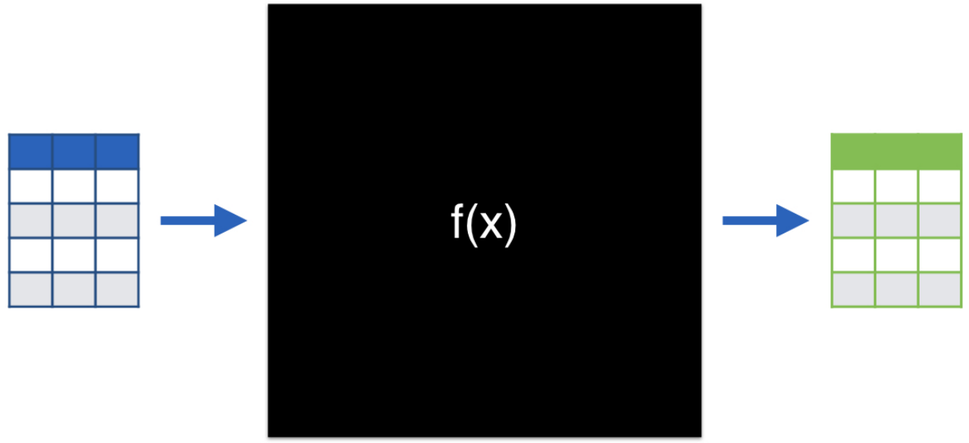
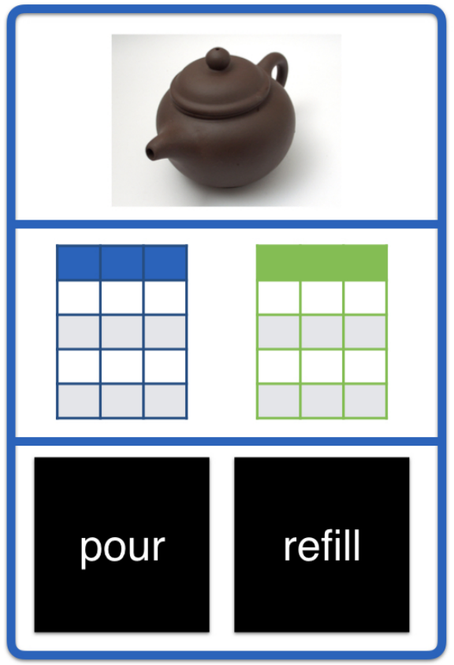
What is Object-Oriented Programming?

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Richie Cotton
Data Evangelist at DataCamp

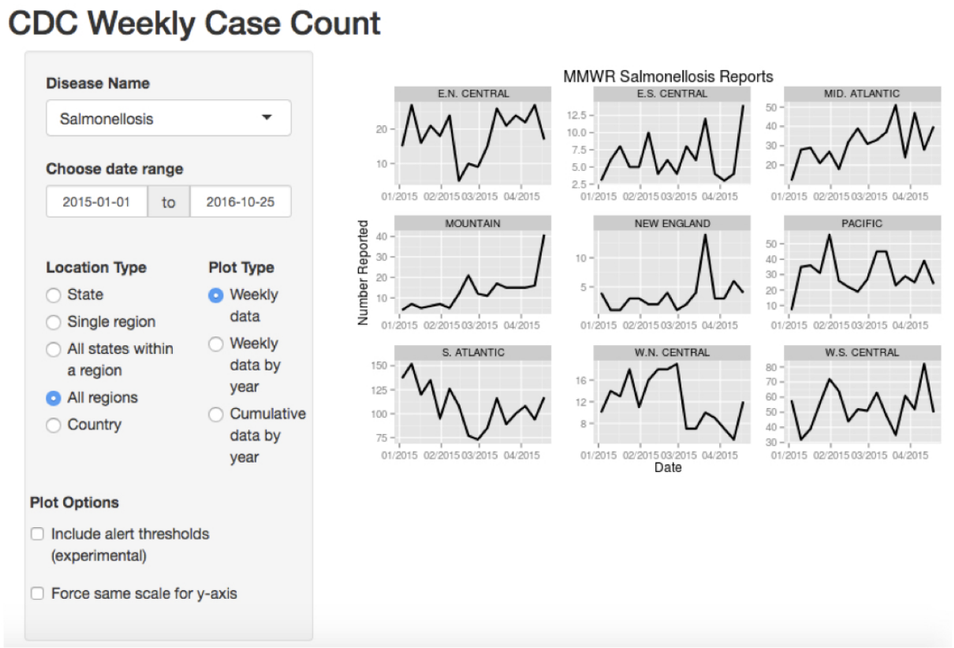



```
calculate_something  <- function(x,  y, z) {
  # do something
  return(the_result)
}
```



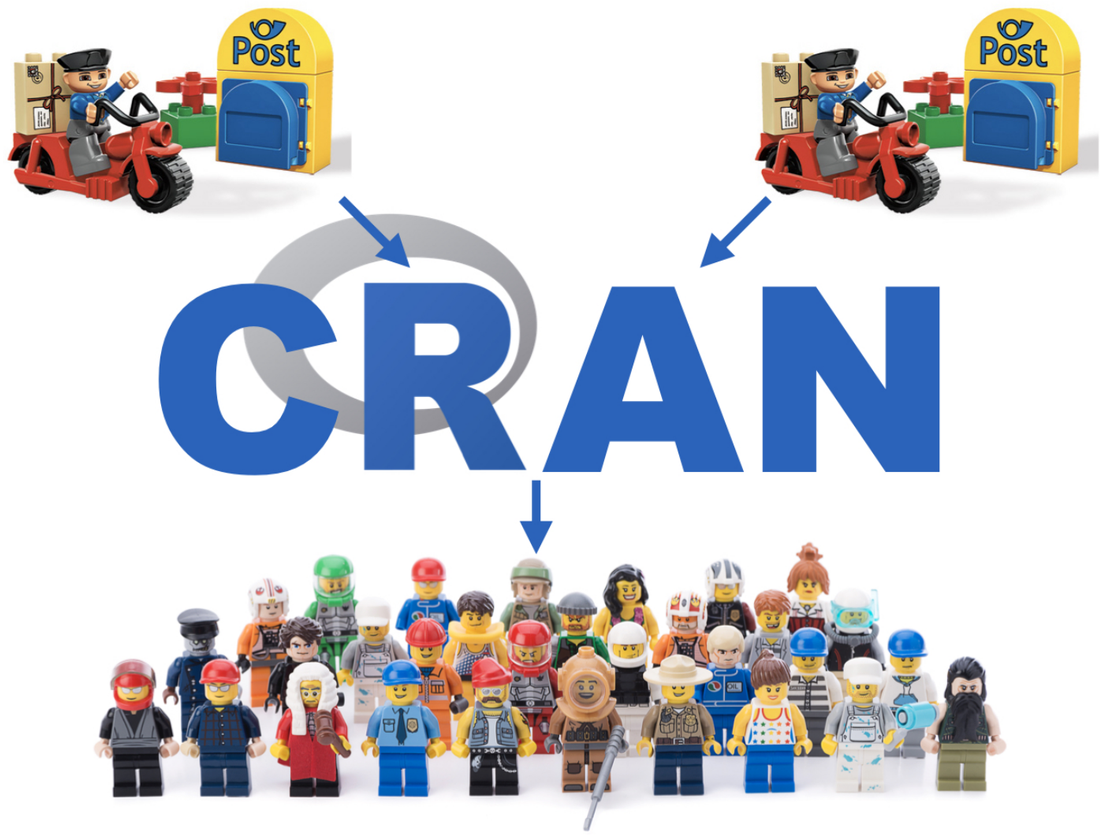
A method is just a function, talked about inan OOP context

Bioconductor
OPEN SOURCE SOFTWARE FOR BIOINFORMATICS



Summary

- With **functional programming**, think about the **functions first.**
- With **object-oriented programming** (OOP) think about the **data structures first.**
- **Don't** use OOP for **general purpose data analyses.**
- **Do** use OOP when you have a **limited number of complex objects.**



Summary

- Use **S3 regularly**
- Use **R6** when you need **more power**
- Use **S4** for **Bioconductor**
- **Maybe** use **ReferenceClasses**

How does R Distinguish Variables?

- `class()` is your **first choice** for determining the kind of variable
- `typeof()` is also **occasionally useful**
- `mode()` and `storage.mode()` are old functions; **don't use them**

```
(x <- rexp(10))
```

```
0.195051 2.191040 0.498703 0.976122 0.299001
0.105187 0.090073 2.328233 3.043201 2.129631
```

```
class(x) <- "random_numbers"
```

```
x
```

```
0.195051 2.191040 0.498703 0.976122 0.299001
0.105187 0.090073 2.328233 3.043201 2.129631
```

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

```
class(x)
```

```
"random_numbers"
```

```
typeof(x)
```

```
"double"
```

Generics and Methods

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Richie Cotton
Data Evangelist at DataCamp

datacamp

```
summary(c(TRUE, FALSE, NA, TRUE))
```

```
   Mode   FALSE    TRUE    NA's
logical       1       2       1
```

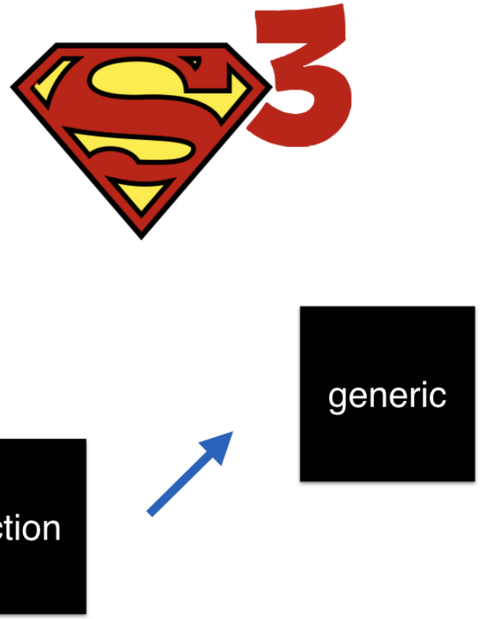
```
summary(rgamma(1000, 1))
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
0.000354 0.276500 0.690300 1.020000 1.384000 9.664000
```

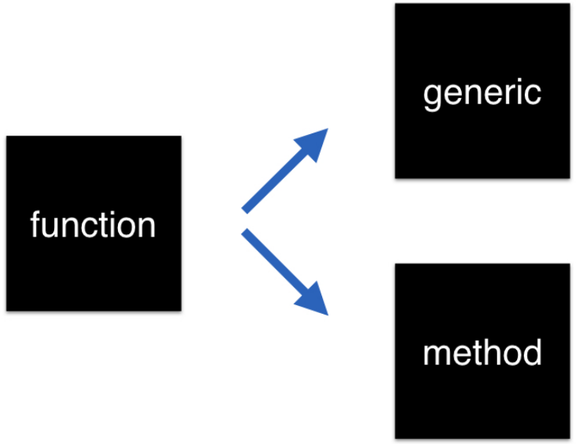
function overloading = input-dependent function behavior


S3

function


S3
generic
function


S3


function
generic
method

```
print
```

```
function (x, ...)
UseMethod("print")
<bytecode: 0x1062f0870>
<environment: namespace:base>
```

Methods are named generic.class

- `print.Date`
- `summary.factor`
- `unique.array`

Method signatures contain generic signatures

```
args(print)
```

```
function (x, ...)
NULL
```

```
args(print.Date)
```

```
function (x, max = NULL, ...)
NULL
```

pass arguments between methods with `...`

include it in both generic and methods

```
print.function
```

```
function (x, useSource = TRUE, ...)
.Internal(print.function(x, useSource, ...))
```

```
print.Date
```

```
function (x, max = NULL, ...)
{
    if (is.null(max))
        max <- getOption("max.print", 9999L)
    if (max < length(x)) {
        print(format(x[seq_len(max)]), max = max, ...)
        cat(" [ reached getOption(\"max.print\") -- omitted",
            length(x) - max, "entries ]\n")
    }
    else print(format(x), max = max, ...)
    invisible(x)
}
```

~~lower.leopard.case~~

~~lower.leopard.case~~

lower_snake_case

~~lower.leopard.case~~

lower_snake_case

lowerCamelCase

Summary

- Functions **split** into **generic + method**
- Methods named `generic.class`
- Method args **contain generic** args
- Include a `...` arg
- Use `lower_snake_case` or `lowerCamelCase`

Let's practice!

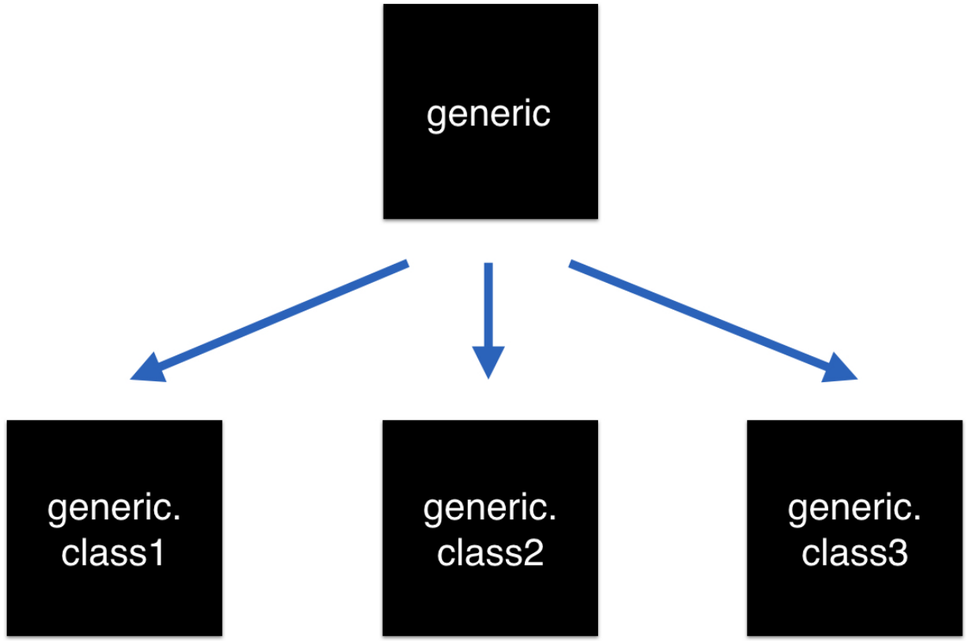
OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Methodical Thinking

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Richie Cotton
Data Evangelist at DataCamp

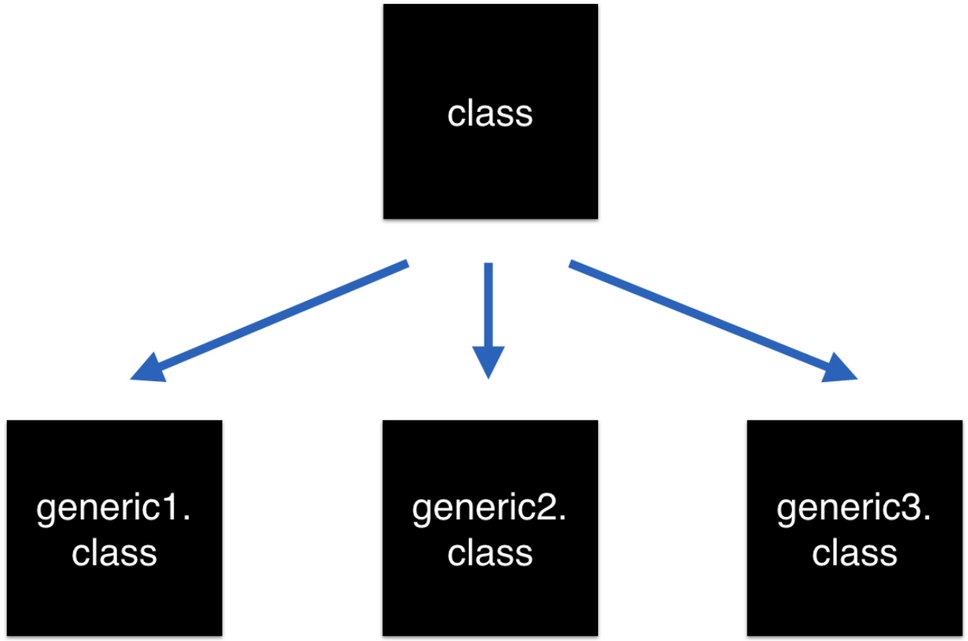
datacamp


generic
generic.
class1
generic.
class2
generic.
class3

```
methods
```

```
methods("mean") # or methods(mean)
```

```
mean.Date     mean.default  mean.difftime mean.POSIXct
mean.POSIXlt
see '?methods' for accessing help and source code
```


class
generic1.
class
generic2.
class
generic3.
class

```
methods(class = "glm") # or methods(class = glm)
```

```
add1            anova           coerce
confint         cooks.distance  deviance
drop1           effects         extractAIC
family          formula         influence
initialize      logLik          model.frame
nobs            predict         print
residuals       rstandard       rstudent
show            slotsFromS3     summary
vcov            weights
see '?methods' for accessing help and source code
```

`methods()` returns **S3** *and* **S4** methods

```
.S3methods(class = "glm")
```

```
add1            anova           confint
cooks.distance  deviance        drop1
effects         extractAIC      family
formula         influence       logLik
model.frame     nobs            predict
print           residuals       rstandard
rstudent        summary         vcov
weights
see '?methods' for accessing help and source code
```

```
.S4methods(class = "glm")
```

```
coerce      initialize  show        slotsFromS3
see '?methods' for accessing help and source code
```

Summary

- `methods()` **finds methods** for a generic
- `...` or for a **class**
- `.S3methods()` finds **only S3** methods

Let's practice!

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Method Lookup for Primitive Generics

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Richie Cotton
Data Evangelist at DataCamp

datacamp

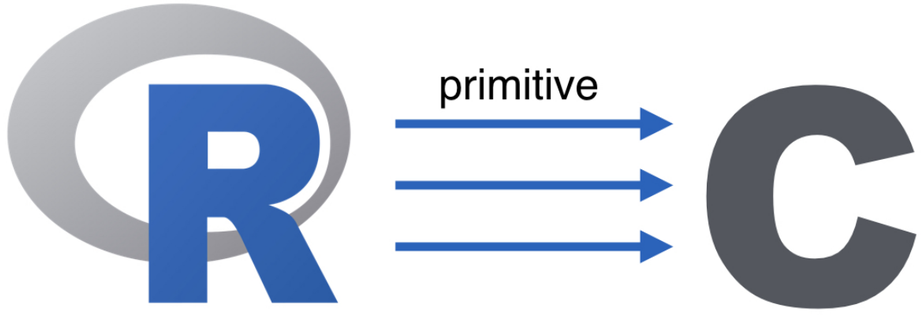
- Writing code
- Debugging code
- Maintaining code

- Running code

R vs. C

- **C** code often runs faster
- **R** code is usually easier to **write**
- ... and easier to **debug**


R
C


primitive
R
C

```
exp
```

```
function (x)  .Primitive("exp")
```

```
+
```

```
function (e1, e2)  .Primitive("+")
```

```
if
```

```
.Primitive("if")
```

```
sin
```

```
function (x)  .Primitive("sin")
```

```
-
```

```
function (e1, e2)  .Primitive("-")
```

```
for
```

```
.Primitive("for")
```

```
.S3PrimitiveGenerics
```

```
"anyNA"          "as.character"   "as.complex"
"as.double"      "as.environment" "as.integer"
"as.logical"     "as.call"        "as.numeric"
"as.raw"         "c"              "dim"
"dim<-"          "dimnames"       "dimnames<-"
"is.array"       "is.finite"      "is.infinite"
"is.matrix"      "is.na"          "is.nan"
"is.numeric"     "length"         "length<-"
"levels<-"       "names"          "names<-"
"rep"            "seq.int"        "xtfrm"
```

```
all_of_time <- c("1970-01-01", "2012-12-21")
as.Date(all_of_time)
```

```
"1970-01-01" "2012-12-21"
```

```
class(all_of_time) <- "date_strings"
as.Date(all_of_time)
```

```
Error in as.Date.default(all_of_time) :
  do not know how to convert 'all_of_time' to class"Date"
```

```
length(all_of_time)
```

```
2
```

Summary

- Some R functions are actually **written in C**
- The **primitive** interface gives **best performance**
- `.S3PrimitiveGenerics` lists **primitive S3 generics**
- Primitive generics **don't throw an error** when no method is found

Let's practice!

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Too Much Class

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Richie Cotton
Data Evangelist at DataCamp

datacamp

```
x <- c(1, 3, 6, 10, 15)
class(x) <- c(
  "triangular_numbers", "natural_numbers", "numeric"
)
```

```
is.numeric(x)
```

```
TRUE
```

```
is.triangular_numbers(x)
```

```
Error: could not find function "is.triangular_numbers"
```

```
inherits(x, "triangular_numbers")
```

```
TRUE
```

```
inherits(x, "natural_numbers")
```

```
TRUE
```

```
inherits(x, "numeric")
```

```
TRUE
```

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

```
what_am_i.triangular_numbers <- function(x, ...) {
  message("I'm triangular numbers")
  NextMethod("what_am_i")
}
```

```
what_am_i.natural_numbers <- function(x, ...) {
  message("I'm natural numbers")
  NextMethod("what_am_i")
}
```

```
what_am_i.numeric <- function(x, ...) {
  message("I'm numeric")
}
```

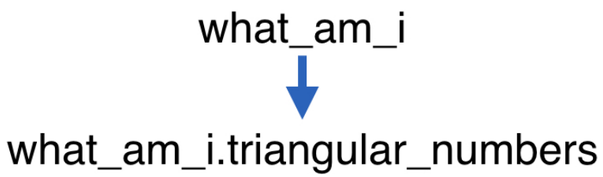
```
what_am_i(x)
```

```
I'm triangular numbers
I'm natural numbers
I'm numeric
```

what_am_i

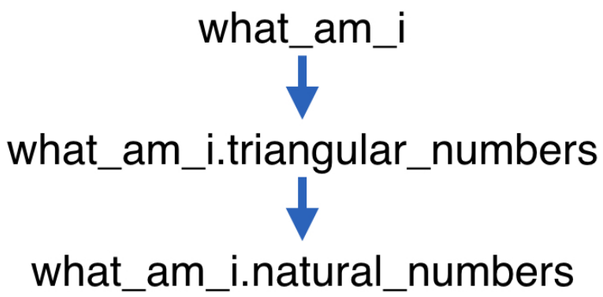
```
what_am_i(x)
```

```
I'm triangular numbers
I'm natural numbers
I'm numeric
```



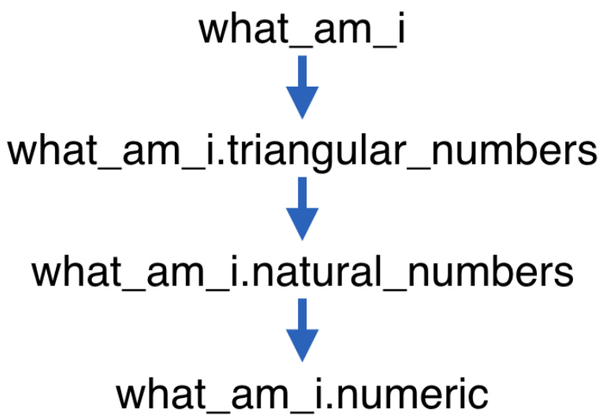
```
what_am_i(x)
```

```
I'm triangular numbers
I'm natural numbers
I'm numeric
```



```
what_am_i(x)
```

```
I'm triangular numbers
I'm natural numbers
I'm numeric
```



Summary

- **Multiple classes** are allowed
- Use `inherits()` to test for **arbitrary classes**
- Use `NextMethod()` to **chain method calls**

Let's practice!

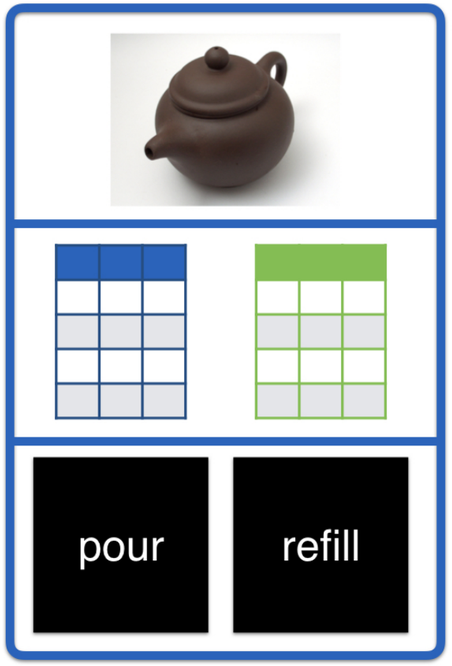
OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

The Object Factory

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

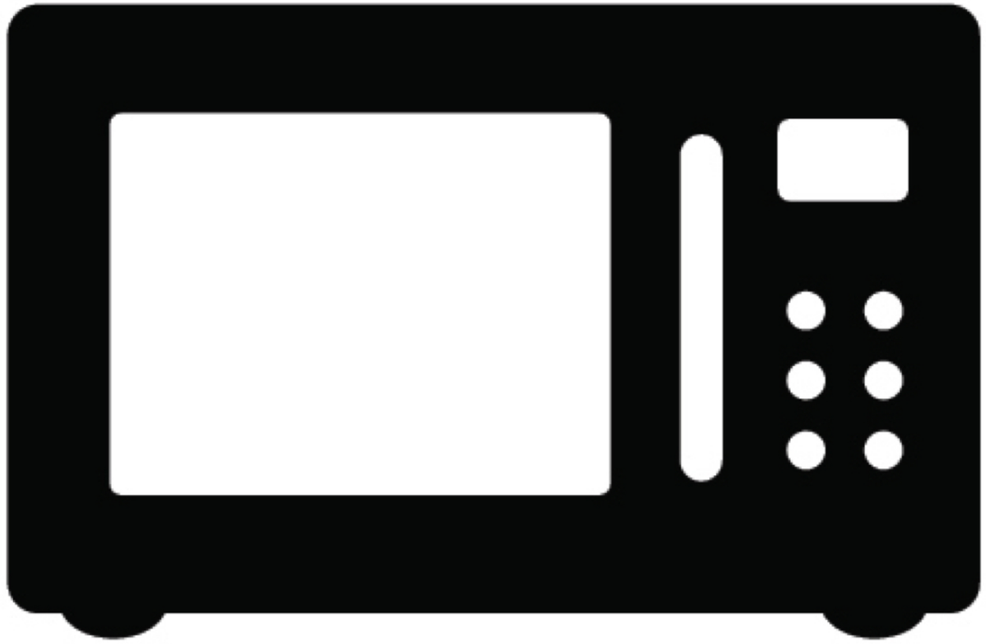
Richie Cotton
Data Evangelist at DataCamp

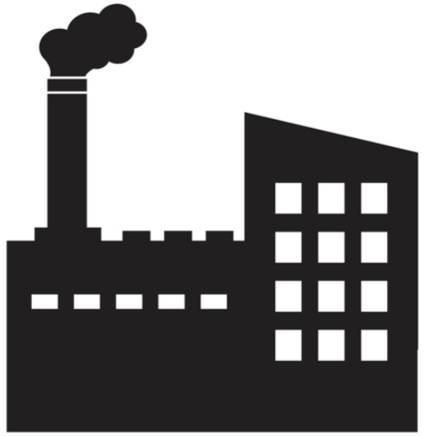
datacamp

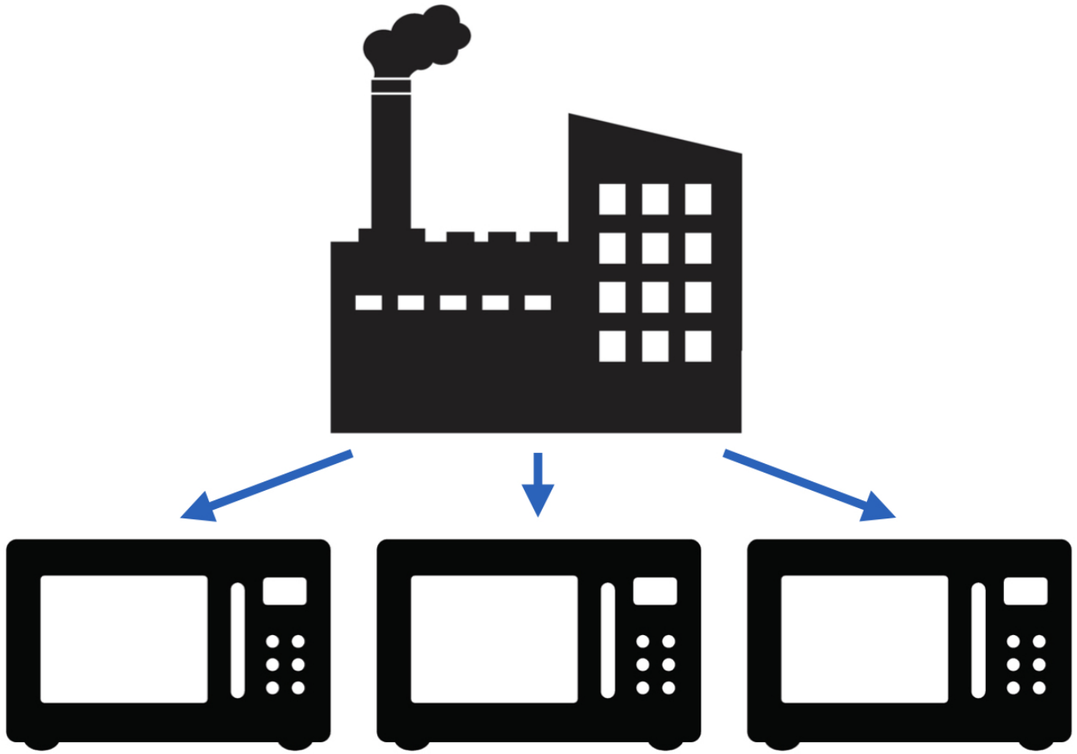

pour
refill

class generators are templates for objects

class generators are templates for objects a.k.a. factories







```
library(R6)
thing_factory <- R6Class(
  "Thing",
  private = list(
    a_field = "a value",
    another_field = 123
  )
)
```

Coming soon ...

public

active

```
a_thing <- thing_factory$new()
```

```
another_thing <- thing_factory$new()
```

```
yet_another_thing <- thing_factory$new()
```

Summary

- Load the **R6** package to work with R6!
- Define **class generators** with `R6Class()`
- Class names should be **UpperCamelCase**
- **Data fields** stored in `private` list
- Create objects with factory's `new()` method

Let's practice!

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Hiding Complexity with Encapsulation

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Richie Cotton
Data Evangelist at DataCamp

datacamp

Encapsulation

Encapsulation

```
microwave_oven_factory <- R6Class(
  "MicrowaveOven",
  private = list(
    power_rating_watts = 800,
    door_is_open = FALSE
  ),
  public = list(
    open_door = function() {
      private$door_is_open <- TRUE
    }
  )
)
```

`private$` accesses **private** elements

`self$` accesses **public** elements

Summary

- **Encapsulation = separating** implementation from UI
- Store **data** in `private` list
- Store **methods** in `public` list
- Use `private$` to access **private** elements
- `...` and `self$` to access **public** elements

Let's practice!

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Getting and Setting with Active Bindings

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Richie Cotton
Data Evangelist at DataCamp

CONTROLLED ACCESS ZONE

getting = read the data field

setting = write the data field

Active Bindings

defined like functions

accessed like data variables

```
thing_factory <- R6Class(
  "Thing",
  private = list(
    ..a_field = "a value",
    ..another_field = 123
  ),
  active = list(
    a_field = function() {
      if(is.na(private$..a_field)) {
        return("a missing value")
      }
      private$..a_field
    },
    another_field = function(value) {
      if(missing(value)) {
        private$..another_field
      } else {
        assert_is_a_number(value)
        private$..another_field <- value
      }
    }
  )
)
```

```
a_thing <- thing_factory$new()
a_thing$a_field
```

```
"a value"
```

```
a_thing$a_field <- "a new value"
```

```
Error in (function (value)  : a_field is read-only.
```

```
a_thing$another_field <- 456
```

```
a_thing$another_field <- "456"
```

```
Error in (function (value) : is_a_number :
value is not of class 'numeric'; it has class 'character'.
```

Summary

- **Control** private access with **active bindings**
- **Defined** like **functions**
- **Accessed** like **data**

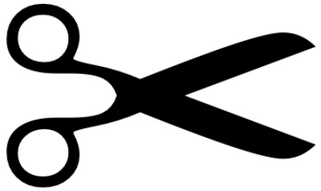
Let's practice!

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

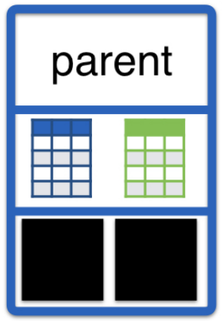
Propagating Functionality with Inheritance

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

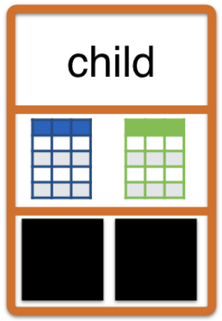
Richie Cotton
Data Evangelist at DataCamp



```
thing_factory <- R6Class(
  "Thing",
  private = list(
    a_field = "a value",
    another_field = 123
  ),
  public = list(
    do_something = function(x, y, z) {
      # do something here
    }
  )
)
```

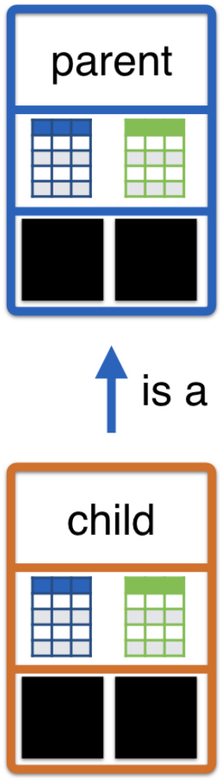


the class you inherit from

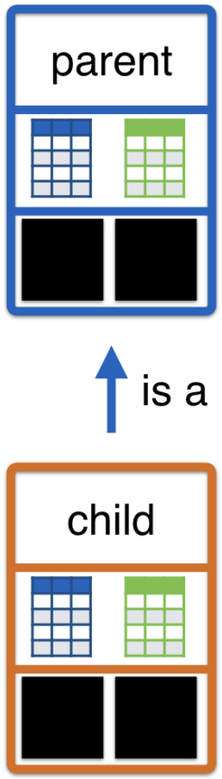


the class that inherits fields and methods

```
child_thing_factory <- R6Class(
  "ChildThing",
  inherit = thing_factory
  public = list(
    do_something_else = function() {
      # more functionality
    }
  )
)
```



- a fancy microwave is a microwave



- a fancy microwave is a microwave
- not all microwaves are fancy microwaves

```
a_thing <- thing_factory$new()
class(a_thing)
```

```
"Thing" "R6"
```

```
inherits(a_thing, "Thing")
```

```
TRUE
```

```
inherits(a_thing, "R6")
```

```
TRUE
```

```
a_child_thing <- child_thing_factory$new()
class(a_child_thing)
```

```
"ChildThing" "Thing"      "R6"
```

```
inherits(a_child_thing, "ChildThing")
```

```
TRUE
```

```
inherits(a_child_thing, "Thing")
```

```
TRUE
```

```
inherits(a_child_thing, "R6")
```

```
TRUE
```

Summary

- **Propagate functionality** using **inheritance**
- Use the `inherit` **arg** to `R6Class()`
- **Children get** their **parent's functionality**
- ... but the **converse is not true**

Let's practice!

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Embrace, Extend, Override

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Richie Cotton
Data Evangelist at DataCamp

```
thing_factory <- R6Class(
  "Thing",
  public = list(
    do_something = function() {
      message("the parent do_something method")
    }
  )
)
```

```
child_thing_factory <- R6Class(
  "ChildThing",
  inherit = thing_factory,
  public = list(
    do_something = function() {
      message("the child do_something method")
    },
    do_something_else = function() {
      message("the child do_something_else method")



    }
  )
)
```

```
a_child_thing <- child_thing_factory$new()
```

```
a_child_thing$do_something()
```

```
the child do_something method
```

`private$` accesses **private** fields

`self$` accesses **public** methods in **self**

`super$` accesses **public** methods in **parent**

```
child_thing_factory <- R6Class(
  "ChildThing",
  inherit = thing_factory,
  public = list(
    do_something = function() {
      message("the child do_something method")
    },
    do_something_else = function() {
      message("the child do_something_else method")
     self$do_something()
     super$do_something()
    }
  )
)
```

```
a_child_thing <- child_thing_factory$new()
```

```
a_child_thing$do_something_else()
```

```
the child do_something_else method
the child do_something method
the parent do_something method
```

Summary

- **Override** by giving the **same name**
- **Extend** by giving a **new name**
- `self$` accesses **public** methods in **self**
- `super$` accesses **public** methods in **parent**

Let's practice!

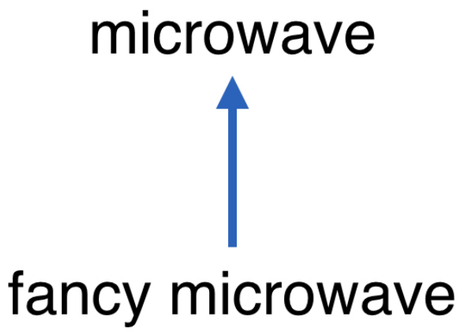
OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

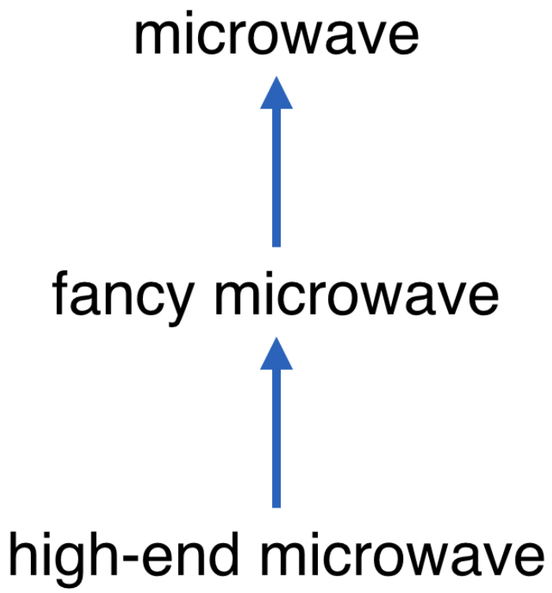
Multiple Levels of Inheritance

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Richie Cotton
Data Evangelist at DataCamp


CLOCK
AM / PM
AUTO NITE LIGHT
TIMER
CONVENIENCE COOKING
POPCORN
REHEAT
BEVERAGE
COOK
POTATO
VEGETABLE
SNACKS
TIME COOK
DEFROST AUTO / TIME
REMINDER
HELP
DELAY START
START
CLEAR OFF
POWER LEVEL
SURFACE LIGHT
BEEPER VOLUME
TURNTABLE
VENT FAN


microwave
fancy microwave


microwave
fancy microwave
high-end microwave

```
thing_factory <- R6Class(
  "Thing"
)
```

```
child_thing_factory <- R6Class(
  "ChildThing",
  inherit = thing_factory
)
```

```
grand_child_thing_factory <- R6Class(
  "GrandChildThing",
  inherit = child_thing_factory
)
```

```
thing_factory <- R6Class(
  "Thing",
  public = list(
    do_something = function() {
      message("the parent do_something method")
    }
  )
)
```

```
child_thing_factory <- R6Class(
  "ChildThing",
  inherit = thing_factory,
  public = list(
    do_something = function() {
      message("the child do_something method")
    }
  )
)
```

```
grand_child_thing_factory <- R6Class(
  "GrandChildThing",
  inherit = child_thing_factory,
  public = list(
    do_something = function() {
      message("the grand-child do_something method")
      super$do_something()
      super$super$do_something()
    }
  )
)
```

```
a_grand_child_thing <- grand_child_thing_factory$new()
a_grand_child_thing$do_something()
```

```
the grand-child do_something method
the child do_something method
Error in a_grand_child_thing$do_something():
  attempt to apply non-function
```

```
child_thing_factory <- R6Class(
  "ChildThing",
  inherit = thing_factory,
  public = list(
    do_something = function() {
      message("the child do_something method")
    }
  ),
  active = list(
    super_ = function() super
  )
)
```

```
grand_child_thing_factory <- R6Class(
  "GrandChildThing",
  inherit = child_thing_factory,
  public = list(
    do_something = function() {
      message("the grand-child do_something method")
      super$do_something()
      super$super_$do_something()
    }
  )
)
```

```
a_grand_child_thing <- grand_child_thing_factory$new()
a_grand_child_thing$do_something()
```

```
the grand-child do_something method
the child do_something method
the parent do_something method
```

Summary

- **R6** objects can only **access their direct parent**
- **Intermediate** classes can **expose their parent**
- Use an **active binding** named `super_`
- `super_` should simply return `super`

Let's practice!

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Environments, Reference Behavior, & Shared Fields

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Richie Cotton
Curriculum Architect at DataCamp

datacamp

list

list

environment

```
env <- new.env()
```

```
lst <- list(x = pi ^ (1:5), y = matrix(month.abb, 3))
```

```
env$x <- pi ^ (1:5)
env[["y"]] <- matrix(month.abb, 3)
```

```
lst
```

```
$x
3.141593   9.869604  31.006277  97.409091 306.019685

$y
     [,1]  [,2]  [,3]  [,4]
[1,] "Jan" "Apr" "Jul" "Oct"
[2,] "Feb" "May" "Aug" "Nov"
[3,] "Mar" "Jun" "Sep" "Dec"
```

```
env
```

```
<environment: 0x103f3dfc8>
```

```
ls.str(lst)
```

```
x :  num [1:5] 3.14 9.87 31.01 97.41 306.02
y :  chr [1:3, 1:4] "Jan" "Feb" "Mar" "Apr" "May" ...
```

```
ls.str(env)
```

```
x :  num [1:5] 3.14 9.87 31.01 97.41 306.02
y :  chr [1:3, 1:4] "Jan" "Feb" "Mar" "Apr" "May" ...
```

```
lst2 <- lst
(lst$x <- exp(1:5))
```

```
2.718282   7.389056  20.085537  54.598150 148.413159
```

```
lst2$x
```

```
3.141593   9.869604  31.006277  97.409091 306.019685
```

```
identical(lst$x, lst2$x)
```

```
FALSE
```

```
env2 <- env
(env$x <- exp(1:5))
```

```
2.718282   7.389056  20.085537  54.598150 148.413159
```

```
env2$x
```

```
2.718282   7.389056  20.085537  54.598150 148.413159
```

```
identical(env$x, env2$x)
```

```
TRUE
```

copy by value

copy by value

copy by reference

```
thing_factory <- R6Class(
  "Thing",
  private = list(
    shared = {
      e <- new.env()
      e$a_shared_field = 123
      e
    }
  ),
  active = list(
    a_shared_field = function(value) {
      if(missing(value)) {
        private$shared$a_shared_field
      } else {
        private$shared$a_shared_field <- value
      }
    }
  )
)
```

```
a_thing <- thing_factory$new()
another_thing <- thing_factory$new()
```

```
a_thing$a_shared_field
```

```
123
```

```
another_thing$a_shared_field
```

```
123
```

```
a_thing$a_shared_field <- 456
another_thing$a_shared_field
```

```
456
```

Summary

- **Create** environments with `new.env()`
- **Manipulate** them using **list syntax**
- **Environments** copy **by reference**
- **Share R6 fields** using an **environment** field

Let's practice!

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Cloning R6 Objects

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Richie Cotton
Curriculum Architect at DataCamp

- Environments use copy by reference
- So do **R6** objects

```
thing_factory <- R6Class(
  "Thing",
  private = list(
    ..a_field = 123
  ),
  active = list(
    a_field = function(value) {
      if(missing(value)) {
        private$..a_field
      } else {
        private$..a_field <- value
      }
    }
  )
)
```

```
a_thing <- thing_factory$new()
a_copy <- a_thing
a_thing$a_field <- 456
```

```
a_copy$a_field
```

```
456
```

`clone()` copies by value

```
a_clone <- a_thing$clone()
```

```
a_thing$a_field <- 789
a_clone$a_field
```

```
456
```

```
container_factory <- R6Class(
  "Container",
  private = list(
    ..thing = thing_factory$new()
  ),
  active = list(
    thing = function(value) {
      if(missing(value)) {
        private$..thing
      } else {
        private$..thing <- value
      }
    }
  )
)
```

```
a_container <- container_factory$new()
a_clone <- a_container$clone()
```

```
a_container$thing$a_field <- "a new value"
a_clone$thing$a_field
```

```
"a new value"
```

```
a_deep_clone <- a_container$clone(deep = TRUE)
```

```
a_container$thing$a_field <- "a different value"
a_deep_clone$thing$a_field
```

```
"a new value"
```

Summary

- **R6** objects copy **by reference**
- **Copy** them **by value** using `clone()`
- `clone()` is **autogenerated**
- `clone(deep = TRUE)` is for **R6** fields

Let's practice!

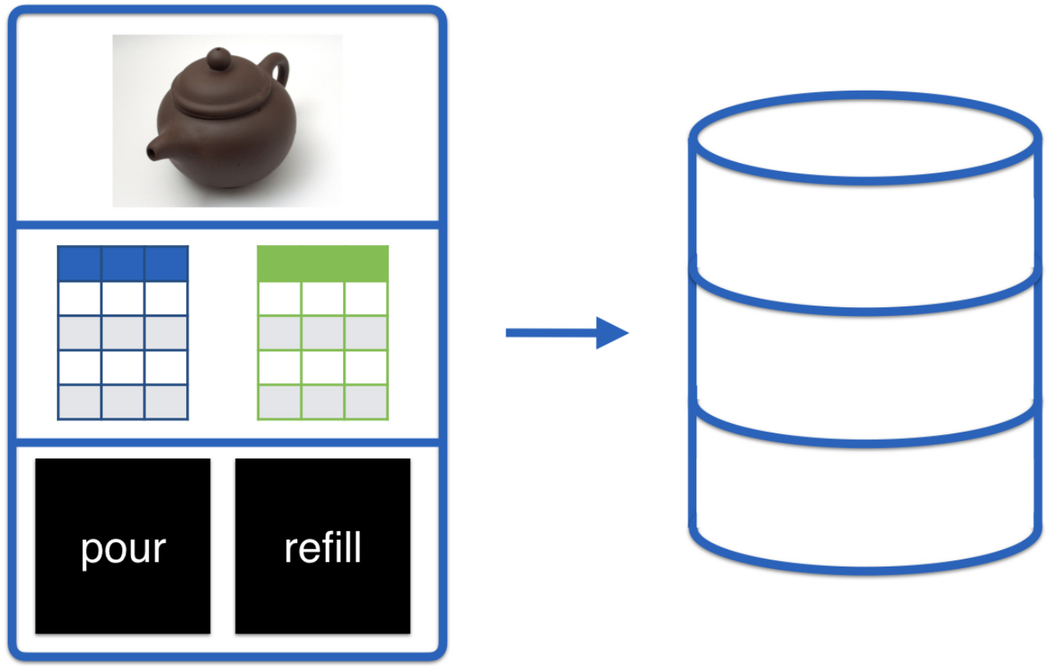
OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

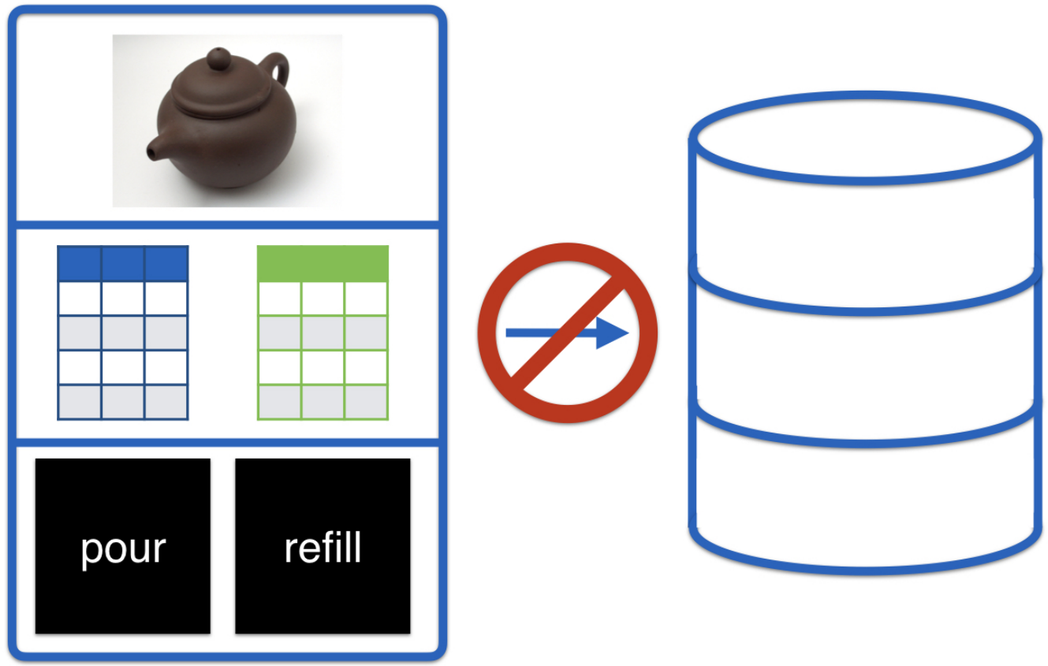
Shut it Down

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R

Richie Cotton
Curriculum Architect at DataCamp

datacamp


pour
refill


pour
refill

`initialize()` customizes **startup**

`finallize()` customizes **cleanup**

```
thing_factory <- R6Class(
  "Thing",
  private = list(
    ..a_field = 123
  ),
  public = list(
    initialize = function(a_field) {
      if(!missing(a_field)) {
        private$a_field = a_field
      }
    },
    finalize = function() {
      message("Finalizing the Thing")
    }
  )
)
```

```
a_thing <- thing_factory$new()
```

```
rm(a_thing)
```

```
gc()
```

```
Finalizing the Thing
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 443079 23.7     750400 40.1   592000 31.7
Vcells 718499  5.5    1308461 10.0  1092342  8.4
```

```
library(RSQLite)
database_manager_factory <- R6Class(
  "DatabaseManager",
  private = list(
   conn = NULL
  ),
  public = list(
    initialize = function(a_field) {
      private$conn <- dbConnect("some-database.sqlite")
    },
    finalize = function() {
      dbDisconnect(private$conn)
    }
  )
)
```

Summary

- `finalize()` **cleans up** after **R6** objects
- It is useful when **working with databases**
- It gets called during **garbage collection**

Let's practice!

OBJECT-ORIENTED PROGRAMMING WITH S3 AND R6 IN R