Chapter 14: S4

Ryan Heslin

October 22, 2022

Basics

Everything S4-related lives in `methods`.

```
library(methods)
```

The `setClass` constructor declares a class with given slots. `new` is the constructor.

```
setClass("Person",
  slots = c(
    name = "character",
    age = "numeric"
  )
)
john <- new("Person", name = "John Smith", age = NA_real_)
```

Yes, you are responsible for defining accessors, usually as S4 generics.

```
setGeneric("age", function(x) standardGeneric("age"))
```

```
[1] "age"
```

```
setGeneric("age<-", function(x, value) standardGeneric("age<-"))
```

```
[1] "age<-"
```

```r
setMethod("age", "Person", function(x) x@age)
setMethod("age<-", "Person", function(x, value) {
  x@age <- value
  x
})

age(john) <- 50
age(john)
```

```
[1] 50
```

1.

It has six slots, each a number storing a different unit of time, plus an odd one called `.Data`. The documentation lists 11 accessors.

```r
x <- lubridate::period(4, units = "years")
str(x)
```

```
Formal class 'Period' [package "lubridate"] with 6 slots
  ..@ .Data : num 0
  ..@ year  : num 4
  ..@ month : num 0
  ..@ day   : num 0
  ..@ hour  : num 0
  ..@ minute: num 0
```

2.

The docs say:

```
     Authors of formal (‘S4’) methods can provide documentation on
     specific methods, as well as overall documentation on the methods
     of a particular function.  The ‘"?"’ operator allows access to
     this documentation in three ways.

     The expression ‘methods?f’ will look for the overall documentation
     methods for the function ‘f’.  Currently, this means the
     documentation file containing the alias ‘f-methods’.

     There are two different ways to look for documentation on a
     particular method.  The first is to supply the ‘topic’ argument in
     the form of a function call, omitting the 'type' argument.  The
     effect is to look for documentation on the method that would be
     used if this function call were actually evaluated. See the
     examples below.  If the function is not a generic (no S4 methods
     are defined for it), the help reverts to documentation on the
     function name.

     The '"?"' operator can also be called with 'doc_type' supplied as
     'method'; in this case also, the 'topic' argument is a function
     call, but the arguments are now interpreted as specifying the
     class of the argument, not the actual expression that will appear
     in a real call to the function.  See the examples below.

     The first approach will be tedious if the actual call involves
     complicated expressions, and may be slow if the arguments take a
     long time to evaluate.  The second approach avoids these issues,
     but you do have to know what the classes of the actual arguments
     will be when they are evaluated.

     Both approaches make use of any inherited methods; the signature
     of the method to be looked up is found by using 'selectMethod'
     (see the documentation for 'getMethod').  A limitation is that
     methods in packages (as opposed to regular functions) will only be
     found if the package exporting them is on the search list, even if
     it is specified explicitly using the '?package::generic()'
     notation.
```

So `methods?f` brings up general method documentation, and documentation on particualr methods can be found by either specifying types in the call, or by passing objects with the relevant types, as in the example `?combo(1:10, letters)`

Classes

`setClass` takes three arguments

- Class name
- Named character vector of names and classes of slots
- List of slot prototypes - optional, but strongly recommended

```r
setClass("Person",
  slots = c(
    name = "character",
    age = "numeric"
  ),
  prototype = list(
    name = NA_character_,
    age = NA_real_
  )
)

setClass("Employee",
  contains = "Person",
  slots = c(
    boss = "Person"
  ),
  prototype = list(
    boss = new("Person")
  )
)

me <- new("Person", name = "Hadley")
str(me)
```

```
Formal class 'Person' [package ".GlobalEnv"] with 2 slots
  ..@ name: chr "Hadley"
  ..@ age : num NA

#> Formal class 'Person' [package ".GlobalEnv"] with 2 slots
#>   ..@ name: chr "Hadley"
#>   ..@ age : num NA
```

`is` tests inheritance:

```
is(new("Person"))
```

```
[1] "Person"

#> [1] "Person"
is(new("Employee"))
```

```
[1] "Employee" "Person"

#> [1] "Employee" "Person"
```

In R, as in Python, classes are both defined and instantiated at runtime. Modifying a class after declaring it is an error:

```
setClass("A", slots = c(x = "numeric"))
a <- new("A", x = 10)

setClass("A", slots = c(a_different_slot = "numeric"))
a
```

```
An object of class "A"
Slot "a_different_slot":
```

```
Error in slot(object, what): no slot of name "a_different_slot" for this object of class "A"
```

```
#> An object of class "A"
#> Slot "a_different_slot":
#> Error in slot(object, what): no slot of name "a_different_slot" for this object
#> of class "A"
```

You should define a helper constructor for users and reserve `new` for you, the developer:

```
Person <- function(name, age = NA) {
  age <- as.double(age)

  new("Person", name = name, age = age)
}

Person("Hadley")
```

```
Person
  Name: Hadley
  Age:  NA
```

```
#> An object of class "Person"
#> Slot "name":
#> [1] "Hadley"
#>
#> Slot "age":
#> [1] NA
```

`setValidity` converts a function into a validator for the class:

```
setValidity("Person", function(object) {
  if (length(object@name) != length(object@age)) {
    "@name and @age must be same length"
  } else {
    TRUE
  }
})
```

```
Class "Person" [in ".GlobalEnv"]

Slots:

Name:       name       age
```

```
Class: character   numeric

Known Subclasses: "Employee"
```

`new` will call this validator, though `validObject` can check objects for you.

1.

Some fields are deprecated, so I omit them.

```
args(utils::person)
```

```
function (given = NULL, family = NULL, middle = NULL, email = NULL,
    role = NULL, comment = NULL, first = NULL, last = NULL)
NULL
```

```
setOldClass("list")
setClassUnion("listORcharacter", c("list", "character"))
fields <- names(formals(utils::person))
fields <- fields[!fields %in% c("first", "last")]
n_fields <- length(fields)

slots <- rep("character", length.out = n_fields)
names(slots) <- fields
slots[c("comment", "role")] <- "listORcharacter"

prototypes <- replicate(NA_character_, n = n_fields, simplify = FALSE)

names(prototypes) <- names(slots)
prototypes[c("comment", "role")] <- list(NULL)

setClass("MyPerson", slots = slots, prototype = rep(NA_character_, length.out = length(fields)))

all_slots_same_length <- function(object) {
  slots <- slotNames(object)
  out <- TRUE

  if (length(slots) > 0) {
    target <- length(slot(object, slots[[1]]))
    for (slt in slots[-1]) {
      this_length <- length(slot(object, slt))
      if (this_length != target) {
        out <- paste0("@", slt, " must have length ", target, " not ", this_length)
        break
      }
    }
  }
  out
}
setValidity("MyPerson", all_slots_same_length)
```

```
Class "MyPerson" [in ".GlobalEnv"]

Slots:

Name:             given          family          middle
Class:        character       character       character

Name:             email            role         comment
Class:        character listORcharacter listORcharacter
```

```
me <- new("MyPerson",
  given = "Ryan", family = "Heslin", middle = "William",
  email = "not.publicly.available@gmail.com", role = "Author", comment = "Ugh, hate that guy"
)
me@role <- list("underachiever")
```

2.

It creates a virtual class: one useful as a parent, but not possible to instantiate.

```
setClass("dummy")
new("dummy")
```

```
Error in new("dummy"): trying to generate an object from a virtual class ("dummy")
```

3.

Use a helper to convert to integer codes before instantiating. Maybe allow an option for NA as a level.

```
setClass("my_factor", contains = "integer",  slots = c( levels = "character", labels = "character", ord
prototype = list( levels = character(), labels = character(), ordered = FALSE,
NA_level = FALSE )
```

For dates, both should inherit from a virtual class. `POSIXlt` needs various date components (month, year, etc.), plus a timezone and a format. The time fields should default to zero. `POSIXct` just needs to keep track of seconds since the UNIX epoch, leaving formatting to the print method.

For data frames:

```
new("MyDataFrame", contains = "list",  slots = c(data = "list", rownames = "character", colnames = "cha
colnames = character() ))
```

We would have to validate the lengths of `rownames` and `colnames`.

Generics and Methods

Generics do method dispatch. Don't wrap the generic name in braces, for some obscure reason.

```
setGeneric("myGeneric", function(x) standardGeneric("myGeneric"))
```

```
[1] "myGeneric"
```

Only one argument should be used: `signature`, controlling the arg used for dispatch. Arguments can be included to force methods to use them.

```
setGeneric("myGeneric",
  function(x, ..., verbose = TRUE) standardGeneric("myGeneric"),
  signature = "x"
)
```

```
[1] "myGeneric"
```

`setMethod` has a `signature` argument that can take multiple values, enabling multiple dispatch

```
setMethod("myGeneric", "Person", function(x) {
  # method implementation
})
```

The `show` method handles printing. You have to retrieve the args for a generic before writing a method for it.

```
args(getGeneric("show"))
```

```
function (object)
NULL
```

```
#> function (object)
#> NULL
```

Example:

```r
setMethod("show", "Person", function(object) {
  cat(is(object)[[1]], "\n",
    "  Name: ", object@name, "\n",
    "  Age:  ", object@age, "\n",
    sep = ""
  )
})
```

Define reusable generics so interface components can be shared across classes:

```r
setGeneric("name", function(x) standardGeneric("name"))
```

```
[1] "name"
```

```r
setMethod("name", "Person", function(x) x@name)

name(john)
```

```
[1] "John Smith"
```

```
#> [1] "John Smith"
```

Setters, if provided, should always validate.

```r
setGeneric("name<-", function(x, value) standardGeneric("name<-"))
```

```
[1] "name<-"
```

```r
setMethod("name<-", "Person", function(x, value) {
  x@name <- value
  validObject(x)
  x
})

name(john) <- "Jon Smythe"
name(john)
```

```
[1] "Jon Smythe"
```

```
#> [1] "Jon Smythe"
```

```
name(john) <- letters
```

```
Error in validObject(x): invalid class "Person" object: @name and @age must be same length
```

```
#> Error in validObject(x): invalid class "Person" object: @name and @age must be
#> same length
```

1.

```
setGeneric("age", function(x) standardGeneric("age"))
```

```
[1] "age"
```

```
setMethod("age", "Person", function(x) x@age)
age(john)
```

```
[1] 50
```

2.

`setGeneric` actually creates the generic, while the function passed to it invokes it.

3.

If the name of the class is hard-coded in `show`, subclasses will print the name of the parent class.

4.

We get a warning, but the method still works.

```r
setMethod("age", "Person", function(y) x@age)
age(john)
```

```
[1] 50
```

Method Dispatch

This gets complicated because S4 supports both multiple dispatch and multiple inheritance.

The general rule: start with the most specific class of teh arguments, then work upward through parent classes until a method is found.

Methods may be defined for two pseudoclsses:

- ANY, matching all classes
- MISSING, matching a `NA` argument

  If a class has multiple parents, the nearest one with a defined method is chosen (e.g., a parent over a grandparent). If the distance is the same, the method is "ambiguous." This triggers a warning, and the alphabetically earliest class (seriously?) gets picked. Never allow this to happen.

  Multiple inheritance makes designing clean, unambiguous method graphs a nightmare, so avoid it where possible. Multiple dispatch isn't so bad, because there are often fewer class combinations to deal with. Rule of thumb: if predicting dispatch requires a diagram, simplify.

3.

Because there are multiple terminal classes equidistant from the classes of the arguments, you would have to define methods for each combination, and then the result would still be ambiguous.

S4 and S3

The `slots` and `contains` arguments can use S3 classes only if they are prepared using `setOldClass`:

```r
setOldClass("data.frame")
setOldClass(c("ordered", "factor"))
setOldClass(c("glm", "lm"))
```

Better to implement them directly in S4:

```
setClass("factor",
  contains = "integer",
  slots = c(
    levels = "character"
  ),
  prototype = structure(
    integer(),
    levels = character()
  )
)
setOldClass("factor", S4Class = "factor")
```

S4 classes that inherit from an S3 class have a special `.Data` slot that contains the S3 object:

```
RangedNumeric <- setClass(
  "RangedNumeric",
  contains = "numeric",
  slots = c(min = "numeric", max = "numeric"),
  prototype = structure(numeric(), min = NA_real_, max = NA_real_)
)
rn <- RangedNumeric(1:10, min = 1, max = 10)
```

Converting and S3 generic to an S4 makes it the `ANY` method:

```
setGeneric("mean")
```

```
[1] "mean"
```

```
selectMethod("mean", "ANY")
```

```
Method Definition (Class "derivedDefaultMethod"):

function (x, ...)
UseMethod("mean")
<bytecode: 0x55ede94e8d30>
<environment: namespace:base>

Signatures:
        x
target  "ANY"
defined "ANY"
```

```
#> Method Definition (Class "derivedDefaultMethod"):
#>
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0x7f7fad935c68>
#> <environment: namespace:base>
#>
#> Signatures:
#>         x
#> target  "ANY"
#> defined "ANY"
```

This can be, but shouldn't be, done with regular functions.

1.

```r
setClass("factor",
  contains = "integer",
  slots = c(
    levels = "character",
    labels = "character",
    ordered = "logical",
    NA_level = "logical"
  ),
  prototype = structure(
    integer(),
    levels = character(),
    labels = character(),
    orderd = FALSE,
    NA_level = FALSE
  )
)
setOldClass("factor", S4Class = "factor")
```

2.

```r
setGeneric("length")
```

```
[1] "length"
```

```r
setMethod("length", "Person", function(x) length(x@name))
length(john)
```

```
[1] 1
```