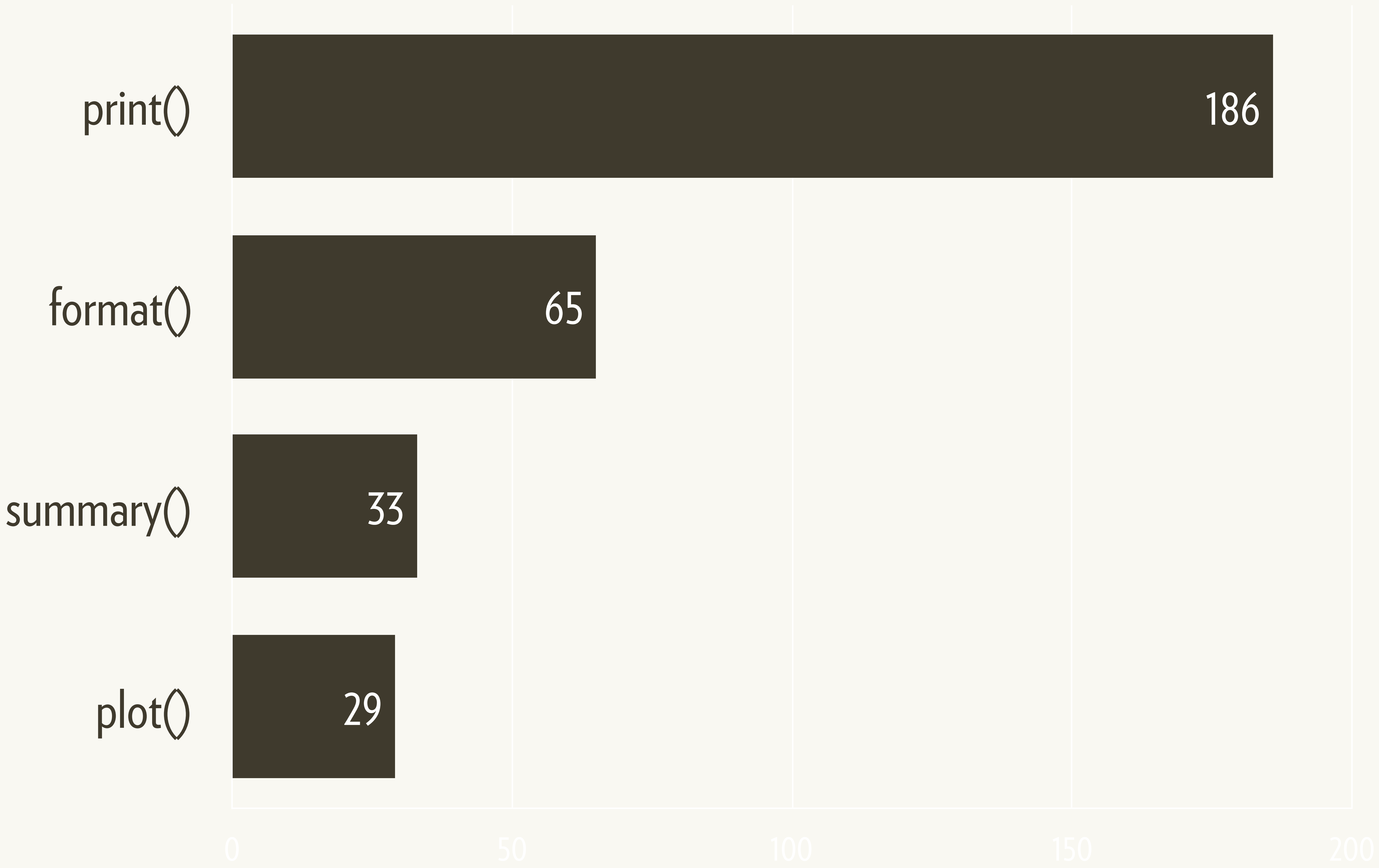
```
rv <- function(x, probs = NULL) {
  if (is.rv(x)) x <- as.numeric(x)
  if (is.null(probs)) {
    probs <- rep(1, length(x)) / length(x)
  } else {
    if (length(x) != length(probs)) stop("Values and probability...")
    check_probs(probs)
 }

  # Simplify by summing probabilities with equal x's. Need to use
  # addNA since otherwise tapply silently drops groups with missing values
  grp <- addNA(x, ifany = TRUE)
  x_new <- as.vector(tapply(x, grp, `[`, 1))
  probs <- as.vector(tapply(probs, grp, sum))

  # Set probs and class attributes
  structure(x_new, probs = probs, class = "rv")
}
```

Strict

Helpful

Methods

Most common methods are to do with output



Use methods() to find methods!

```
# See what methods are defined for print and summary
methods("print")
methods("summary")

# See what methods are defined for data.frame
# and factor
methods(class = "data.frame")
methods(class = "factor")

`[.factor`
print.factor
getS3method("[", "factor")
```

Methods belong to **functions**, not classes

Method calls look different in other languages

```
# S3 and S4
method(object, 1, 2, 3)

# Many other languages
object.method(1, 2, 3)

# RC/R6
object$method(1, 2, 3)
```

	factor	Date	data frame
relevel	✓		
mean		✓	
rep	✓	✓	
print	✓	✓	✓

Message passing

	factor	Date	data frame
relevel	✓		
mean		✓	
rep	✓	✓	
print	✓	✓	✓

Generic function

Methods belong to **functions**, not classes

This is called "generic function" style OO.

It's also what Julia uses.

Print method is most important

```
# To write method, first identify generic and find
# its arguments
print

# Can tell it's a generic function because it uses
# UseMethod()

# Methods follow simple naming scheme
print.rv <- function(x, ...) {
  ...
}
```

Your turn

Fill in the template on the next slide to create a print method for rv objects.

Good print methods are really hard, so aim to get the important data out, even if it doesn't look great.

Fill in this template:

```
print.rv <- function(x, ...) {
  cat("THIS IS MY METHOD\n")
}
```

Prints text directly to console

```
dice <- rv(1:6)
dice
```

`print()` is usually called automatically

This is my method:

```
print.rv <- function(x, ...) {
  X <- format(x, digits = 3)
  P <- format(probs(x), digits = 3)
  out <- cbind(X = X, "P(X)" = P)
  rownames(out) <- rep("", nrow(out))
  print(out, quote = FALSE)
}

dice <- rv(1:6)
dice
```

Another common method is plot

```
plot.rv <- function(x, ...) {
  name <- deparse(substitute(x))
  ylim <- range(0, probs(x))

  plot(as.numeric(x), probs(x), type = "h", ylim = ylim,
    xlab = name, ylab = paste0("P(", name, ")"), ...)
  points(as.numeric(x), probs(x), pch = 20)
  abline(h = 0, col = "gray")
}
```

Your turn

Implement a mean method. The mean summarises the "middle" of the distribution. Mean = E(X) = "Sum" of all outcomes, weighted by their probability.

What name should your function have? What arguments?

x	-1	0	1	2	3
P(x)	0.2	0.1	0.3	0.1	0.3

```
mean.rv <- function(x, ...) {
  sum(x * probs(x))
}
```

Inheritance

Class	class()	Implicit class
Time	`POSIXct, POSIXt`	`numeric, double`
Generalised linear model	`glm, lm`	`list`
Data frame	`data.frame`	`list`
Tibble	`tbl_df, tbl, data.frame`	`list`

http://adv-r.had.co.nz/OO-essentials.html

Extensibility

Write your own generics so others can extend

```
mymethod <- function(x, y, ...) {
  UseMethod("mymethod")
}

# Now anyone can extend your framework by
# implementing methods for their objects.

# Documentation extremely important. Few
# packages ever get popular enough to garner
# significant contributions
```

Learning more

S4

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

`setClass()` defines classes.
`setGeneric()` defines generic functions.
`setMethod()` defines methods.

R6

Package inspired by ReferenceClasses, but much faster & fixes major problem.

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

Do not copy-on-modify. Has mutable state, or reference semantics.

Recommendations

Use **S3** unless:

You have complex network of classes and methods — use **S4**.

You have objects with changing state — use **R6**.

Other resources

John Chambers, "*Extending R*", https://amzn.com/1498775713

https://www.google.com/search?q=bioconductor+s4

http://adv-r.had.co.nz/OO-essentials.html