adv-r.had.co.nz

S3 · Advanced R.

R has three object oriented (OO) systems: [[S3]], [[S4]] and [[R5]]. This page describes S3.

Central to any object-oriented system are the concepts of class and method. A **class** defines a type of object, describing what properties it possesses, how it behaves, and how it relates to other types of objects. Every object must be an instance of some class. A **method** is a function associated with a particular type of object.

S3 implements a style of object oriented programming called generic-function OO. This is different to most programming languages, like Java, C++ and C#, which implement message-passing OO. In message-passing style, messages (methods) are sent to objects and the object determines which function to call. Typically this object has a special appearance in the method call, usually appearing before the name of the method/message: e.g. `canvas.drawRect("blue")`. S3 is different. While computations are still carried out via methods, a special type of function called a **generic function** decides which method to call. Methods are defined in the same way as a normal function, but are called in a different way, as we'll see shortly.

The primary use of OO programming in R is for print, summary and plot methods. These methods allow us to have one generic function, e.g. `print()`, that displays the object differently depending on its type: printing a linear model is very different to printing a data frame.

Object class

The class of an object is determined by its `class` attribute, a character vector of class names. The following example shows how to create an object of class `foo`:

```
x <- 1
attr(x, "class") <- "foo"
x

# Or in one line
x <- structure(1, class = "foo")
x
```

Class is stored as an attribute, but it’s better to modify it using the `class()` function, since this communicates your intent more clearly:

```
class(x) <- "foo"
class(x)
# [1] "foo"
```

You can use this approach to turn any object into an object of class “foo”, whether it makes sense or not.

Objects are not limited to a single class, and can have many classes:

```
class(x) <- c("A", "B")
class(x) <- LETTERS
```

As discussed in the next section, R looks for methods in the order in which they appear in the class vector. So in this example, it would be like class A inherits from class B - if a method isn't defined for A, it will fall back to B. However, if you switched the order of the classes, the opposite would be true! This is because S3 doesn't define any formal relationship between classes, or even any definition of what an individual class is. If you're coming from a strict environment like Java, this will seem pretty frightening (and it is!) but it does give your users a tremendous amount of freedom. While it's very difficult to stop someone from doing something you don't want them to do, your users will never be held back because there is something you haven't implemented yet.

Generic functions and method dispatch

Method dispatch starts with a generic function that decides which specific method to dispatch to. Generic functions all have the same form: a call to `UseMethod` that specifies the generic name and the object to dispatch on. This means that generic functions are usually very simple, like `mean`:

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

Methods are ordinary functions that use a special naming convention: `generic.class`:

```
mean.numeric <- function(x, ...) sum(x) /
length(x)
mean.data.frame <- function(x, ...) sapply(x,
mean, ...)
mean.matrix <- function(x, ...) apply(x, 2, mean)
```

(These are somewhat simplified versions of the real code).

As you might guess from this example, `UseMethod` uses the class of x to figure out which method to call. If `x` had more than one class, e.g. `c("foo","bar")`, `UseMethod` would look for `mean.foo` and if not found, it would then look for `mean.bar`. As a final fallback, `UseMethod` will look for a default method, `mean.default`, and if that doesn't exist it will raise an error. The same approach applies regardless of how many classes an object has:

```
x <- structure(1, class = letters)
bar <- function(x) UseMethod("bar", x)
bar.z <- function(x) "z"
bar(x)
# [1] "z"
```

Once `UseMethod` has found the correct method, it's invoked in a special way. Rather than creating a new evaluation environment, it uses the environment of the current function call (the call to the generic), so any assignments or evaluations that were made before the call to UseMethod will be accessible to the method. The arguments that were used in the call to the generic are passed on to the method in the same order they were received.

Because methods are normal R functions, you can call them directly. However, you shouldn't do this because you lose the benefits of having a generic function:

```
bar.x <- function(x) "x"
# You can call methods directly, but you
shouldn't!
bar.x(x)
# [1] "x"
bar.z(x)
# [1] "z"
```

Methods

To find out which classes a generic function has methods for, you can use the `methods` function. Remember, in R, that methods are associated with functions (not objects), so you pass in the name of the function, rather than the class, as you might expect:

```
methods("bar")
# [1] bar.x bar.z
methods("t")
# [1] t.data.frame t.default    t.ts*
# Non-visible functions are asterisked
```

Non-visible functions are functions that haven't been exported by a package, so you'll need to use the `getAnywhere` function to access them if you want to see the source.

Internal generics

Some internal C functions are also generic, which means that the method dispatch is not performed by R function, but is instead performed by special C functions. It's important to know which functions are internally generic, so you can write methods for them, and so you're aware of the slight differences in method dispatch. It's not easy to tell if a function is internally generic, because it just looks like a typical call to a C:

```
length <- function (x)  .Primitive("length")
cbind <- function (..., deparse.level = 1)
  .Internal(cbind(deparse.level, ...))
```

As well as `length` and `cbind`, internal generic functions include `dim`, `c`, `as.character`, `names` and `rep`. A complete list can be found in the global variable `.S3PrimitiveGenerics`, and more details are given in `?InternalMethods`.

Internal generic have a slightly different dispatch mechanism to other generic functions: before trying the default method, they will also try dispatching on the **mode** of an object, i.e. `mode(x)`. The following example shows the difference:

```
x <- structure(as.list(1:10), class = "myclass")
length(x)
# [1] 10

mylength <- function(x) UseMethod("mylength", x)
mylength.list <- function(x) length(x)
mylength(x)
# Error in UseMethod("mylength", x) :
```

```
#  no applicable method for 'mylength' applied to
an object of class
#  "myclass"
```

Inheritance

The `NextMethod` function provides a simple inheritance mechanism, using the fact that the class of an S3 object is a vector. This is very different behaviour to most other languages because it means that it's possible to have different inheritance hierarchies for different objects:

```
baz <- function(x) UseMethod("baz", x)
baz.A <- function(x) "A"
baz.B <- function(x) "B"

ab <- structure(1, class = c("A", "B"))
ba <- structure(1, class = c("B", "A"))
baz(ab)
baz(ba)
```

`NextMethod()` works like `UseMethod` but instead of dispatching on the first element of the class vector, it will dispatch based on the second (or subsequent) element:

```
baz.C <- function(x) c("C", NextMethod())
ca <- structure(1, class = c("C", "A"))
cb <- structure(1, class = c("C", "B"))
baz(ca)
baz(cb)
```

The exact details are a little tricky: `NextMethod` doesn’t actually work with the class attribute of the object, it uses a global variable (`.Class`) to keep track of which class to call next. This means that manually changing the class of the object will have no impact on the inheritance:

```
# Turn object into class A - doesn't work!
baz.D <- function(x) {
  class(x) <- "A"
  NextMethod()
}
da <- structure(1, class = c("D", "A"))
db <- structure(1, class = c("D", "B"))
baz(da)
baz(db)
```

Methods invoked as a result of a call to `NextMethod` behave as if they had been invoked from the previous method. The arguments to the inherited method are in the same order and have the same names as the call to the current method, and are therefore are the same as the call to the generic. However, the expressions for the arguments are the names of the corresponding formal arguments of the current method. Thus the arguments will have values that correspond to their value at the time NextMethod was invoked. Unevaluated arguments remain unevaluated. Missing arguments remain missing.

If `NextMethod` is called in a situation where there is no second class it will return an error. A selection of these errors are shown below so that you know what to look for.

```
c <- structure(1, class = "C")
baz(c)
# Error in UseMethod("baz", x) :
#   no applicable method for 'baz' applied to an
object of class "C"
baz.c(c)
# Error in NextMethod() : generic function not
specified
baz.c(1)
# Error in NextMethod() : object not specified
```

(Contents adapted from the R language definition. This document is licensed with the GPL-2 license.)

Double dispatch

How does + work.

Object styles

Two basic ways to create an object in S3: with a list or with attributes. With attributes is best if your object behaves like a vector, list if you're starting from scratch.

Best practices

- Create a construction method that checks the types of the input, and returns a list with the correct class label. `XXX <- function(...) {}`

- Write a function to check if an object is of your class: `is.XXX`

```
<-   function(x) inherits(x, "XXX")
```

- When implementing a vector class, you should implement these methods: `length`, `[`, `[<-`, `[[`, `[[<-`, `c`. (If `[` is implemented `rev`, `head`, and `tail` should all work).

- When implementing anything mathematical, implement `Ops`, `Math` and `Summary`.

- When implementing a matrix/array class, you should implement these methods: `dim` (gets you nrow and ncol), `t`, `dimnames` (gets you rownames and colnames), `dimnames<-` (gets you colnames<-, rownames<-), `cbind`, `rbind`.

- If you're implementing more complicated `print()` methods, it's a better idea to implement `format()` methods that return a string, and then implement `print.class <- function(x, ...) cat(format(x, ...), "\n"`. This makes for methods that are much easier to compose, because the side-effects are isolated to a single place.