How S4 Methods Work

John Chambers

August 29, 2006

1 Methods and Classes; Functions and Objects

Methods and classes in the S language are essentially programming concepts to enable good organization of function implementations and of general objects, respectively. Programming in R starts out usually as writing functions, at least once we get past the strict cut-and-paste stage. The functions are the actions of the language; calls to them express what the user wants to happen. The arguments to the functions and the values returned by function calls are the objects. These objects represent everything we deal with. Actions create new objects (such as summaries and models) or present the information in the objects (by plots, printed summaries, or interfaces to other software). R is a functional, object-based system where users program to extend the capacity of the system in terms of new functionality and new kinds of objects.

Once programming in R reaches a moderately ambitious level, the total complexity of the added functionality can start to obscure what the software is doing. And the information in the objects may represent some important structure, to support new ideas and the information needed for computations.

The purpose of the method- and class-related software, usually referred to as S4 methods and classes, is to structure such programming into individual pieces. The pieces have two important properties: Their range of application is limited, so that they can make a clear statement about a strictly limited part of the program design; and they come with self-description, in the sense of metadata that allows the software itself to examine the methods and classes.

This document discusses the information describing methods and classes in R and how that information is used during an R session. The actual computations center on function calls, since nearly everything that happens in R occurs through a function call. Although the title talks about methods, and for programmers it's the construction of methods (and classes) that occupies most attention, the way to understand how things work is to see *functions* and classes as the prime organizing principles. Methods are the connectors that join the two together, and the essential understanding of "how it works" comes from the rules by which the R evaluator selects a method for a particular function call.

2 Functions, Classes, and Object-oriented Programming

Since methods and classes were introduced into S in the early 1990s, terms like "object-oriented programming" have turned up frequently in discussions. The term has many connotations and may help to orient readers, but it can as easily confuse them, so we need to clarify early on a distinction.

Languages to which the object-oriented programming (OOP) term is typically applied are mostly what might better be called *class*-oriented programming, well-known examples being C++ or Java. In these languages the essential programming unit is the class definition. Objects are generated as instances of a class and computations on the objects consist of *invoking methods on* that object. Depending on how strict the language is, all or most of the computations must be expressed in this form. Method invocation is an operator, operating on an instance of a class. Software organization is essentially simple and hierarchical, in the sense that all methods are defined as part of a particular class.

That's not how S works; as mentioned, the first and most important programming unit is the function. From the user's perspective, it's all done by calling functions (even if some of the functions are hidden in the form of operators). Methods and classes provide not class-oriented programming but function- and class-oriented programming. It's a richer view, but also a more complicated one.

The key concept to keep in mind is that function-and-class programming has that two-way, hyphenated organization: by functions and by classes. A method no longer "belongs" to a class. Methods belong primarily to the generic function for which they are defined. To understand how the computations work, the first organizing principle needed is to think of the generic function as *including* all the currently active methods for that function. This does in fact happen during an R session: A generic function will collect or *cache* all the methods for that function belonging to all the R packages that have been loaded in the session. When the function is called, the R evaluator then *selects* a method from those available, by examining how well different methods match the actual arguments in the call.

But precisely in this selection process we have to face the greater complexity of the function-class paradigm, compared to the simpler class paradigm. Methods are stored within the generic function according to matched pairing of one or more classes each with one of the formal arguments of the function. It's the closeness of the corresponding actual arguments to the paired class that determines how well the method matches the call. And that closeness in turn is determined by the definition of the class. As far as program design goes, the definition of suitable classes often motivates the design of methods.

From the users' view, though, the generic function has (or at least should have) a natural definition in terms of what it is intended to do: `plot()` displays graphics to represent an object or the relation between two objects; arithmetic operators such as `"+"` carry out the corresponding intuitive numerical computations or extensions of those. Methods should map those intuitive notions naturally and reliably into the concepts represented by the class definitions. Doing that well is the important part of dealing with methods and classes. The present discussion explains how a particular set of class and method definitions will be used in computations.

We want to end up with a description of how methods are selected, but getting there will go more smoothly if we take two detours to say how first classes and then methods are organized. We also need to bring in a third organizing principle explicitly, the R *package*. The next three sections take each of these in turn.

3 Classes and Inheritance

The definition of a class is obtained from a metadata object in some R package. When that definition becomes available, the definition is then cached for the current R session. It's this cached version that concerns us, because it is used to drive method selection and because it can differ slightly from the metadata object, as we will see.

The class definition contains a definition the slots in objects from the class and other information of various kinds, but the most important information for the present discussion defines what other classes this class extends; that is, the inheritance or to use the most common term, the *superclasses* of this class. In R, the names of the superclasses can be seen as the value of `extends(thisClass)`. The key is that, by definition, an object from any class can be used in a computation designed for any of the superclasses of that class. Therefore, it's precisely the superclasses of the object's classes that define candidate methods for a particular argument in a particular function call.

To be precise, if the relevant object has class `thisClass`, then the candidate classes are:

```
c(thisClass, extends(thisClass), "ANY")
```

where `"ANY"` is the universal superclass for all objects. A method specifying any one of these classes in its signature for the corresponding argument to the generic function claims that it will work if given an object from `thisClass`, possibly after the object has been coerced to the superclass.

Superclasses can be specified in three ways:

1. Directly in the `contains=` argument to `setClass()`, with the result that all the data (that is, all the slots) of the superclass will be included in this class as well.
2. Through a separate `setIs(thisClass, superClass)` specification, which may require providing methods to coerce objects from `thisClass` to `superClass`.
3. Through the creation of a class union of which `thisClass` is a member.

The class union case may cause the cached definition of a class to differ from the original metadata. For example:

```
setClassUnion("Numbers", c("numeric", "integer", "complex"))
```

The assertion is that "Numbers" is a superclass of each of the three specified member classes. So if a method has been defined with "Numbers" in its signature, an actual argument of class "numeric" can be supplied. Therefore, "Numbers" must be added to the superclasses of the cached definition of class "numeric", although it was not in the metadata (and even though in this case "numeric" was sealed, so ordinary computations can't change its definition).

Superclasses specified in any of the three ways form the *direct* superclasses of this class. The superclasses of the superclasses then define all the rest of the inheritance. The names of all these classes are the value of extends(thisClass). When inheritance is used to select a method, more than one of the superclasses may provide an alternative. In this case, the evaluator uses a concept of the distance between the class and a superclass to partially disambiguate the possibilities.

Just how this disambiguation should be done is a matter of opinion, and no solution is likely to be ideal in all examples. The current R implementation defines the distance as the number of generations of superclasses involved; that is, direct superclasses have distance 1, their direct superclasses have distance 2, and so on. In addition the distance to "ANY" is implicitly larger than the distance to any actual superclass, and a class has 0 distance to itself.

For example, in the Matrix package on CRAN, the class "ddenseMatrix" (for matrices of doubles stored in dense form), contains class "dMatrix" and class "denseMatrix" directly (in the contains= argument of the call to setClass() that creates the "ddenseMatrix" class. The class "dmatrix" contains only class "Matrix" directly (and so does class "denseMatrix"). therefore, "ddenseMatrix" contains 3 classes in total: "dmatrix" and "densematrix" with distance 1 and "Matrix" with distance 2.

4 Methods and Generic Functions

Conceptually, a generic function extends the idea of a function in R by allowing different methods to be selected corresponding to the classes of the objects supplied as arguments in a call to the function. The generic function should specify:

- The overall purpose and meaning; that is the *function* being performed, in the informal sense of that word.
- The formal arguments, along with a signature that specifies which arguments can be used to select methods.

For a standard generic function the body of the function only selects a method, by calling standardGeneric(), though the generic function can optionally do some computations of its own, before and/or after selecting the method. The signature of the generic function is an ordered subset of the formal arguments, by default all the eligible arguments in the order they appear in the function. The special argument ... is not eligible to be in a signature.

Arguments in the signature will be evaluated as soon as the generic function is called; therefore, any arguments that need to take advantage of lazy evaluation must not be in the signature. These

are typically arguments treated literally, often via the `substitute()` function. For example, if one wanted to turn `substitute()` itself into a generic, the first argument, `expr`, would not be in the signature since it must not be evaluated but rather treated as a literal.

When a method is to be selected from `StandardGeneric()` a class is associated with each of the arguments in the signature. If that argument is missing the class is `"missing"`; otherwise, it is the class of the object supplied as the argument. This list of classes is called the *target* signature for the call. All the methods for the generic function are likewise stored in a table, indexed by similar combinations of classes. This table contains both methods directly defined for the combination of classes and also those found via inheritance on previous calls to the function. The first step in selection is to check for an exact match in the table; if this succeeds, selection is complete and the evaluator goes on to use the selected method to evaluate the call to the function.

If no exact match occurs, a search for inherited methods takes place. Possible methods would correspond to a list of classes in which each element matched either the corresponding class in this call or else one of the superclasses of that class, including `"ANY"`. The search function creates all combinations of such classes for all the arguments actively involved in the signature, and matches all of them to a table of *directly* defined methods. Each of the methods in this table is stored according to the *defined* signature, the signature that appeared in the corresponding `setMethod()` call that created the method.

The result of the search will be zero, one or more possible methods. If zero, method selection has failed. If more than one method is selected, the search selects those that provide the closest match in terms of the distance between the actual class and the corresponding superclass in the definition of the method. Ties are possible, which the evaluator regards as ambiguous. It issues a warning and then selects the lexicographically first method (see the details below).

The selected method is added to the table of all methods, indexed under the target signature. So further calls with the same target signature will not require another search. Any changes in the underlying information driving the search could invalidate an inherited selection, however. Attaching another package involving the generic (say, by having new methods for it) is such a change. In this case the relevant generic functions are reset, in effect discarding all the inherited selections. Not every possible change is detected automatically. If, for example, you redefine one of the relevant classes during a session, you are in danger of still seeing invalidated selections: call `resetGeneric()` explicitly to be safe.

That's the basic picture. The following are details that may occasionally matter.

Comparing methods

The stored distance of each class in the target signature to each of its superclasses defines the distance of each candidate method, for that argument. Where more than one argument is involved in selection, there will be a distance for each argument. Method selection must judge some of the candidates as closer to the target and if ambiguities remain, must either generate an error or pick one of the tied possibilities. The important point for programming with R is that ambiguous

method selections usually indicate the need to be more specific about method definitions. Any strategy on the part of the system is likely to be wrong at least some of the time for a particular application. In particular, the best method to select for one generic function may not be best for another, when the same inheritance rule is used.

The current implementation (as of version 2.4.0 of R) works as follows. The individual distances, as explained in section 3, are the number of steps in the path of superclasses between the target and the defined class, with the distance to `"ANY"` larger than any other distance. If several arguments are involved, the distances for each argument are added and the total distance is used to compare candidate methods. In the case of ties, the first of the candidates is chosen. Because the candidate signatures are effectively an outer product of the individual superclass lists, and because of the way those lists are created, the effect is to select the first possible method in a lexicographic ordering.

For example, class `"ddenseMatrix"` has two direct superclasses, `"dMatrix"` and `"denseMatrix"`, appearing in that order in the class definition. So for an argument with target class `"ddenseMatrix"`, candidate methods for class `"dMatrix"` would come first in the list, even though both superclasses are at distance 1. When a second argument is involved, the combined list of classes is like a matrix with rows corresponding to the first argument and columns to the second, so the first choice or the first argument comes first.

This determines the effective ordering of methods, and it probably reflects as reasonable a decision as any. But again, the important point is that complex disambiguating rules are not the route to good software: Thinking through the intended method selection is much better.

Even the decision about which methods are ambiguous remains somewhat arbitrary. The current rule regards all indirect inheritance as farther away than direct inheritance. If the indirection is through the *same* superclass, no one could object. But the distinction is not totally convincing otherwise. For example, class `"dgeMatrix"` has `"generalMatrix"` as a direct superclass and `"dMatrix"` as a superclass at distance 2. Does this mean that the former is a closer match for methods? Not necessarily, because the two superclasses have totally different meanings: The first has to do with the form of the matrix, the second to do with the data type for the entries. Depending on the function, one might need to inherit via one or the other. Relying on a disambiguating rule would be a bad idea.

Group generic functions

Generic functions can have a group generic defined, meaning another function whose methods are supposed to work for this function as a special case. For example the function `"+"` has group generic function `Arith()`, which in turn has group generic function `Ops()`. In selecting a method for `"+"`, methods for `Arith()` are candidates for target signatures where there is no corresponding method for `"+"`, and methods for `Ops()` are candidates if neither of the other two functions has a method.

This is a second dimension of inheritance, but it is applied in a slightly different way, partly to conform to the above definition of group generic functions and partly to keep the complexity of group generic methods from leaking into the more common case where no group generic is defined.

When there is a group function, and an inherited method is needed, the first step is to look for a direct match to the target signature. For example, suppose the expression `x + 1` is being evaluated with `x` having class `"dgeMatrix"` from package `Matrix`. Then the target signature is `"dgeMatrix"` and `"numeric"`. There is no directly defined method for `"+"` for this combination, but there is for function `Arith()`. So the search for an inherited method would return immediately with the group generic method.

If there is no direct match for a group generic, the inheritance computation proceeds as before, except that all candidate methods will be matched for the successive group generic functions as well. For each signature, a group method will be a candidate if there is no corresponding meted for the actual function. So the final set of candidate methods includes group methods on the same footing, so long as they don't override directly defined methods. The distance calculations go ahead as before, but potentially on a larger set of candidates.

Primitive functions

In all discussions of methods, primitive functions unfortunately have to be considered separately. These are functions in the base package that are evaluated directly in `C`; there is no object of type `"function"` corresponding to them and they have no formal arguments. In principle, however, method selection works the same for these as for true functions. The only difference is that selection is initiated by special code inside the `R` evaluator, rather than through encountering a call to `standardGeneric()`.

The generic functions for primitives are not directly visible (so that nothing impedes the efficiency of calls to those functions). You can see the generic, however by calling `getGeneric()`:

```
> getGeneric("+")
standardGeneric for "+" defined from package "base"
  belonging to group(s): Arith

function (e1, e2)
standardGeneric("+", .Primitive("+"))
<environment: 0x2b82eb0>
Methods may be defined for arguments: e1, e2
```

The generic functions corresponding to primitives are kept in a list in the `methods` package, and used for method dispatch and other computations, without being assigned in the ordinary way. So long as computations access the generic function through `getGeneric()`, most computations should be unaffected.

One other special behavior required for primitive functions is the interpretation of the default method. The general rule requires all method objects to be of class `"MethodDefinition"`, or a subclass of that class. The exception is that the default method for primitive generic functions is the original primitive object:

```
> getMethod("+", "ANY")
```

```
.Primitive("+")
```

In fact, the evaluator does not call the primitive again for the default method, since doing so would create an infinite recursion. Instead, encountering an object of primitive type for a method signals that there is no method defined for this signature. The internal method search returns a special reference that tells the calling code in the evaluator to go on with the original built-in definition for the function. Programmers do need to be careful when working with selected method definitions to be prepared for a primitive object. It would be possible to use a special form of

Active signature

The signature slot of a generic function determines which arguments may be included in the signature of a method for that function. By default, and usually the signature consists of all the arguments, except for ..., if that is a formal argument, and in the order the arguments appear in the function definition. For example, the generic function for the subset operator is:

```
> getGeneric("[")
standardGeneric for "[" defined from package "base"

function (x, i, j, ..., drop)
standardGeneric("[", .Primitive("["))
<environment: 0x2a2207c>
Methods may be defined for arguments: x, i, j, drop
```

The argument `drop` to the generic is only relevant for arrays; in addition, the argument `j` is included to allow methods to depend on the class of the column index for two-way objects such as matrices or data frames.

In applications that do not involve such objects, it's likely that neither `j` or `drop` would appear in method signatures. Method selection takes this into account, but keeps the computed results of a call to `"["` from depending on such irrelevant questions. If all the methods currently cached for a generic involve only the first k of n arguments in the signature, only those k arguments are treated as active. This means that the indexing in the table only uses these arguments and that you only see these arguments when calling `showMethods()`. All the methods then implicitly correspond to class `"ANY"` for the remaining arguments, but this information is not explicitly recorded. Some efficiency advantages for the internal tables result . If all the methods for argument `j`, say corresponded implicitly to `"ANY"`, but argument `j` is included in the active signature, then each target signature with a distinct class for `j` has to be found by inheritance and separately cached. If `j` is not in the active signature, only distinct signatures for `x` and `i` need to be stored.

Note two points about the active signature, however, that are required for consistent semantics:

1. As soon as a method has to be cached that uses one of the currently inactive arguments, that argument and all preceding arguments have to be added to the active signature, and the

table of cached methods has to be relabeled accordingly. For example, the `Matrix` package currently has methods involving all four of the arguments in the signature for `"["`. If the currently active signature had involved fewer arguments, then attaching the `Matrix` package would result in expanding the active signature.

2. All the arguments in the full signature are evaluated as described above, not just the active ones. Otherwise, in special circumstances the behavior of the function could change for one method when another method was cached, definitely undesirable.

5 Packages

A final layer of richness, and inevitably of complications as well, comes from the R package mechanism.

A package is a collection of software, written in R and potentially in other languages, plus documentation and overall descriptions. For the present discussion, packages are important because they contain descriptions of functions, classes and methods. When a package joins the computations in an R session by being attached or loaded, the corresponding information is cached by the R system. Packages consist of a fairly amorphous collection of files; perhaps unfortunately, they don't currently have a definition in terms of objects. For our present purpose, they are identified primarily by a character-string name (`methods`, `lattice`, `mle`, etc.).

Class definitions and generic functions both have a `"package"` slot that identifies the package in which they were defined, as a character string. In the case of the generic function this may be the package from which the original function came. The command

```
setGeneric("ncol")
```

for example, will create a generic function from the function `ncol()` in the base package. The `"package"` slot in that generic will be `"base"`. This convention reflects the concept that the generic function retains the functionality of the original, and that methods will be cached in the single function. Implementation details sometimes deviate, at least in the current implementation, but this is the intended model.

While the generic function is intended to be associated with one particular package, methods for that function may be defined in many packages. That's the dual nature of function-and-class programming again; a package that defines new classes will naturally define methods that involve those classes, both for functions that it owns itself and for functions from other packages, particularly from the base package.

The caching mechanism is activated when a package is loaded, for generic functions in either that package or other packages. Method definitions in the new package are cached in the generic function. At present they override an existing method for the same signature, if there is one. In principle, the method selection mechanism could distinguish methods with the same signature from different

packages (most likely, simply by including the package information as part of the signature). There is no intrinsic difficulty, but some substantial changes in the current implementation would be needed.

The namespace mechanism for packages introduces an additional layer of complexity. If a package has a namespace, the implementer chooses whether or not to emphexport objects from the package; if a function object is not exported, it is available to calls from other functions in the package, but is not visible outside of the package. Methods and classes are also part of the namespace specification. If methods are specified for a generic that is exported from the namespace or that is part of another package and globally visible, then those methods will be cached in the visible generic function, behaving more or less normally.

If a package has a generic function that is local and not exported, the natural intention would seem to be that the methods are cached only in the function local to the namespace. That is true, so long as the package has been installed with "lazy loading" or some similar mechanism, which you should do for any non-trivial package in any case. The generic is then not cached in the global table of generic functions and in particular will not interfere with an identically named generic from another package. (If the source for your package is run when the package is loaded, however, every generic function will be cached. Do use lazy loading.)

The case of generic functions defined from functions in the base package is messier still, particularly in the distinction between primitive functions and other functions in the base package. More details on this later.