Manuel J. A. Eugster

Programming R

Chapter 7: The S3 object system

Last modification on June 19, 2012

Draft of the R programming book I always wanted to read
http://mjaeugster.github.com/progr
Licensed under the CC BY-NC-SA

7 The S3 object system

Object-oriented programming using the S3 object system.

The main literature for this section is:

- *R Language Definition* by R Core Team (2012)

7.1 Object-oriented programming

Object-oriented programming is a programming paradigm using **objects** to represent a problem as software. Central to any object-oriented language are the concepts of **class** and of **methods**.

A class is a definition, i.e., the blueprint, from which the individual objects are created. A specific object is then an instance of a specific class. Classes contains **structural**, **behavioral**, and **relational** information.

TODO: add more details ...

R has three object-oriented systems:

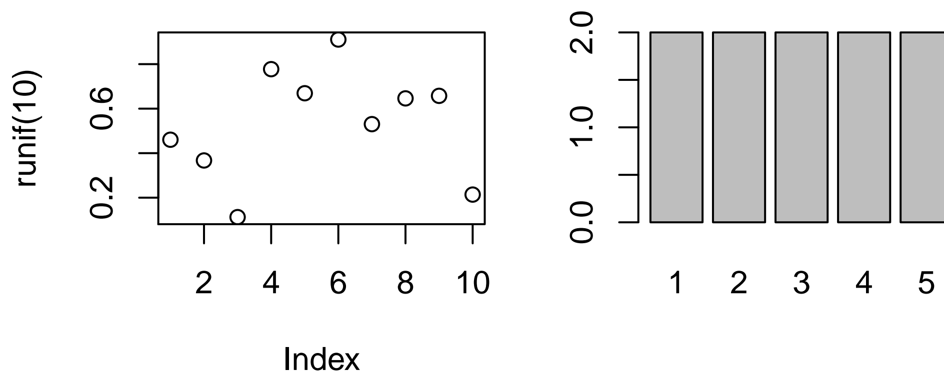
S3: "generic-function" style; see `?class`

S4: formal "generic-function" style with "multiple dispatch"; implemented in the package `methods`, see `?Classes`, `?Methods`.

ReferenceClasses: "message-passing" style; implemented in the package `methods`, see `?ReferenceClasses`.

This document explains the S3 object system (which I find really great!). One of the impacts of the S3 object system is that for two vectors, the same function results in two different plots:

```
> par(mfrow = c(1, 2), mar = c(4, 4, 1, 0))
> plot(runif(10))
> plot(gl(5, 2))
```



7.2 Object classes (Structure)

The S3 system does not provide formal class definitions (i.e., classes as blueprints objects). Instead, the class system is facilitated through the `class` attribute of an object; and this attribute simply is a character vector of class names.

An object's class is set using the `class()` function,

```
> a <- 11
> class(a) <- "PrimeNumber"
```

or, equivalently, by setting the `class` attribute,

```
> a <- 42
> attr(a, "class") <- "PrimeNumber"
>
> ## One-liner:
> a <- structure(a, class = "PrimeNumber")
```

In both cases the object now has an additional attribute which defines its class:

```
> a


[1] 42
attr(,"class")
[1] "PrimeNumber"


> class(a)


[1] "PrimeNumber"
```

```
> attributes(a)

$class
[1] "PrimeNumber"
```

Note that this approach allows to turn any object into an object of class `"PrimeNumber"`, whether or not it makes sense.

```
> structure("10", class = "PrimeNumber")

[1] "10"
attr(,"class")
[1] "PrimeNumber"

> structure("Hello World!", class = "PrimeNumber")

[1] "Hello World!"
attr(,"class")
[1] "PrimeNumber"
```

Every object has a class,

```
> class(pi)

[1] "numeric"
```

If there is no class attribute set,

```
> attributes(pi)

NULL
```

the object has an **implicit class**, `matrix`, `array`, `data.frame` or the result of `mode(x)` (cf. Chambers, 2008, Section 6.2).

The Monte-Carlo π example. We use the Monte-Carlo π estimation as example throughout this chapter.

If a circle of radius R is inscribed inside a square with side length $2 * R$, then the area of the circle will be $\pi * R^2$ and the area of the square will be $(2 * R)^2$. So the ratio of the area of the circle to the area of the square will be $\frac{\pi}{4}$. This means that, if you pick n points at random inside the square, approximately $n * \frac{\pi}{4}$ of those points should fall inside the circle. See, for example, Andersson (2010).

The following function is an implementation of this algorithm:

```
> mcpi <- function(n) {
+   stopifnot(is.numeric(n))
+   stopifnot(n >= 0)
+
+   x <- runif(n)
+   y <- runif(n)
+
+   inside <- ((x^2 + y^2) <= 1)
+
+   pi <- 4 * sum(inside) / n
+
+   ret <- list(pi = pi,
+               n = n,
+               sim = data.frame(x = x, y = y, inside = inside))
+
+   class(ret) <- "mcpi"
+
+   ret
+ }
```

It returns a list with three elements: (1) the estimated `pi`, (2) the number of drawn points `n`, (3) and the simulation data, a `data.frame` with the columns `x` (numeric), `y` (numeric), and `inside` (logical). In order to declare this list as a “special” object, i.e., an object of class `mcpi`, we define its class attribute before returning the list.

When executing the function,

```
> set.seed(1234)
> mcpi(5)


$pi
[1] 3.2
```

```
$n
[1] 5

$sim
       x        y inside
1 0.1137 0.640311   TRUE
2 0.6223 0.009496   TRUE
3 0.6093 0.232551   TRUE
4 0.6234 0.666084   TRUE
5 0.8609 0.514251  FALSE

attr(,"class")
[1] "mcpi"
```

we receive the list with the set class attribute. In the following we write methods for this `mcpi` class to make the handling of such objects straightforward.

7.3 Class methods (Behavior)

In order to define the behavior of objects we have to define **generic functions** and corresponding **methods** implementing the concrete behavior of specific classes.

Generic functions are functions with the statement `UseMethod("...")` in their body. Prominent generic functions already defined in the `base` package are `print()`, `summary()`, and `plot()`:

```
> print


function (x, ...)
UseMethod("print")
<bytecode: 0x02b85fbc>
<environment: namespace:base>
```

Methods implementing generic functions follow a simple naming convention, namely `generic.class`. The function `methods()` lists all available methods for an S3 generic function; e.g., all available `print` methods are:

```
> head(methods("print"))


[1] "print.acf"      "print.anova"    "print.aov"      "print.aovlist"
[5] "print.ar"       "print.Arima"
```

A generic function (in fact the statement `UseMethod()`) then uses the class of the first argument to figure out which method to call. If no such method is found, the `default`-method (`generic.default`) is used, if it exists, or an error results. This is called **method dispatching**.

For example,

```
> a <- 11
> class(a)

[1] "numeric"

> print(a)

[1] 11
```

Here, the generic function `print()` first looks for a method `print.numeric()`. As such a method is not available, it looks for the method `print.default()`, finds it, and passes the call to this function. The concrete executed function call therefore is `print.default(a)`.

Now, let us define the object `a` as an object of class `PrimeNumber`:

```
> class(a) <- "PrimeNumber"
> print(a)

[1] 11
attr(,"class")
[1] "PrimeNumber"
```

`print(a)` still executes `print.default(a)` as there is no `print`-method for the class `PrimeNumber` available. However, we easily can implement one:

```
> print.PrimeNumber <- function(x, ...) {
+   cat("Prime number:", x, "\n")
+ }
```

The implementation has to follow the signature of the generic function (e.g., `args(print)`). The method dispatch now finds the specialized `print`-method and dispatches to it:

```
> print(a)

Prime number: 11
```

The Monte-Carlo π example. In case of this example we can use this object-oriented system to provide a more user-friendly handling of the result.

```
> set.seed(1234)
> p <- mcpi(1000)
```

We can implement a `print`-method:

```
> print.mcpi <- function(x, ...) {
+   cat(x$pi, "(estimated by the Monte-Carlo method)\n")
+ }
>
> p

3.164 (estimated by the Monte-Carlo method)
```

And a `summary`-method for more details:

```
> summary.mcpi <- function(object, ...) {
+   hits <- sum(object$sim$inside)
+
+   print(object)
+   cat(sprintf("Estimated by %s hits from %s trials.\n",
+               hits, object$n))
+ }
>
> summary(p)

3.164 (estimated by the Monte-Carlo method)
Estimated by 791 hits from 1000 trials.
```

If we want to provide a method to access the estimated π value, we have to define a generic method:

```
> estpi <- function(x, ...) {
+   UseMethod("estpi", x)
+ }
```

And then the implementation of this generic method for the `mcpi` class:

```
> estpi.mcpi <- function(x, ...) {
+   x$pi
+ }
>
> estpi(p)

[1] 3.164
```

7.4 Inheritance (Relation)

Objects can have more than one class:

```
> class(a) <- c("PrimeNumber", "Number")
```

This is a simple way to define class/object hierarchies. The order of the class names defines the inheritance hierarchy; here `PrimeNumber` inherits from `Number` (is-a relation).

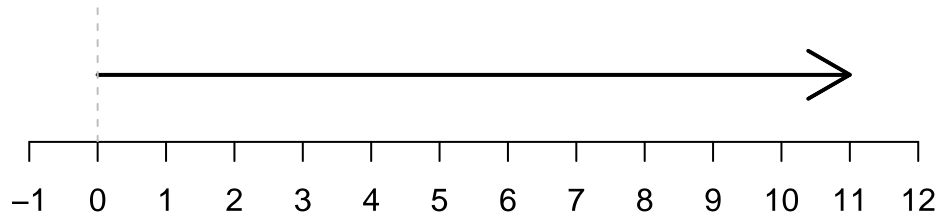
This means, if a method for the class `Number` is defined, and no method for the class `PrimeNumber`, the `PrimeNumber` object inherits the method from its superclass.

For example, the function

```
> plot.Number <- function(x, y, ...) {
+   x0 <- ifelse(x > 0, -1, +1)
+   x1 <- ceiling(x) + 1
+
+   plot(1, type = "n", xlim = c(x0, x1), ylim = c(-1, 1),
+        axes = FALSE, xlab = "Number line")
+   arrows(0, 0, x, 0, lwd = 2)
+   axis(1, at = seq(x0, x1))
+   abline(v = 0, col = "gray", lty = 2)
+ }
```

plots a number line in order to visualize a number. If we want to plot the object `a` which is of class `PrimeNumber` and inherits from class `Number`, no `plot`-method for `PrimeNumber` is found, but for `Number`:

```
> par(mar = c(4, 0, 0, 0))
> plot(a)
```



In detail, the method dispatch with multiple classes is as follows: When a generic function `generic` is applied to an object with class attribute `c("first", "second")`, the system searches for a function called `generic.first` and, if it finds it, applies it to the object. If no such function is found, a function called `generic.second` is tried. If no class name produces a suitable function, the function `generic.default` is used (if it exists). If there is no class attribute, the implicit class is tried, then the default method.

As a good practice, I suggest to always add an object's "original" classes to the new classes:

```
> a <- 11
> class(a) <- c("PrimeNumber", "Number", class(a))
> class(a)

[1] "PrimeNumber" "Number"      "numeric"
```

The Monte-Carlo π example. Let us add another algorithm to estimate π, namely the Leibniz formula for π. It states that

$$\sum_{i=0} \infty \frac{(-1)^i}{2i+1} = \frac{\pi}{4}.$$

So we can use this infinite series to approximate π by computing the first n parts. The following function implements this algorithm:

```
> leibnizpi <- function(n) {
+   stopifnot(is.numeric(n))
+   stopifnot(n >= 0)
+
+   approx <- 1
```

```
+   for ( k in seq(length = n) ) {
+     approx <- approx + (-1)^k / (2*k + 1)
+   }
+
+   pi <- 4 * approx
+
+   ret <- list(pi = pi, n = n)
+   class(ret) <- c("leibnizpi", "piest", class(ret))
+
+   ret
+ }
```

We introduce the class `piest` as a superclass for all algorithms estimating π. In further consequence, we have to adapt the `mcpi()` function to return an object of class `piest` as well. Then we can, for example, implement specific `print()` and `summary()` methods for this algorithm and a general `piest()` methods for all kinds of algorithms estimating π.

Bibliography

Eve Andersson. Calculation of pi using the monte carlo method. Website, 2010. Available online at http://www.eveandersson.com/pi/monte-carlo-circle; visited on July 8th 2010.

John Chambers. *Software for Data Analysis: Programming with R*. Springer, 2008. ISBN 9780387759357.

R Core Team. *The R language definition*, 2012. URL http://cran.r-project.org/doc/manuals/R-lang.html.