Object-Oriented Programming in R

Kylie Ariel Bemis

4 February 2020

References for Object-Oriented Programming in R

Advanced R by Hadley Wickam

- `Book freely available at http://adv-r.had.co.nz`

R Language Definition by the R Core Team

- `https://cran.r-project.org/doc/manuals/R-lang.html`

What is a data frame?

```
head(cars)
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

What is a data frame?

```
typeof(cars)
```

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

```
attributes(cars)
```

```
## $names
## [1] "speed" "dist"
##
## $class
## [1] "data.frame"
##
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
## [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
## [47] 47 48 49 50
```

A data frame is a list?

If a data frame is a list, then why does it print differently than a list?

Consider the following code:

```
plot(cars)
plot(cars$speed, cars$dist)
plot(dist ~ speed, data=cars)
```

What does each do?

A data frame is a class

A data frame is an S3 **class** that builds on top of the basic `list` data type.

There are specialized `print()` and `plot()` methods (among many others!) that change behavior depending on what **class** of object they receive.

Introduction to Object-Oriented Programming in R

Object-oriented programming (OOP) is a way of organizing code around commonly re-used data "classes" and "methods".

A **class** is a blueprint for a way of organizing data.

- E.g., a `data.frame` is a class for heterogenous, tabular data

An **object** is a particular instance of a class.

- E.g., `iris` and `mtcars` are particular instances of `data.frames`.

Using **inheritance** allows subclasses to specialize superclasses.

- E.g., a `tibble` inherits most of its behavior from `data.frame`.

A **method** is a function associated with behavior specialized to a particular class. In R this is done using **generic functions**.

- E.g., `plot` is a generic function. It works differently depending on what kind of object is being plotted.

A simple example

Consider a "pet simulator" game. It may consist of the following elements:

- An `Animal` class with child classes `Cat` and `Dog`.
- A generic function called `speak`.
- A `speak` method for both the `Cat` and `Dog` classes
- An object named `Mittens` as an instance of the `Cat` class
- An object named `Duke` as an instance of the `Dog` class
- `speak(Mittens)`
  - "Meow!"
- `speak(Duke)`
  - "Woof!"

OOP in R versus other languages

In most object-oriented programming languages like C++ and Java, *methods belong to classes*. This relationship can be seen in the way they call their methods are called via `object.method()`:

- E.g., `Mittens.speak()`
- E.g., `Duke.speak()`

R takes a functional programming approach to OOP, so that *methods belong to generic functions*. This relationship can be seen in how methods in R are called via `method(object)`.

- E.g., `speak(Mittens)`
- E.g., `speak(Duke)`

This may seem confusing at first if you are familiar with OOP from a langauge like C++ or Java, but it's just a different way of thinking about OOP.

Object Systems in R

There are two major object-oriented programming systems in R:

- S3 classes:
  - Very simple class system
  - No formal class definitions
  - Single dispatch (methods only specialized on first argument)
- S4 classes:
  - More complex class system
  - Formal class definitions
  - Multiple dispatch (methods specialized on multiple arguments)

When to use which?

- Use S3 for simple data structures without complex dependencies
- Use S4 for more complex data structures

S3 is more common in base R and CRAN packages.

S4 is more common in Bioconductor packages.

Exceptions: RC

There is a third OOP system in R called Reference Classes, which we won't talk about in this class, because they break fundamental assumptions about data in R. They are useful, however, for classes which care about mutable state, such as GUIs.

The S3 OO System

The S3 class system is based on adding `attributes` to any of R's base types.

That means S3 classes are based on:

- `integer`
- `numeric`
- `character`
- `list`

...etc.

S3 classes are defined by their `class` attribute which can be accessed and set by the `class()` function.

What are some S3 classes you already know?

Existing S3 classes: `factor`

```
fc <- factor(c("a", "a", "b", "c"))
typeof(fc) # base type
```

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

```
class(fc) # class
```

```
## [1] "factor"
```

```
attributes(fc)
```

```
## $levels
## [1] "a" "b" "c"
##
## $class
## [1] "factor"
```

Existing S3 classes: data.frame

```
df <- data.frame(x=1:3, y=4:6)
typeof(df) # base type
```

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

```
class(df) # class
```

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

```
attributes(df)
```

```
## $names
## [1] "x" "y"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] 1 2 3
```

Existing S3 classes: `tibble`

```
library(tibble)
tb <- tibble(x=1:3, y=4:6)
typeof(tb) # base type
```

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

```
class(tb) # class -- tbl_df "inherits" from data.frame!
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
attributes(tb)
```

```
## $names
## [1] "x" "y"
##
## $row.names
## [1] 1 2 3
##
## $class
## [1] "tbl_df"     "tbl"        "data.frame"
```

Existing S3 classes: `lm`

```
fit <- lm(Sepal.Width ~ Sepal.Length, data=iris)
typeof(fit) # base type
```

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

```
class(fit) # class
```

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

```
attributes(fit)
```

```
## $names
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
##
## $class
## [1] "lm"
```

S3 Generic Functions

S3 generic functions are defined by a call to `UseMethod()`.

Consider the following generic functions for extracting the fitted response values and residuals from a model.

```
fitted
```

```
## function (object, ...)
## UseMethod("fitted")
## <bytecode: 0x7fa45d57b2b0>
## <environment: namespace:stats>
```

```
residuals
```

```
## function (object, ...)
## UseMethod("residuals")
## <bytecode: 0x7fa45c1d7990>
## <environment: namespace:stats>
```

S3 Methods

Use `methods()` to see the methods defined for various classes.

```
methods(fitted)
```

```
## [1] fitted.default*       fitted.isoreg*        fitted.kmeans*
## [4] fitted.nls*           fitted.smooth.spline*
## see '?methods' for accessing help and source code
```

```
methods(residuals)
```

```
## [1] residuals.default*       residuals.glm
## [3] residuals.HoltWinters*   residuals.isoreg*
## [5] residuals.lm             residuals.nls*
## [7] residuals.smooth.spline* residuals.tukeyline*
## see '?methods' for accessing help and source code
```

S3 Methods (cont'd)

S3 methods are defined by the naming convention `generic.class()`.

S3 method dispatch actually relies this naming scheme, and there are no additional requirements for defining an S3 method for a particular class.

For `residuals`, we saw a `residuals.lm` method, but there was no `fitted.lm` method.

If there is no specific method for a class, the default method is called, as defined by a function called `generic.default()`.

S3 Methods (cont'd)

We can use `getS3method` to find a particular S3 method.

```
getS3method("fitted", "default")
```

```
## function (object, ...)
## {
##     xx <- if ("fitted.values" %in% names(object))
##         object$fitted.values
##     else object$fitted
##     napredict(object$na.action, xx)
## }
## <bytecode: 0x7fa45d1a4828>
## <environment: namespace:stats>
```

S3 Methods (cont'd)

We can also view all the methods defined for a specific class using the `methods()` function.

```
methods(class="lm")
```

```
##  [1] add1             alias            anova            case.names
##  [5] coerce           confint          cooks.distance   deviance
##  [9] dfbeta           dfbetas          drop1            dummy.coef
## [13] effects          extractAIC       family           formula
## [17] hatvalues        influence        initialize       kappa
## [21] labels           logLik           model.frame      model.matri
## [25] nobs             plot             predict          print
## [29] proj             qr               residuals        rstandard
## [33] rstudent         show             simulate         slotsFromS3
## [37] summary          variable.names   vcov
## see '?methods' for accessing help and source code
```

Defining an S3 class

We can define an S3 class either by using `structure()`, or by simply setting the `class` attribute of an existing object.

```
a <- structure(list(), class="Animal")
class(a)
```

```
## [1] "Animal"
```

```
a <- list()
class(a) <- "Animal"
class(a)
```

```
## [1] "Animal"
```

Create a constructor for an S3 class

Typically, we should provide a constructor for our class to make it easier to create an object of that class. Note we use S3 inheritance in this example.

```
Cat <- function(name) structure(list(name=name),
                                 class=c("Cat", "Animal"))
Dog <- function(name) structure(list(name=name),
                                 class=c("Dog", "Animal"))
Cat("Mittens")
```

```
## $name
## [1] "Mittens"
##
## attr(,"class")
## [1] "Cat"    "Animal"
```

```
Dog("Duke")
```

```
## $name
## [1] "Duke"
##
## attr(,"class")
## [1] "Dog"    "Animal"
```

Define a custom `print` method

We can create a custom print method for our classes by defining a `print` method. This is the generic function that gets called whenever we type the name of a variable and hit "Enter".

To do this, we simply define a function following the naming convention `generic.class()`. We'd like all animals to use the same `print` method, so we'll define it on `Animal`.

```
print.Animal <- function(object) {
  print(paste("A", class(object)[1] ,
              "named", object$name))
}
Cat("Mittens")
```

```
## [1] "A Cat named Mittens"
```

```
Dog("Duke")
```

```
## [1] "A Dog named Duke"
```

Defining an S3 generic function

We now create a generic function for `speak` by creating a function that calls `UseMethod` with the name of our generic function.

```
speak <- function(object) UseMethod("speak")
```

Defining S3 methods

We now create a `speak` methods for our classes by following the naming convention `generic.class()`.

```
speak.default <- function(object) print("*weird noises*")
speak.Animal <- function(object) print("*weird animal noises*")
speak.Cat <- function(object) print("Meow!")
speak.Dog <- function(object) print("Woof!")
```

S3 example

Let's create two pets:

```
Mittens <- Cat("Mittens")
Duke <- Dog("Duke")
```

What will each of the following print out?

```
speak(Mittens)
speak(Duke)
speak(list())
```

```
speak(Mittens)
```

```
## [1] "Meow!"
```

```
speak(Duke)
```

```
## [1] "Woof!"
```

```
speak(list())
```

```
## [1] "*weird noises*"
```

The S4 OO System

The S4 class system works similarly to S3 from a user perspective, but adds mmore formality and rigor.

The S4 class system adds:

- Formal definitions of the data structure
  - S4 classes have `slots` (accessed via `@`) defined to be a specific data type
  - Inheritance is formally defined rather than via an attribute
- Method dispatch on multiple arguments, not only the first one
- Validity of the object can be rigorously checked

What are some S4 classes you already know?

Existing S4 classes: SummarizedExperiments

```
getClass("SummarizedExperiment")
```

```
## Class "SummarizedExperiment" [package "SummarizedExperiment"]
##
## Slots:
##
## Name:              colData              assays                NAMES
## Class:           DataFrame              Assays character_OR_NULL
##
## Name:    elementMetadata            metadata
## Class:           DataFrame                list
##
## Extends:
## Class "Vector", directly
## Class "Annotated", by class "Vector", distance 2
## Class "vector_OR_Vector", by class "Vector", distance 2
##
## Known Subclasses:
## Class "RangedSummarizedExperiment", directly, with explicit c
```

Existing S4 classes: MSnSets

```
getClass("MSnSet")
```

```
## Class "MSnSet" [package "MSnbase"]
##
## Slots:
##
## Name:       experimentData     processingData                  qu
## Class:              MIAPE         MSnProcess           data.fra
##
## Name:           assayData          phenoData        featureDa
## Class:          AssayData AnnotatedDataFrame AnnotatedDataFra
##
## Name:          annotation       protocolData  .__classVersion
## Class:          character AnnotatedDataFrame           Versio
##
## Extends:
## Class "eSet", directly
## Class "VersionedBiobase", by class "eSet", distance 2
## Class "Versioned", by class "eSet", distance 3
```

Existing S4 classes: DelayedArrays

```
getClass("DelayedArray")
```

```
## Class "DelayedArray" [package "DelayedArray"]
##
## Slots:
##
## Name:  seed
## Class:  ANY
##
## Extends:
## Class "DelayedUnaryIsoOp", directly
## Class "DelayedUnaryOp", by class "DelayedUnaryIsoOp", distanc
## Class "DelayedOp", by class "DelayedUnaryIsoOp", distance 3
## Class "Array", by class "DelayedUnaryIsoOp", distance 4
##
## Known Subclasses:
## Class "DelayedMatrix", directly, with explicit coerce
## Class "DelayedArray1", directly
## Class "RleArray", directly
## Class "RleMatrix", by class "RleArray", distance 2
```

Existing S4 classes: DataFrames

```
getClass("DataFrame")
```

```
## Class "DataFrame" [package "S4Vectors"]
##
## Slots:
##
## Name:           rownames              nrows           listData
## Class: character_OR_NULL            integer               list
##
## Name:        elementType   elementMetadata           metadata
## Class:         character DataTable_OR_NULL               list
##
## Extends:
## Class "DataTable", directly
## Class "SimpleList", directly
## Class "DataTable_OR_NULL", by class "DataTable", distance 2
## Class "List", by class "SimpleList", distance 2
## Class "Vector", by class "SimpleList", distance 3
## Class "list_OR_List", by class "SimpleList", distance 3
## Class "Annotated", by class "SimpleList", distance 4
```

Creating an S4 class

S4 classes are defined via a call to `setClass`.

```
setClass("Animal4",
         contains = "VIRTUAL",
         slots = c(name = "character"),
         validity = function(object) {
           if ( length(object@name) != 1 )
             stop("slot 'name' must be length 1")
         })
setClass("Cat4", contains = "Animal4")
setClass("Dog4", contains = "Animal4")
```

We do not expect to actually create `Animal` objects, so we make it a `VIRTUAL` object. (Virtual classes cannot be instantiated.)

Create a constructor for an S4 class

Using `setClass` doesn't actually create or modify an existing object, so we should create constructors for our classes.

New instances of S4 classes are created using `new()`, but it is rude to ask the user to call `new()` directly.

```
Cat4 <- function(name) new("Cat4", name=name)
Dog4 <- function(name) new("Dog4", name=name)
Mittens4 <- Cat4("Mittens")
Duke4 <- Dog4("Duke")
Mittens4
```

```
## An object of class "Cat4"
## Slot "name":
## [1] "Mittens"
```

```
Duke4
```

```
## An object of class "Dog4"
## Slot "name":
## [1] "Duke"
```

Define a custom show method

S4 classes use the show generic function instead of the print generic function. S4 methods are defined using setMethod.

```
setMethod("show", "Animal4", function(object) {
  print(paste("A", class(object)[1] ,
              "named", object@name))
})
Cat4("Mittens")
```

```
## [1] "A Cat4 named Mittens"
```

```
Dog4("Duke")
```

```
## [1] "A Dog4 named Duke"
```

Defining an S4 generic function

Just like S3 generic functions are defined by a call to `UseMethod()`, S4 generic functions are defined using `setGeneric()` with a call to `standardGeneric()`.

```
setGeneric("speak", function(object) standardGeneric("speak"))
```

```
## [1] "speak"
```

Defining S4 methods

S4 methods are defined using `setMethod()`, which takes the class signature for the method, and the function to call.

```
setMethod("speak", "Cat4", function(object) print("Meow!"))
setMethod("speak", "Dog4", function(object) print("Woof!"))
```

Viewing existing S4 methods

We can view existing S4 methods with `showMethods()`.

```
showMethods("speak")
```

```
## Function: speak (package .GlobalEnv)
## object="ANY"
## object="Cat4"
## object="Dog4"
```

What is the method for class “ANY”?

Viewing existing S4 methods (cont'd)

We can view a specific method using `selectMethod()`.

```
selectMethod("speak", "ANY")
```

```
## Method Definition (Class "derivedDefaultMethod"):
##
## function (object)
## UseMethod("speak")
## <bytecode: 0x7fa45f687e60>
##
## Signatures:
##         object
## target  "ANY"
## defined "ANY"
```

It's our S3 generic function!

S4 example

```
Mittens4 <- Cat4("Mittens")
Duke4 <- Dog4("Duke")
```

```
speak(Mittens4)
```

```
## [1] "Meow!"
```

```
speak(Duke4)
```

```
## [1] "Woof!"
```

Proteomics example: MassSpectrum S4 class and methods

The **"MassSpectrum-class.R"** and **"MassSpectrum-methods.R"** files include a basic implementation of an S4 class for working with raw mass spectra in R.

Take a moment to explore the provided implementation and understand how it works.

A more complete example of these classes are available in the `MSExample` package available at https://github.com/kuwisdelu/MSExample. We will use this package tomorrow, so you can install it by doing:

```
remotes::install_github("kuwisdelu/MSExample")
```