Classes and Methods in the S Language

John M. Chambers

This document introduces formally defined classes and methods for the S language. It is adapted from a small part of the related material in the book *Programming with Data*, [Cha98]. The adapted version describes compatible implementations of classes and methods for S-Plus and R.

1 Defining Classes and Methods

.

Methods in S define how a particular function should behave, based on the class of the arguments to the function. You can find yourself involved in programming new methods for several reasons, usually arising either from working with a new class of objects or with a new function:

1. You need to define a new class of objects to represent data that behaves a bit differently from existing classes of objects.

2. Your application requires a new function, to do something different from existing functions, and it makes sense for that function to behave differently for different classes of objects.

You may also just need to revise an existing method or differentiate classes that were treated together before. However it comes about, defining a new method is the workhorse of programming with objects in S.

Conversely, defining a new class is less common, but often the crucial step. Classes encapsulate how we think about the objects we deal with, what information the objects contain, what makes them valid. You will likely write many more methods than class definitions, particularly since each new class definition typically generates a number of new methods. But the usefulness of your project will depend on good design of the object classes, probably more than anything.

To begin, though, we will discuss the simpler project of designing methods. Let's take on a project to write a function that returns a "one-line" description of an object. We could just type the name of the object, of course, and S would show us that object. Also, the function `summary`, supplied with S, is designed to summarize the essential information in an object, usually in a page or so. Our project takes `summary` one step further, with the goal of a one-liner.

We will name the function `whatis` and give it just one argument, the object we're interested in. One definition might be:

```
> whatis <- function(object) data.class(object)
> whatis(1:10); whatis(state.x91); whatis(whatis)
[1] "integer"
[1] "matrix"
[1] "function"
```

Okay, but not much of a programming contribution. What else might we want to know about the object? Something about how big it is, perhaps. We can use the S function `length`. So we might try, as a second attempt:

```
> whatis <- function(object) paste("An object of class",
+    data.class(object), "and length", length(object))
```

The S function `paste` pastes strings from all its arguments into single strings. Let's try this definition on a few objects:

```
> whatis(x1)
[1] "An object of class numeric and length 14"
> whatis(xm)
[1] "An object of class matrix and length 42"
> whatis(whatis)
[1] "An object of class function and length 2"
```

Well, better but not great. The idea of length is fine for numeric objects, and generally for the `vector` objects (see the references [Cha98], [BCW88]). But for a matrix we would like to know the number of rows and columns, and for a function it may not be clear what we would like, but certainly the length has no obvious relevance at all. Let's go back to the simpler definition, `data.class(object)`, and think things over.

1.1 Defining Methods

Around now we realize that the generic purpose of the function (in this case, to produce an informative one-line summary) needs to be implemented by different methods for different kinds of objects. The class/method mechanism in S provides exactly this facility. We will define methods for those classes of objects where we can see a useful, simple summary. The existing function still plays a role, now as the *default* method, to be used when none of the explicit methods applies. For this purpose we will want to return to a simple definition, say:

```
> whatis <- function(object) paste("An object of class",
+    data.class(object))
```

That's a definition that will, at least, work for all objects and not give back information that might be confusing for some objects.

For a definition of `whatis` for ordinary vectors of numbers, character strings, or other kinds of data, the `length` is quite reasonable. This is where virtual classes are so helpful ([Cha98]). We don't need to implement a method for every actual vector class, just one method for all vectors.

The method is defined by a call to the function `setMethod`:

```
setMethod("whatis",
  "vector",
  function(object)
    paste(data.class(object), "vector of length", length(object))
)
```

We tell `setMethod` three things: what generic function is involved, what classes of arguments the method corresponds to, and what the definition of the method is. S uses the term *signature* for the second item: in general, it matches any of the arguments of the function to the name of a class. The definition of the method is a function; it must always have exactly the same arguments as the generic. This is the first method defined for `whatis`, so S just takes the ordinary function as defining the generic.

The `vector` method will do fine for those ordinary vectors, but for objects with more complicated classes, we can do more. Consider matrices, for example. We would like to know the number of rows and number of columns. What else should we include in a one-line summary? Well, matrices are examples of S *structures*: objects that take a vector and add some structure to it. So we might ask whether the relevant information about the underlying vector could be included. We decided before that the class and the length are useful descriptions of vectors, but in this case we don't need the length if we know the number of rows and columns. We can include the class of the vector, though, and this is useful since matrices can include any vector class as data. All that information can be wrapped up in the function:

```
whatIsMatrix <- function(object)
   paste(data.class(as(object, "vector")), "matrix with",
         nrow(object), "rows and", ncol(object), "columns")
```

In order to make this the matrix method for `whatis`, we call the function `setMethod` again.

```
setMethod("whatis", "matrix", whatIsMatrix)
```

We can call `showMethods` to see all the methods currently defined for a generic [1].

```
> showMethods("whatis")
     Database   object
[1,] "."        "ANY"
[2,] "."        "matrix"
[3,] "."        "vector"
```

Three methods are defined, all on the working database (which happened to appear as `"."` in the `search` list of databases). The method corresponding to class `ANY` is the one used if none of the other methods matches the class of `object`; in other words, the default method.

At this point, there are some observations worth noting.

- We did not define the default method. Because there was an existing definition for the function, S assumed that definition should continue to be used when no explicit method applied.

- The call to `setMethod` clearly had some side effect, since the new method definition persisted. For nearly all applications, you don't want to worry about exactly *what* side effect; to be safe, just use tools such as `setMethod`, `dumpMethod` and `source` to set, dump, and redefine the methods.

- We remarked on the usefulness of virtual classes, not only `vector`, but also `structure`. There are a number of virtual classes in S, and you can define more; see section [Cha98, Section 7.1] for a discussion.

- In the `matrix` method, we created a special function, `whatIsMatrix`. That is usually a good idea, so we can test the function before explicitly making it a method. There are two ways to use the function: making its current definition the method, as we did here, or writing the method

[1] **Note on R/S-PlusCompatibility**: The appearance of the output is shown for S-Plus, and will differ somewhat in R

as a call to the special function. Either way is fine, but in the first case redefining `whatIsMatrix` will *not* change the method, because it was the value, the *object*, that was passed to `setMethod`.

If you want to edit the definition of a particular method, the function `dumpMethod` will write it out to a file. In fact, `dumpMethod` works even if no method has been explicitly defined for this class. For this reason, it's the best general way to start editing a new method:

```
> dumpMethod("whatis", "numeric")
Method specification written to file "whatis.numeric.S"
```

I asked for the method for class `numeric`; there is no explicit method, but since class `numeric` extends class `vector`, that method was written out. The file name chosen combines the name of the generic and the signature of the method we're looking at. The file `"whatis.numeric.S"` contains:

```
setMethod("whatis", "numeric",
function(object)
paste(data.class(object), "vector of length", length(object))
)
```

The `setMethod` call, when evaluated, will define the new method, but currently just contains the method for class `"vector"`. After we edit the file,

```
source("whatis.numeric.S")
```

will define the method. As an exercise, you might try writing a method for numeric objects: perhaps in addition to length, the `min` and `max` might be interesting.

Let's look at a few calls to `whatis` with the methods defined so far.

```
> whatis(1:10)
[1] "integer vector of length 10"
> whatis(x1)
[1] "numeric vector of length 14"
> whatis(x1 > 0)
[1] "logical vector of length 14"
> whatis(letters)
[1] "character vector of length 26"
> whatis(xm)
[1] "numeric matrix with 14 rows and 3 columns"
> whatis(paste)
[1] "An object of class function"
```

The case of a function object still falls through to the default method, because a `function` object is not a vector. There is nothing particularly difficult in dealing with functions as objects, but you will need to find some tools to help. If you'd like to try writing a `whatis` method for function objects, see [Cha98, page 79] for some utilities that work in both S-Plus and R (with the `SMethods` package).

1.2 Defining a New Class

Designing a class is an extremely important step in programming with data, allowing you to mold S objects to conform to your application's needs. The key step is to decide what information your class should contain. What does the data mean and how are we likely to use it? There are often different ways to organize the same information; no single choice may be unequivocally right, but some time pondering the consequences of the choices will be well invested.

The mechanism for creating classes is fairly simple: you call one function, `setClass`, to define the new class, and then write some associated functions and methods to make the class useful. For many new classes, this includes some or all of the following:

1. software to create objects from the class, such as generating functions or methods to read objects from external files;

2. perhaps a method to validate an object from the class;

3. methods to show (print and/or plot), or to summarize the objects for users;

4. data manipulation methods, especially methods to extract and replace subsets;

5. methods for numeric calculations (arithmetic and other operators, math functions) that reflect the character of the objects.

We will sketch a few of those here, using a relatively simple, but practical, example. In [Cha98, Section 1.7], a more extended example is given.

Suppose we are tracking a variable of interest along some axis, perhaps by a measuring device that records a value y_1 at a position x_1, a value y_2 at x_2, and so on. The vector y is the fundamental data, but we need sometimes to remember the positions, x, as well. This example was developed by my colleague Scott Vander Wiel for an application with x being distance along a length of fiber optic cable, and y some measurement of the cable at that position. Clearly, though, the concept is a very general one.

How do we want to think of such data? Basically, we want to operate on the measurements, but always carry along the relevant positions and use them when this makes sense; for example, when plotting. What is the natural way to implement this concept? We could represent the data as a matrix, with two columns. This leaves the user, however, to remember when to work with one or the other column, or both. We could represent the data as a list with two components, but this has a similar problem.

S provides a class definition mechanism for such situations. We can decide what information the class needs, and then define methods for functions to make the class objects behave as users would expect. Users of the methods can for most purposes forget about how the class is implemented and just work with the concept of the data.

Classes can be defined in terms of named *slots* containing the relevant information. In this case, the choice of slots is pretty obvious: positions and measurements, or `x` and `y`. The S function `setClass` records the definition of a class. Its first two arguments are the name of the class and a representation for it—pairs of slot names and the class of the corresponding slot. The function `representation` constructs this argument. Let's call the new class `track`, reflecting the concept of tracking the measurement at various positions.

```
setClass("track", representation(x = "numeric", y = "numeric"))
```

S now knows the representation of the class and can do some elementary computations to create and print objects from the class. The operator `"@"` or the function `"slot"` can be used to extract or set the slots of an object from the class. The operator takes an object as its left operand and a quoted or unquoted string, naming the slot, as its right operand. The slot name here is *unevaluated*; that is, `t1@x` looks for a slot named `"x"`, rather than evaluating an object named `x`. If you do need to compute the name of the slot, that's how the function `slot` works; its first argument is the object and its second argument is the string specifying the slot, with both arguments evaluated in the ordinary S way. When working with slots, though, you more often will know the name explicitly, since it's part of the definition of the class.

To create a `track` object from positions `pos1` and responses `resp1`:

```
> tr1 <- new("track", x = pos1, y = resp1)
```

The function new returns a new object from any non-virtual class. Its first argument is the name of the class, all other arguments are optional and if they are provided, S tries to interpret them as supplying the data for the new object. In particular, as we did here, the call to new can supply data for the slots, using the slot names as argument names.

Since S knows the representation of the class, an object can be shown using the known names and classes of the slots. The default method for show will do this:

```
> tr1
An object of class "track"

Slot "x":
 [1] 156 182 211 212 218 220 246 247 251 252 254 258 261 263

Slot "y":
 [1] 348 325 333 345 325 334 334 332 347 335 340 337 323 327
```

It's also possible to convert objects into the new class by a call to as. For example, a named list, xy, with elements x and y could be made into a track object by as(xy, "track").

Most classes will come with *generating* functions to create objects more conveniently than by calling new. S imposes no restrictions on these functions, but for convenience they tend to have the same name as the class. Their arguments can be anything. For track objects, a simple generator would look much like what we did directly:

```
track <-
# an object representing measurements 'y', tracked at positions 'x'.
function(x, y)
{
  x <- as(x, "numeric")
  y <- as(y, "numeric")
  if(length(x) != length(y))
    stop("x, y should have equal length")
  new("track", x = x, y = y)
}
```

From a user's perspective, a major advantage of a generating function is that the call is independent of the details of the representation.

```
> tr1 <- track(pos1, resp1)
```

For some classes, we also want to be able to read the objects simply from text files, using the scan function (see [Cha98, Section 1.7] for examples). For now we'll assume that track objects are just generated from positions and measurements that are already available.

Once we can create objects from the class conveniently, we want to look at them. A method for the function show will be called when S automatically displays the result of a computation.[2] This function takes only one argument, object, the object to be displayed. The display can be either printed (the usual choice), or plotted. Two other functions, print and plot, can also have methods defined for these two cases. These functions have a more general form, but the method for a new class of objects is usually called with just one argument. The function summary is expected to produce a short description, ideally to fit on a single page regardless of the size of the object.

Methods for show should display the objects so that all the essential information is visible in a simple, intuitive way. Easier said than done, in many cases! For track objects, we want to associate corresponding values from the x and y slots. It should be clear to users which is which, but we also want to remind them that this is a special class, and not a list or a matrix. You might want to stop reading at this point and think about or sketch out some possibilities.

At any rate, here is one line of thinking. A fairly nice way to pair the values is to make up a matrix with two rows or columns for the x, y values. But a two-column matrix will waste all sorts of space printing, so it better be a two-row matrix. We can get that by using the S function rbind, which pastes its arguments together as rows of a matrix. Let's try it:

```
> xy <- rbind(tr1@x, tr1@y)
> xy
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
[1,]  156  182  211  212  218  220  246  247  251
[2,]  348  325  333  345  325  334  334  332  347

     [,10] [,11] [,12] [,13] [,14]
[1,]   252   254   258   261   263
[2,]   335   340   337   323   327
```

Not too bad, but we need to distinguish the rows by providing row names:

[2] **Note on R/S-PlusCompatibility**: At the time this is written, R does not recognize formal methods when automatically printing objects. You can get somewhat the same effect by defining objects with the name print.*Class* where *Class* is the class of the object. Meanwhile, imagine all the automatic printing in the examples replaced by, e.g., show(xy).

"x" and "y" should be good enough. The column names are not too nice either, because they make readers think that a `track` really is a matrix, and when we get on to thinking about other methods, that is not the way we want to think of `track` objects. We could put in empty column labels, but it might be better to be reminded of how many points there are in the object, so let's use `1`, `2`, ... The row and column names of the matrix are set by replacing the `dimnames` (see the online documentation `?dimnames`).

```
> dimnames(xy) <- list( c("x", "y"),
+   1:ncol(xy))
> xy
    1   2   3   4   5   6   7   8   9  10  11  12  13
x 156 182 211 212 218 220 246 247 251 252 254 258 261
y 348 325 333 345 325 334 334 332 347 335 340 337 323

   14
x 263
y 327
```

Good enough for now. We now package up the method as a function, and supply it to `setMethod`. The best way to do this, as usual, is to call `dumpMethod`:

```
> dumpMethod("show", "track")
Method specification written to file "show.track.S"
```

Whether or not a method has been defined explicitly, this will write out the current method, maybe the default method. Then we can edit the definition in the file to be what we want, in this case:

```
setMethod("show", "track",
  function(object) {
    xy <- rbind(object@x,
     object@y)
    dimnames(xy) <- list( c("x", "y"),
         1:ncol(xy))
    show(xy)
 })
```

Once we source in this file, the new method is defined.

```
> source("show.track.S")
> tr1
    1   2   3   4   5   6   7   8   9  10  11  12  13
x 156 182 211 212 218 220 246 247 251 252 254 258 261
```

```
y 348 325 333 345 325 334 334 332 347 335 340 337 323

   14
x 263
y 327
```

The style of this method is typical of many: we construct a new kind of object, and then apply the original generic function to this object. This allows us to reuse the existing method (in this case, for printing matrices), without having to worry about the details of that method. Use this technique liberally, it often provides the most elegant implementation of methods. Just be sure that when you recall the generic function you are getting a different method: otherwise an infinite loop will result. (S catches such loops fairly cleanly, but they still can be confusing to debug.)

For many classes, plotting the objects is as important as printing them. For `track` objects, this is certainly true. The first question is how to plot the objects by themselves: a method to interpret expressions such as `plot(tr1)`. A natural plot is just a scatter plot with the positions on the x axis and the measurements on the y axis. Obviously, this was partly what suggested the names of the slots in the first place. The second question is how to plot tracks against other objects: in this case we would usually just take the `y` measurements to define the object, and ignore the `x` values. [3]

The function `plot` has arguments `x` and `y`, plus other arguments for supplying various parameters.

```
> args(plot)
plot(x, y, ...)
```

For `plot`, we want to provide methods that depend on both the `x` and `y` arguments. The method is set the same way as before, but now we give as the second argument to `setMethod` a signature with two arguments specified. For example, if we want to plot a `track` object on its own, the `y` argument to `plot` is missing. This is a perfectly legitimate class to specify.

```
setMethod("plot",
   signature(x = "track", y = "missing"),
   function(x, y, ...)  whatever ...
)
```

[3] **Note on R/S-PlusCompatibility**: In R, the current definition of `plot` has only `x` as an argument, so the examples here that use both arguments can not be applied. You can either re-define plot to have two arguments or follow the style shown with a different function: see the online documentation for `setMethod` for some examples in this style.

There remains the definition of the method. Let's take a simple approach and just call `plot` again, with the two slots as arguments: `plot(x@x, x@y)`. To establish the corresponding method:

```
setMethod("plot",
   signature(x = "track", y = "missing"),
   function(x, y, ...) plot(x@x, x@y)
)
```

As it happens, the same result is obtained from converting the object to its "unclassed" form: `unclass(tr1)` is a named list with elements `x` and `y`, and a plot method for such objects produces the same scatter plot. Either way, the only major drawback is that the labels for the plot are not very nice. From looking at the documentation of `plot`, we see that arguments `xlab` and `ylab` supply labels. We can edit the method by calling `dumpMethod`, for example, and editing the resulting file to supply labels:

```
setMethod("plot",
   signature(x = "track", y = "missing"),
   function(x, y, ...)
      plot(unclass(x), xlab = "Position",
           ylab = "Value", ...)
)
```

Evaluating this task redefines the method with fairly reasonable labels.

So how about plotting `track` objects against other objects? Here we want to specify the `x` argument, but not the `y` at all, if the `track` object is to be on the x axis. We can just omit the `y` argument from the signature. The method is simple again (aside from labels):

```
setMethod("plot",
   signature(x = "track"),
   function(x, y, ...) plot(x@y, y, ...)
)
```

Similarly, if the `track` is on the y axis:

```
setMethod("plot",
   signature(y = "track"),
   function(x, y, ...) plot(x, y@y, ...)
)
```

(We have to be careful to keep the two uses of the names `x` and `y` clear here—for the slots in the object and for the arguments and axes in the plot.) With these three methods, we are now prepared to plot using track objects in a pretty reasonable style.

Not perfectly, however. Notice that the labels in the plots will not show the expression defining the track object (e.g., `tr1`). Also, there are other plotting functions besides `plot`, to do things like drawing lines, adding scatters of points, or displaying text at co-ordinates defined by the data. It would be nice to include all these as well in the methods for `track` objects. All this can be accomplished and quite elegantly, but it is a little too far into the structure of S methods to include here. See [Cha98] for more examples and details.

References

[BCW88] R. A. Becker, J. M. Chambers, and A. R. Wilks. *The New S Language.* Chapman and Hall, 1988.

[Cha98] John M. Chambers. *Programming with Data: A Guide to the S Language.* Springer-Verlag, 1998.