M. Tiefelsdorf. *Modelling spatial processes*, volume 87 of *Lecture notes in earth sciences*. Springer, Berlin, 2000. 14

M. Tiefelsdorf and B. Boots. The exact distribution of Moran's I. *Environment and Planning A*, 27:985–999, 1995. 15

M. Tiefelsdorf, D. A. Griffith, and B. Boots. A variance-stabilizing coding scheme for spatial link matrices. *Environment and Planning A*, 31:165–180, 1999. 14

W. N. Venables and B. D. Ripley. *Modern applied statistics with S-PLUS*. Springer, New York, 1999. 14

*Roger Bivand*
*Economic Geography Section, Department of Economics, Norwegian School of Economics and Business Administration, Bergen, Norway*
Roger.Bivand@nhh.no

# Object-Oriented Programming in R

*by John M. Chambers & Duncan Temple Lang*

Although the term *object-oriented programming* (OOP) is sometimes loosely applied to the use of methods in the S language, for the computing community it usually means something quite different, the style of programming associated with Java, C++, and similar languages. OOP in that sense uses a different basic computing model from that in R, specifically supporting mutable objects or references. Special applications in R can benefit from it, in particular for inter-system interfaces to OOP-based languages and event handling. The **OOP** module in the Omegahat software implements the OOP model for computing in R.

## S language philosophy and style

When you write software in R, the computations are a mixture of calls to functions and assignments. Although programmers aren't usually consciously thinking about the underlying philosophy or style, there is one, and it affects how we use the language.

One important part of the S language philosophy is that functions ordinarily don't have side effects on objects. A function does some computation, perhaps displays some results, and returns a value. Nothing in the environment from which the function was called will have been changed behind the scenes.

This contrasts with languages which have the notion of a pointer or *reference* to an object. Passing a reference to an object as an argument to a function or routine in the language allows the called function to alter the object referred to, in essentially arbitrary ways. When the function call is complete, any changes to that object persist and are visible to the caller.

In general, S functions don't deal with references, but with objects, and function calls return objects, rather than modifying them. However, the language does include assignment operations as an explicit means of creating and modifying objects in the local frame. Reading the S language source, one can immediately see where any changes in an object can take place: only in the assignment operations for that specific object.[1]

Occasionally, users ask for the addition of references to R. Providing unrestricted references would radically break the style of the language. The "raw pointer" style of programming used in C, for example, would be a bad temptation and could cause chaos for R users, in our opinion.

A more interesting and potentially useful alternative model, however, comes from the languages that support OOP in the usual sense of the term. In these languages, the model for programming is frequently centered around the definition of a class of objects, and of methods defined for that class. The model does support object references, and the methods can alter an object remotely. In this sense, the model is still sharply different from ordinary R programming, and we do not propose it as a replacement.

However, there are a number of applications that can benefit from using the OOP model. One class of examples is inter-system interfaces to languages that use the OOP model, such as Java, Python, and Perl. Being able to mimic in R the class/method structure of OOP software allows us to create a better and more natural interface to that software. R objects built in the OOP style can be used as regular objects in those languages, and any changes made to their state persist. The R code can work directly in terms of the methods in the foreign language, and much of the interface software can be created automatically, using the ability to get back the metadata defining classes (what's called *reflectance* in Java).

Mutable objects (i.e., object references) are also particularly useful when dealing with asynchronous

[1] Assuming that the function doesn't cheat. Almost anything is possible in the S language, in that the evaluator itself is available in the language. For special needs, such as creating programming tools, cheating this way is admirable; otherwise, it is unwise and strongly deprecated.

events. For example, when a user clicks on a help button in a graphical user interface (GUI), we might first check to see if we have previously created the help window and if not, create the window and store a reference to it for use in the future. Here, updating the state of an object associated with the help action is convenient and natural. Similarly, cumulating data from a connection or stream when becomes available can be done easily by updating the state of an OOP object.

## The OOP model

In the OOP languages of interest here, functions are no longer the central programming tool. The basic unit of software is the definition of a class of objects. The class definition can include the data structure (the *slots* or *fields*) of the class, and the *methods* that can be invoked on objects from the class.

Methods in this model play somewhat the role of functions in R. But, in contrast to methods in R, these methods are associated with the class and the objects or particular realizations of the class. You invoke methods *on* an object. To illustrate, let's use an example in R. A simple application of OOP-style computing that we will discuss below is to create R objects that represent FTP (File Transfer Protocol) connections to remote sites.

One of the things you need to do with an FTP connection is to login. In the OOP model, `login` is a method defined for this class of objects. One invokes this method on an object. So, if the S object `franz` is an instance from an appropriate FTP class, the computation might look like:

```
franz$login("anonymous", "jmc@lucent.com")
```

In words, this says: for the object `franz` find the appropriate definition of the `login` method, and call it with the two strings as additional arguments. The exact notation depends on the language. We're using the familiar `$` operator, which in fact turns out to be convenient for implementing OOP programming in R. Java, Python, Perl, and other languages each have slightly different notation, but the essential meaning carries over.

Invoking methods rather than calling functions is the main difference in appearance. Object references and the ability to change an object through a method are the main differences in what actually happens. Where an application naturally suits such references, the OOP model often fits well. Object references and OOP suit the example of an FTP connection.

FTP is a simple but effective way to connect to a remote site on the Web and transfer data back and forth. Creating a connection to a particular site from a session in R, say, gives us a "thing"—an object, let's say. Unlike more usual R objects such as vectors of numbers, an FTP connection is very much a single thing, referring to that actual connection to the remote site. Computations may change the state of that object (e.g., whether we have successfully logged in to the site, or where we are currently in the file system). When they do, that changed state needs to be visible to all copies of the object: whatever R function call we're in, the FTP object for this connection *refers to* the same connection object.

In contrast, if one R function passes a vector of numbers to another R function, and that function rearranges its copy of the numbers, it's *not* the usual model that both copies change! We write R software all the time in the confidence that we can pass arguments to other functions without worrying about hidden side effects.

Two different computational models, each useful in the right context.

## OOP in R

The ability to have object references, in effect, in R can be implemented fairly directly, through a programming "trick". The closure concept in R allows functions to be created that can read and assign to variables in their parent environment. These variables are then in effect references, which can be altered by the functions, acting like OOP methods. Using closures, we might implement the FTP class in the following manner:

```
FTP <- function(host) {
  con <- NULL

  login <- function(id, passwd) {
    if(!is.null(con)) {
      stop("already logged in")
    }
    con <<- .Call("FTPLogin", machine,
                  id, passwd)
  }

  return(list(login=login))
}
```

We can use this

```
franz <- FTP("franz.stat.wisc.edu")
franz$login("anonymous", "jmc@lucent.com")
```

The first line creates a new instance of the FTP class with its own version of the `machine` and `con` variables. The call to `login()` updates the object's `con` value and subsequent calls can see this new value. More information and examples of closures are given in Gentleman and Ihaka (2000).

This approach is simple and fairly efficient, and can be quite useful. However, we are proposing here a somewhat more formal mechanism. Being more formal is helpful, we think, partly because the mapping to analogous OOP systems in other languages is then clearer. Formal definitions in software also have the advantage that they can be queried to let

software help write other software. We use such "reflectance" in other languages when building interfaces from R, and being formal ourselves brings similar advantages. Interfaces *from* other languages can query OOP class definitions in R. For example, we can automatically define Java or Python classes that mimic or even extend R classes. Programming with formal OOP classes in R should be easier also, since the formal approach provides tools for defining classes and methods similar to those that have worked well in other languages, while at the same time being simple to use in R. Finally, the formal OOP approach makes it more feasible to have an OOP formalism that is compatible between R and S-Plus, making software using the approach available to a wider audience.

## Defining classes

OOP programming begins by defining a class; specifically by creating a *class object* with a call to the `setOOPClass` function:

```
> setOOPClass("FTP")
```

The call to `setOOPClass` creates an OOP class definition object, with `"FTP"` as the class name, and also assigns the object with the same name. Class objects contain definitions for the methods available in the class. Objects from the class will usually contain data, stored in specified *fields* in the class. In our model, these fields are not accessed directly; access is encapsulated into methods to get and set fields. Classes can inherit from other OOP classes, and the class can itself have methods and fields. The class object, `FTP`, is an OOP object itself, so we can use OOP methods to set information in the class object.

For our FTP example, the class contains two fields to hold the name of the machine and the connection object:

```
> FTP$setFields(machine = "character",
                con = "connection")
```

This has the side effect of creating methods for setting and getting the values of these fields. To use the class, we create a constructor function which is responsible for storing the name of the host machine.

```
FTP$defineClassMethod(
  "new", function(machine) {
    x <- super(new())
    x$setMachine(machine)
    x
  }
)
```

Next we define the `login()` method for objects of this class.

```
FTP$defineMethod(
  "login", function(id, passwd) {
    setConnection(
      .Call("FTPLogin",
            getMachine(), id, passwd))
  }
)
```

The OOP methods `defineMethod` and `defineClassMethod` modify the object `FTP`.

Besides defining classes directly, R programmers can create interfaces to class definitions in other languages. If our `FTP` class used the interface to Perl, it might be created directly from a known class in Perl:

```
> FTP <- PerlClass("FTP", package="Net")
```

Methods for such classes can be defined automatically, using reflectance information. The extent to which this happens varies with the other language—Java provides a lot of information, Perl less.

Once the class is defined, objects can be created from it.

```
> franz <- FTP$new("franz.stat.wisc.edu")
```

Objects from an OOP class can be assigned and passed to functions just like any objects in R. But they are fundamentally different, in that they contain an object reference. If `franz` is passed to an R function, and that function calls an OOP method that changes something in its argument, you can expect to see the effect of the change in the object `franz` as well.

## Further information

The software and documentation for the OOP package for R is available from the Omegahat Web site at http://www.omegahat.org/OOP/.

## Bibliography

Robert Gentleman and Ross Ihaka. Lexical scope and statistical computing. *Journal of Computational and Graphical Statistics*, 9(3):491–508, September 2000. 18

*John Chambers*
*Bell Labs, Murray Hill, New Jersey, U.S.A*
jmc@research.bell-labs.com

*Duncan Temple Lang*
*Bell Labs, Murray Hill, New Jersey, U.S.A*
duncan@research.bell-labs.com