Group Generic Functions

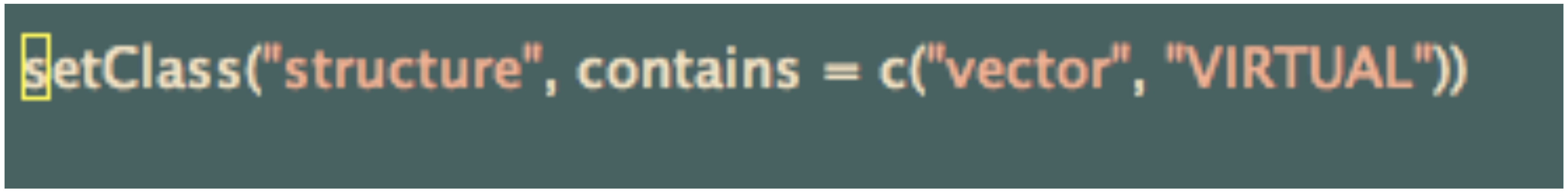
Generic functions can optionally share a "group": a function with the same arguments (signature, to be precise), that defines common behavior.

Group generic functions exist so methods can be defined to apply to all members of the group.

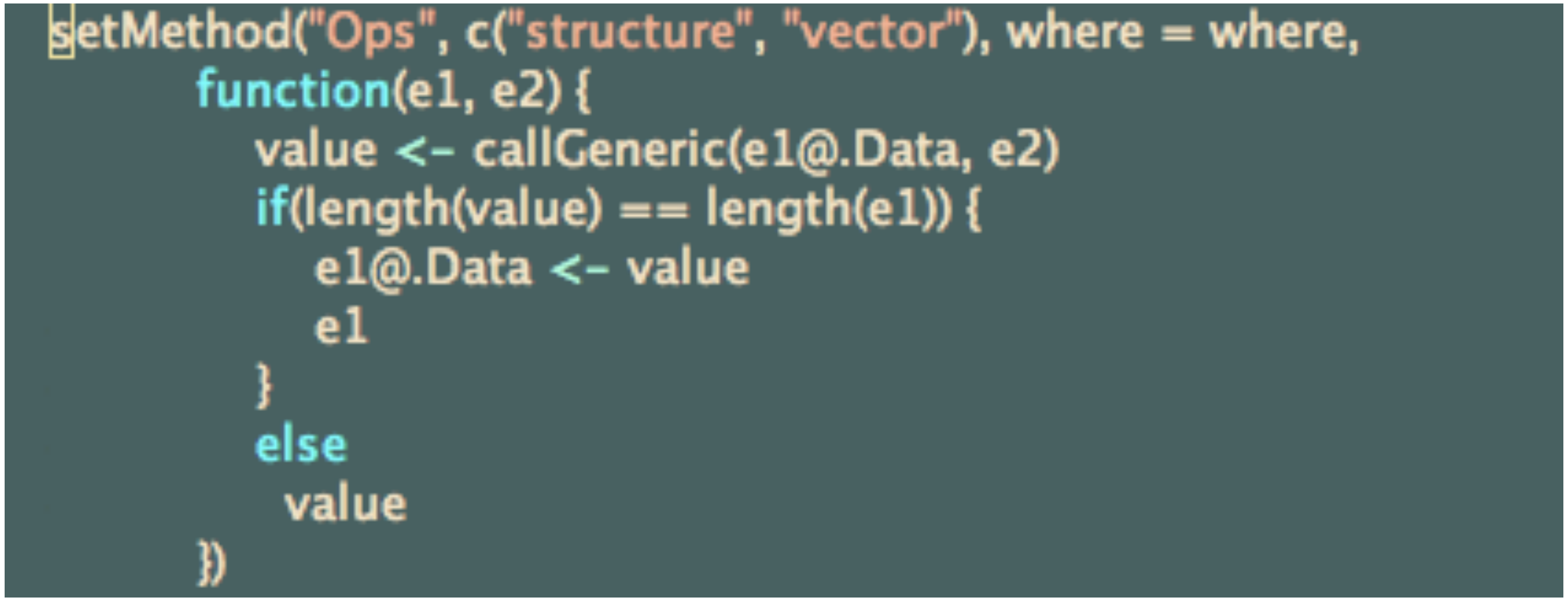
Examples: `"Arith"`, `"Compare"`, `"Math"`.

Groups can nest: `"Arith"`, `"Compare"` in `"Ops"`.

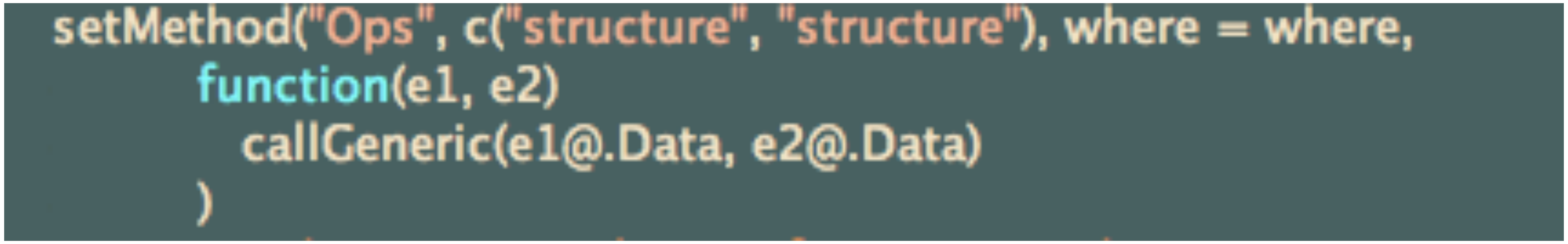
Essentially, inheritance for functions. Group methods are eligible for selection, but rank behind direct methods if both have the same "distance".



```
setClass("structure", contains = c("vector", "VIRTUAL"))
```



```
setMethod("Ops", c("structure", "vector"), where = where,
          function(e1, e2) {
            value <- callGeneric(e1@.Data, e2)
            if(length(value) == length(e1)) {
              e1@.Data <- value
              e1
            }
            else
             value
          })
```



```
setMethod("Ops", c("structure", "structure"), where = where,
          function(e1, e2)
            callGeneric(e1@.Data, e2@.Data)
          )
```

`callGeneric()` and `callNextMethod()`

`callGeneric()` calls the current generic function. Useful, indeed essential, in group generic methods.

`callNextMethod()` calls the "next" method; namely, the method that would have been selected if the current method didn't exist.

Both functions can be called with no arguments, in which case the current arguments are passed down. Or they can have arguments, treated like those in the generic, but *not* used to select the next method. For details see : `?callNextMethod`.

Creating Generic Functions

Generic functions are created automatically when a method is defined for an ordinary function.

You can create them explicitly, e.g., `setGeneric("f")`.

Either way: `f()` becomes the default method.

Why `setGeneric()` instead of just `setMethod()`?

•Slightly cleaner (no message generated);

•Or, you want to control default method or signature.

Generic Functions with no default

Some generic functions don't have a natural default: You *only* know how to do this for some classes.

Call `setGeneric()` with two arg's: name and a function definition (with the call to `standardGeneric()`).

Generic Function Signatures

Signature of a generic function is the subset of the formal arguments that can appear in the signature of a method.

By default it's all the arguments in order, except `"..."`.

THE CATCH: all arguments in the signature must be evaluated, to select the method. Can't use lazy evaluation on them, so can't use `subsitute()`, for example.

Solution: to keep lazy evaluation, exclude the argument from the signature.

Implicit Generic Functions

Sometimes you want to leave a function in its ordinary form, but still want to control something about it if methods are defined:

you want to exclude some arguments from signatures

you don't want the ordinary function to become the default method.

Solution: set up the generic the way you would like it, then call `setImplicitGeneric()`.

That stores away the generic version and restores the ordinary version.

Assignment

In the plot() method for class trackNumeric, we turned the numbers into three characters and used those as plotting symbols that we used as N values for the pch parameter.

Suppose we wanted something similar for class "trackVector". Now the data may be "numeric" but also, for example, "logical", or "character". The assignment is to produce a plot() method for "trackVector".

One way to think about a solution is to imagine a function, say makePoints(data), that takes the vector of data and returns the vector of symbols to use. In this form, a plot method for "trackNumeric" would be:

```
makePoints <- function(data) { points = c(".", "o", "*") which =
   as.integer(cut(data, length(points))) points[which] }

setMethod("plot", c("trackNumeric", "missing"), function(x, y, ...) {
          callNextMethod(x, pch = makePoints(x@.Data), ...)  })
```

Use setGeneric and setMethod to create a generic version of makePoints and a corresponding method for "plot" for c("trackVector", "missing").

Write a method for makePoints() when data is "logical", that returns a character vector with "T" and "F" corresponding to TRUE and FALSE values.

To get the functions, classes and methods, copy files "Examples/Feb27.R" and "Examples/with_gps1.rda" from the course web page and do

```
   source("Feb27.R")
```

You should have the classes and objects discussed in the lecture. Objects include tv1 from class "trackVector" and tvNA, another "trackVector" object with logical data, corresponding to is.na(tv1). You can use these to try out the plotting method.