Object-oriented concepts

Marcus Wurzer

Everything in R is an object, and R is an object-oriented (OO) language. Object-oriented programming stands for a certain programming paradigm, where generic functions use adapted methods for different classes of objects. Main advantage: Users can simply utilize generic functions for different object classes without knowing details about the object class. `print()` is a typical example of a generic function, used to print many different classes of objects (e.g., data frames or lists) in adequate forms. R has three object-oriented systems:

- **S3**
- **S4**
- **RC**

In connection with that, it is also important to know another system called **base types**.

Base types

We already got to know some base types, namely atomic vectors, lists, functions, environments etc. In contrast to S3, S4, and RC, base types are not really an object system, because only the R core team can create new types. The already known `typeof()` is used to determine an object's base type. Base types are almost always written in C, and the other three object-oriented systems are built on top of them.

S3

S3 essentials

The first, the simplest, and still the most commonly used OO system. If we want to check if an object is an S3 object, we have to use `otype()` ("object type"), included in `pryr` package (`install.packages("pryr")`) - there is no such function in base R:

```
library(pryr)
df <- data.frame(x = 1:10, y = letters[1:10])
otype(df)
```

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

```
otype(df$x)
```

```
## [1] "base"
```

```
otype(df$y)
```

```
## [1] "base"
```

Generic functions: In S3, methods belong to functions, called generic functions (short: **generics**). They do not belong to objects or classes.
Method dispatch: The process of figuring out the correct method to call

For example, function `mean`:

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x564525e8cdb0>
## <environment: namespace:base>
```

`UseMethod()` is the function responsible for method dispatch. Using `ftype()` ("function type") works similar to `otype()` above:

```
ftype(mean)
```

```
## [1] "s3"      "generic"
```

There is a special type of functions that does not call `UseMethod()`, but does method dispatch in C code because the functions implemented in C. An example for that is `sum()`:

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

```
ftype(sum)
```

```
## [1] "primitive" "generic"
```

Given a class, S3 generics call the right S3 method. They have a certain style of naming. For example, the `print()` generic is called...

- `print.data.frame()` for data frames
- `print.table()` for tables
- `print.function()` for functions
- ...

In a similar way, naming is done for other generics like `summary()`, `plot()`, `mean()` etc. Using `methods()`, we can see all the methods that belong to a generic (only the first 20 of a long list are printed here):

```
methods("print")[1:20]
```

```
##  [1] "print.acf"                          "print.activeConcordance"
##  [3] "print.AES"                          "print.anova"
##  [5] "print.aov"                          "print.aovlist"
##  [7] "print.ar"                           "print.Arima"
##  [9] "print.arima0"                       "print.AsIs"
## [11] "print.aspell"                       "print.aspell_inspect_context"
## [13] "print.bibentry"                     "print.Bibtex"
## [15] "print.browseVignettes"              "print.by"
## [17] "print.bytes"                        "print.changedFiles"
## [19] "print.check_bogus_return"           "print.check_code_usage_in_package"
```

The other way round, we can also print all the generics for a given class, e.g., for linear models (class `lm`):

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

```
##  [1] add1           alias          anova          case.names     coerce
##  [6] confint        cooks.distance deviance       dfbeta         dfbetas
## [11] drop1          dummy.coef     effects        extractAIC     family
## [16] formula        hatvalues      influence      initialize     kappa
## [21] labels         logLik         model.frame    model.matrix   nobs
## [26] plot           predict        print          proj           qr
## [31] residuals      rstandard      rstudent       show           simulate
## [36] slotsFromS3    summary        variable.names vcov
```

```
## see '?methods' for accessing help and source code
```

Defining S3 objects

S3 doesn't have a formal definition of a class, we just set the class attribute ad hoc during creation (using `structure()`, which returns a given object with further arguments set) or after the fact using `class()`. `class()` without the assignment operator can then be used to check the class of an object:

```
foo <- structure(list(), class = "foo")
class(foo)
```

```
## [1] "foo"
```

```
foo <- list()
class(foo) <- "foo"
class(foo)
```

```
## [1] "foo"
```

S3 objects are usually built on top of lists, atomic vectors, and sometimes functions. If an object belongs to a certain class, this is called **inheritance** because the object inherits certain class characteristics. `inherits()` can be used to see if an object inherits from a specific class:

```
inherits(foo, "foo")
```

```
## [1] TRUE
```

```
inherits(foo, "numeric")
```

```
## [1] FALSE
```

The class can also be a vector of length > 1 if one object is a special case of another one. Then, the special class inherits from the least specific one. One example are Generalized Linear Models (GLMs) that inherit from Linear Models (LMs):

```
# estimate some GLM (here: Poisson regression)
counts <- c(18, 17, 15, 20, 10, 20, 25, 13, 12)
outcome <- gl(3, 1, 9)
treatment <- gl(3, 3)
glm.D93 <- glm(counts ~ outcome + treatment, family = poisson())
class(glm.D93) # behavior from the most ("glm") to the least specific ("lm")
```

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

```
inherits(glm.D93, "lm")
```

```
## [1] TRUE
```

```
inherits(glm.D93, "glm")
```

```
## [1] TRUE
```

It is possible to change the class of existing objects, but because correctness is not checked for S3, this may lead to non-desirable results:

```
glm.D93
```

```
##
## Call:  glm(formula = counts ~ outcome + treatment, family = poisson())
##
## Coefficients:
## (Intercept)     outcome2     outcome3   treatment2   treatment3
```

```
##   3.045e+00   -4.543e-01   -2.930e-01    1.218e-15    8.438e-16
##
## Degrees of Freedom: 8 Total (i.e. Null);  4 Residual
## Null Deviance:       10.58
## Residual Deviance: 5.129     AIC: 56.76
```

```
class(glm.D93) <- "data.frame"
glm.D93
```

```
##  [1] coefficients      residuals         fitted.values     effects
##  [5] R                 rank              qr                family
##  [9] linear.predictors deviance          aic               null.deviance
## [13] iter              weights           prior.weights     df.residual
## [17] df.null           y                 converged         boundary
## [21] model             call              formula           terms
## [25] data              offset            control           method
## [29] contrasts         xlevels
## <0 rows> (or 0-length row.names)
```

```
glm.D93$coefficients
```

```
##   (Intercept)      outcome2      outcome3    treatment2    treatment3
##  3.044522e+00 -4.542553e-01 -2.929871e-01  1.217511e-15  8.437695e-16
```

Creating new S3 methods and generics

New generics are added by creating a function that calls `UseMethod`:

```
f <- function(x) UseMethod("f")
```

Newly created generics need methods to be useful. Methods can be added by creating regular functions with the correct class name:

```
f.a <- function(x) "Class a" # Method for objects of class "a"
f.b <- function(x) "Class b" # Method for objects of class "b"
a <- structure(list(), class = "a")
b <- structure(list(), class = "b")
class(a)
```

```
## [1] "a"
```

```
class(b)
```

```
## [1] "b"
```

```
f(a)
```

```
## [1] "Class a"
```

```
f(b)
```

```
## [1] "Class b"
```

This also woks for existing generics like `print()` or `mean()`:

```
print.a <- function(x) "a"
print(a)
```

```
## [1] "a"
```

```r
mean.a <- function(x) "a"
mean(a)
```

```
## [1] "a"
```

In the latter case, this doesn't meet the expectations of existing code.

Check methods again:

```r
methods("mean")
```

```
## [1] mean.a        mean.Date     mean.default  mean.difftime mean.POSIXct
## [6] mean.POSIXlt  mean.quosure*
## see '?methods' for accessing help and source code
```

`mean.a` has been added.

S3 Method dispatch

`UseMethod()` creates a vector of function names and looks for each in turn. We use `f`, `f.a`, and `f.b` defined above and add a fall back method for otherwise unknown classes:

```r
f.default <- function(x) "Unknown class"
methods("f")
```

```
## [1] f.a       f.b       f.default
## see '?methods' for accessing help and source code
```

```r
f(structure(list(), class = "a"))
```

```
## [1] "Class a"
```

```r
f(structure(list(), class = c("b", "a"))) # b is the most specific one
```

```
## [1] "Class b"
```

```r
f(structure(list(), class = c("c", "b"))) # no method for c (the most specific one)
```

```
## [1] "Class b"
```

```r
f(structure(list(), class = "c"))
```

```
## [1] "Unknown class"
```

S4

Similar to S3, but adding more formality and rigor.

S4 essentials

There are several ways to recognize S4 objects, with and without usage of the `pryr` package. We create an S4 object using function `mle()` of the `stats4`-package (built-in, i.e., doesn't have to be installed):

```r
library(stats4)
y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)
nLL <- function(lambda) - sum(dpois(y, lambda, log = TRUE))
fit <- mle(nLL, start = list(lambda = 5), nobs = length(y))

otype(fit)
```

```
## [1] "S4"
```

```
isS4(fit)
```

```
## [1] TRUE
```

```
str(fit) # "Formal class" indicates an S4 object
```

```
## Formal class 'mle' [package "stats4"] with 10 slots
##   ..@ call     : language mle(minuslogl = nLL, start = list(lambda = 5), nobs = length(y))
##   ..@ coef     : Named num 11.5
##   .. ..- attr(*, "names")= chr "lambda"
##   ..@ fullcoef : Named num 11.5
##   .. ..- attr(*, "names")= chr "lambda"
##   ..@ fixed    : Named num NA
##   .. ..- attr(*, "names")= chr "lambda"
##   ..@ vcov     : num [1, 1] 1.05
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr "lambda"
##   .. .. ..$ : chr "lambda"
##   ..@ min      : num 42.7
##   ..@ details  :List of 6
##   .. ..$ par        : Named num 11.5
##   .. .. ..- attr(*, "names")= chr "lambda"
##   .. ..$ value      : num 42.7
##   .. ..$ counts     : Named int [1:2] 14 8
##   .. .. ..- attr(*, "names")= chr [1:2] "function" "gradient"
##   .. ..$ convergence: int 0
##   .. ..$ message    : NULL
##   .. ..$ hessian    : num [1, 1] 0.953
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr "lambda"
##   .. .. .. ..$ : chr "lambda"
##   ..@ minuslogl:function (lambda)
##   .. ..- attr(*, "srcref")= 'srcref' int [1:8] 3 8 3 59 8 59 3 3
##   .. .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x564526fd8b88>
##   ..@ nobs     : int 11
##   ..@ method   : chr "BFGS"
```

`is` either lists all the classes an object inherits from or can be used to test for a specific class:

```
is(fit)
```

```
## [1] "mle"
```

```
is(fit, "mle")
```

```
## [1] TRUE
```

```
is(fit, "numeric")
```

```
## [1] FALSE
```

`getGenerics()` lists all S4 generics, `getClasses` all S4 classes, and `showMethods()` all S4 methods:

```
# Not run: too much output
getGenerics()
getClasses()
showMethods()
```

Defining S4 objects

Ad hoc creation of an S4 object is not possible, we must define the representation of the class using `setClass()` and create a new object with `new()`. S4 classes have several properties, three of them are key properties:

- **Name**: The class identifier
- **Slots** (Fields): A named list defining slot names and permitted classes
- Class it inherits from = class it **contains**

```
setClass("Person",
  slots = list(name = "character", age = "numeric"))
setClass("Employee",
  slots = list(boss = "Person"),
  contains = "Person")

alice <- new("Person", name = "Alice", age = 40)
john <- new("Employee", name = "John", age = 20, boss = alice)
```

We use the `@` operator or `slot()` to access the slots of an S4 object (comparable to the `$` and `[[` operators for S3 objects):

```
alice@age
```

```
## [1] 40
```

```
slot(john, "boss")
```

```
## An object of class "Person"
## Slot "name":
## [1] "Alice"
##
## Slot "age":
## [1] 40
```

S4 objects will have a special `.Data` slot if they inherit from a base or S3 class type:

```
setClass("RangedNumeric",
  contains = "numeric",
  slots = list(min = "numeric", max = "numeric"))
rn <- new("RangedNumeric", 1:10, min = 1, max = 10)
rn@min
```

```
## [1] 1
```

```
rn@.Data
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Creating new S4 methods and generics

`setGeneric()` creates a new generic or converts an existing function into a generic. Here, we take the already existing `union()` function (see notebook 2 on set operations):

```
setGeneric("union")
```

```
## [1] "union"
```

`setMethod()` creates new methods, taking the name of the generic, the classes the methods should be associated with, and a function that implements the method. We make `union()` work with data frames (and not just vectors) using the following code:

```
setMethod("union",
  c(x = "data.frame", y = "data.frame"),
  function(x, y) {
    unique(rbind(x, y))
  }
)
```

If we do not utilize an already existing function, but create a new generic from scratch, we need to use the S4 equivalent to `UseMethod()`, which is `standardGeneric()`:

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

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

S4 method dispatch

S4 method dispatch works like S3 method dispatch when dispatching on a single class with a single parent, but with special classes `ANY` and "missing".

RC

In contrast to S3 and S4, RC methods do not belong to functions, but to objects. In addition, RC objects are mutable which means that they are not copied on modify. These characteristics make RC objects more comparable to objects in other programming languages. RCs are a special type of S4 class. There are no R base packages that provide RCs.

RC essentials

We use `setRefClass()` to create a new RC class (similar to `setClass()`in S4). In principle, we only need to specify a **name** argument, but may also define class **fields** (the RC equivalent to the S4 slots). We provide a simple example where we create an RC object to model a bank account (a useful field of application for RCs that work well for objects that change over time), but do not discuss any further details here:

```
Account <- setRefClass("Account",
  fields = list(balance = "numeric"))
```

Note that the `$` operator is used to set and get field values (not `@`):

```
a <- Account$new(balance = 100)
a$balance
```

```
## [1] 100
```

```
a$balance <- 200
a$balance
```

```
## [1] 200
```

We can again use `otype()` and `isS4/is` (RC objects are S4 objects that inherit from "refClass") to recognize an RC object:

```
otype(a)
```

```
## [1] "RC"
```

```
isS4(a)
```

```
## [1] TRUE
```

```
is(a, "refClass")
```

```
## [1] TRUE
```

The copy-on-modify characteristic can be seen when running the following lines of code:

```
b <- a
b$balance
```

```
## [1] 200
```

```
a$balance <- 0
b$balance
```

```
## [1] 0
```

We need the `copy()` function that comes with RC objects to make a copy of an object:

```
c <- a$copy()
c$balance
```

```
## [1] 0
```

```
a$balance <- 100
c$balance
```

```
## [1] 0
```