R Programming

Object-Oriented Programming

(The S4 System)

The S4 Object System

- Because of problems with the S3 object system, John Chambers (the author of R's object systems) has moved to a more formally-based object system.
- This "S4" system is quite similar to the Common Lisp CLOS object system or the object system in the Dylan language.
- The system is still "in development," but already offers significant advantages over the S3 system.
- One of the greatest advantages is the use of a formal system of inheritance.

Formal Classes

- In the formal class system, objects belong to formally defined classes. A class consists of a number of named *slots*, each with a specific type or class.
- A class that represents coordinates in the plane might have two slots, named `x` and `y`, that contain numeric vectors of the same length.
- Classes are *declared* with a `setClass` statement. We could create a class to represent coordinates as follows.

```
> setClass("coords",
           slots = list(x = "numeric",
                        y = "numeric"))
```

Creating Objects

- Once a class has been created, objects from that class can be created with a call to the `new` function.

```
> pts = new("coords",
            x = rnorm(5), y = rnorm(5))
```

- The first argument to the function is the class name; the other arguments provide values for the slots in the object.

Constructor Functions

Generally, it is not a good idea to use `new` in this naked fashion. Instead it is better to embed object creation in a *constructor* function. This makes it possible to carry out some checks of slot validity.

```
> coords =
      function(x, y) {
          if (length(x) != length(y))
              stop("equal length x and y required")
          if (!is.numeric(x) || !is.numeric(y))
              stop("numeric x and y required")
          new("coords", x = as.vector(x),
                        y = as.vector(y))
      }
```

Validity Checking

- It is possible to include the checks in the constructor function as part of the class definition.
- To do this, you add a `validity` argument to the class definition.
- The validity argument is a function that, given an object created by `new`, will check the values in its slots to see of the object is “valid.”
- The function either returns `TRUE` or `FALSE`.
- Even if you do the checking with a validity function, it is still a good idea to encapsulate object creation inside a constructor function.

Constructing and Printing Objects

Objects from the `coords` class, known as *instances* of the class, can now be created with a call to the constructor function.

```
> pts = coords(round(rnorm(5), 2),
               round(rnorm(5), 2))
```

This kind of object can be printed just like any other R value.

```
> pts
An object of class "coords"
Slot "x":
[1]  0.85 -0.51 -1.46 -1.38  1.82

Slot "y":
[1] -1.11  2.22 -0.22 -1.30 -0.29
```

Accessing an Object's Slots

- The values in the slots within an object can be accessed with the *slot access operator* `@`.
- The slots are accessed by name.

```
> pts@x
[1]  0.85 -0.51 -1.46 -1.38  1.82

> pts@y
[1] -1.11  2.22 -0.22 -1.30 -0.29
```

Accessor Functions

- The code fragments `pts@x` and `pts@y` reveal a little too much of the internal structure of the `coords` class.
- Rather than getting the values directly in this way, it is better to use *accessor functions* that provide indirect access.
- That way the internal structure of the class can be changed more easily.

```
> xcoords = function(obj) obj@x
> ycoords = function(obj) obj@y
```

Generic Functions and Methods

- Formal classes are the basis for a clean object-oriented mechanism in R.
- The mechanism uses a special type of function called a *generic function*.
- A generic function acts as a kind of switch that selects a particular function or *method* to invoked.
- The particular method selected depends on the class of a number of nominated arguments.
- The types of the nominated arguments define the *signature* of the method.

The "show" Generic Function

The display (i.e. printing) of objects is handled by the `show` generic function.

We can see that `show` is a special kind of function if we print it.

```
> show
standardGeneric for "show"
    defined from package "methods"

function (object)
standardGeneric("show")
<bytecode: 0x2cca9e0>
<environment: 0x3810fa8>
Methods may be defined for arguments: object
Use  showMethods("show")  for currently available ones
(This generic function excludes non-simple inheritance
```

Defining a "`show`" Method

Methods for show have a single argument called `object`. An appropriate `show` method for `coords` objects can be defined as follows. Here is how we could create a display method for the `coords` class.

```
> setMethod(show, signature(object = "coords"),
            function(object)
            print(data.frame(x = xcoords(object),
                             y = ycoords(object))))
[1] "show"
attr(,"package")
[1] "methods"
```

Notice that the slots are accessed using the accessor functions rather than directly.

Using a Method

The *show* method will be used whenever an (implicit or explicit) attempt is made to print an object.

```
> pts
      x     y
1  0.85 -1.11
2 -0.51  2.22
3 -1.46 -0.22
4 -1.38 -1.30
5  1.82 -0.29
```

Defining New Generic Functions

If a function is not generic, it possible to create a new generic using the `setGeneric` function.

```
> setGeneric("display",
             function(obj)
             standardGeneric("display"))
[1] "display"
```

Writing a “display” Method

Once a function is defined as generic, new methods can be written for it. In the case of the coords class, we may choose to display the coordinates as pairs of values. It is easy to implement a method that will do this.

```
> setMethod("display", signature(obj = "coords"),
            function(obj)
            print(paste("(",
                        format(xcoords(obj)),
                        ", ",
                        format(ycoords(obj)),
                        ")", sep = ""),
                  quote = FALSE))
[1] "display"
```

Using the “`display`” Method

A call to the generic function `display` will be dispatched to the method just defined when the argument is of class `coords`.

```
> display(pts)
[1] ( 0.85, -1.11) (-0.51,  2.22)
[3] (-1.46, -0.22) (-1.38, -1.30)
[5] ( 1.82, -0.29)
```

Bounding Boxes

- One thing we might be interested in having for objects like those in the `coords` class, is a `bbox` method that will compute the two dimensional bounding box for the coordinates.

- We'll make the function generic so that methods can be defined for other classes too.

```
> setGeneric("bbox",
             function(obj)
             standardGeneric("bbox"))
[1] "bbox"
```

Implementing a Bounding Box Method

We can implement a bounding box method for the `coords` class as follows.

```
> setMethod("bbox", signature(obj = "coords"),
            function(obj)
            matrix(c(range(xcoords(obj)),
                     range(ycoords(obj))),
                   nc = 2,
                   dimnames = list(
                     c("min", "max"),
                     c("x:", "y:"))))
[1] "bbox"
```

Example: Bounding Box

```
> pts
      x     y
1  0.85 -1.11
2 -0.51  2.22
3 -1.46 -0.22
4 -1.38 -1.30
5  1.82 -0.29

> bbox(pts)
       x:    y:
min -1.46 -1.30
max  1.82  2.22
```

Inheritance

- The `coords` class provides a way to represent a set of spatial locations.
- Now suppose that we want a class that provides a numerical value to go along with each spatial location.
- We could define an entirely new class to do this, but it is better to simply add the value to our existing `coords` class.
- The new class will then *inherit* the spatial properties of the `coords` class.

Inheritance: The Class Declaration

Here is a declaration of the new class.

```
> setClass("vcoords",
           slots = list(value = "numeric"),
           contains = "coords")
```

This says that a `vcoords` object contains a numeric `value` slot and *inherits* the slots from the `coords` class.

Inheritance: A Constructor Function

```
> vcoords =
      function(x, y, value)
      {
          if (!is.numeric(x) ||
              !is.numeric(y) ||
              !is.numeric(value) ||
              length(x) != length(value) ||
              length(y) != length(value))
                  stop("invalid arguments")
          new("vcoords", x = x, y = y,
              value = value)
      }

> values = function(obj) obj@value
```

Example: A `vcoords` Object

Defining a `vcoords` object is simple

```
> vpts = vcoords(xcoords(pts), ycoords(pts),
                 round(100 * runif(5)))
```

but printing it gives an unexpected result.

```
> vpts
      x     y
1  0.85 -1.11
2 -0.51  2.22
3 -1.46 -0.22
4 -1.38 -1.30
5  1.82 -0.29
```

Inherited Methods

- The printing result occurs because the `vcoords` class doesn't just inherit the slots of the `coords` class. It also inherits its methods.

- A `vcoords` is also a `coords` object. When a search is is made for an appropriate `print` method, and no `vcoords` method is found, the `coords` method is used.

- If we want a method for printing `vcoords` objects, we have to define one.

A Print Method for `vcoords` Objects

```
> setMethod(show, signature(object = "vcoords"),
            function(object)
            print(data.frame(
                     x = xcoords(object),
                     y = ycoords(object),
                     value = values(object))))
[1] "show"
attr(,"package")
[1] "methods"
```

Printing `vcoords` Objects

```
> vpts
      x     y value
1  0.85 -1.11    76
2 -0.51  2.22    27
3 -1.46 -0.22    24
4 -1.38 -1.30    36
5  1.82 -0.29    12
```

Mathematical Transformations

- The `vcoords` class contains a numeric slot that can be changed by mathematical transformations.
- We may wish to:
  - apply a mathematical function to the values,
  - negate the values,
  - add, subtract, multiply or divide corresponding values in two objects,
  - compare values in two objects,
  - etc.
- Defining appropriate methods will allow us to do this.

Mathematical Functions

Here is how we can define an cos method.

```
> setMethod("cos", signature(x = "vcoords"),
            function(x)
            vcoords(xcoords(x),
                    ycoords(x),
                    cos(values(x))))
[1] "cos"
> cos(vpts)
      x     y      value
1  0.85 -1.11  0.8243313
2 -0.51  2.22 -0.2921388
3 -1.46 -0.22  0.4241790
4 -1.38 -1.30 -0.1279637
5  1.82 -0.29  0.8438540
```

Mathematical Functions

A sin method is very similar.

```
> setMethod("sin", signature(x = "vcoords"),
            function(x)
            vcoords(xcoords(x),
                    ycoords(x),
                    sin(values(x))))
[1] "sin"
> sin(vpts)
      x     y      value
1  0.85 -1.11  0.5661076
2 -0.51  2.22  0.9563759
3 -1.46 -0.22 -0.9055784
4 -1.38 -1.30 -0.9917789
5  1.82 -0.29 -0.5365729
```

Group Methods

In fact most of R's mathematical functions would require an almost identical definition. There is actually a short-hand way of defining all the methods with one function definition.

```
> setMethod("Math", signature(x = "vcoords"),
            function(x)
            vcoords(xcoords(x),
                    ycoords(x),
                    callGeneric(values(x))))
[1] "Math"
```

This provides definitions for all the common mathematical functions.

Group Methods

```
> sqrt(vpts)
      x     y    value
1  0.85 -1.11 8.717798
2 -0.51  2.22 5.196152
3 -1.46 -0.22 4.898979
4 -1.38 -1.30 6.000000
5  1.82 -0.29 3.464102

> tan(vpts)
      x     y      value
1  0.85 -1.11  0.6867477
2 -0.51  2.22 -3.2737038
3 -1.46 -0.22 -2.1348967
4 -1.38 -1.30  7.7504709
5  1.82 -0.29 -0.6358599
```

Functions Handled by the Math Group

The following functions are handled by the `Math` group.

> `abs`, `sign`, `exp`, `sqrt`, `log`, `log10`, `log2`,
> `cos`, `sin`, `tan`, `acos`, `asin`, `atan`,
> `cosh`, `sinh`, `tanh`, `acosh`, `asinh`, `atanh`,
> `ceiling`, `floor`, `trunc`,
> `gamma`, `lgamma`, `digamma`, `trigamma`
> `cumprod`, `cumsum`, `cummin`, `cummin`.

There is also a `Math2` group (with a second argument called `digits`) that contains the following functions.

> `round`, `signif`.

Binary Operations

- There are many binary operations R.
- Examples are:
  - The arithmetic operators:

    `+`, `-`, `*`, `^`, `%%`, `%/%`, `/`
  - The comparison operators:

    `==`, `>`, `<`, `!=`, `<=`, `>=`
- These operators are all *generic*, and methods can be defined for them.
- The operators belong to the groups `Arith` and `Compare`, which both belong to the larger group `Ops`.

Binary Operators as Functions

- Any binary operator can be thought of as a function of two variables.
- The function has the form

```
function(e1, e2) {

. . .

}
```

- A call of the form `x + y` can be thought of as a call to a function like the one above.

Compatibility of `vcoords` Objects

- Arithmetic on `vcoords` objects only makes sense if the objects are defined at identical locations.
- Here is function that will check whether two `vccords` objects are defined at the same locations.

```
> sameloc =
        function(e1, e2)
        (length(values(e1)) == length(values(e2))
        || any(xcoords(e1) == xcoords(e2))
        || any(ycoords(e1) == ycoords(e2)))
```

Defining Methods for Arithmetic Operators

Here is a group method that will define all the necessary binary operators for arithmetic on `vcoords` objects.

```
> setMethod("Arith", signature(e1 = "vcoords",
                               e2 = "vcoords"),
      function(e1, e2)
      {
          if (!sameloc(e1, e2))
              stop("identical locations required")
          vcoords(xcoords(e1),
                  ycoords(e1),
                  callGeneric(values(e1),
                              values(e2)))
      })
[1] "Arith"
```

Example: Adding `vcoords`

```
> vpts
      x     y value
1  0.85 -1.11    76
2 -0.51  2.22    27
3 -1.46 -0.22    24
4 -1.38 -1.30    36
5  1.82 -0.29    12

> vpts + vpts
      x     y value
1  0.85 -1.11   152
2 -0.51  2.22    54
3 -1.46 -0.22    48
4 -1.38 -1.30    72
5  1.82 -0.29    24
```

Defining Methods for Comparison Operators

A similar definition will work for the `Compare` group.

```
> setMethod("Compare", signature(e1 = "vcoords",
                                 e2 = "vcoords"),
      function(e1, e2)
      {
          if (!sameloc(e1, e2))
              stop("identical locations required")
          callGeneric(values(e1), values(e2))
      })
[1] "Compare"
```

Notice that this returns a logical vector rather than a `vcoords` object.

Additional Definitions

- The definitions of the binary operators given above only work for combining two `vcoords` objects.
- It may also be useful to define operations like `x + 10` or `x > 3`.
- This can be done by defining addition methods for combining `vcoords` objects and numeric objects.
- Care must be taken to make this work correctly.

Additional Methods

The following method will make expressions like `1 + x` and `3 * y` work correctly.

```
> setMethod("Arith",
      signature(e1 = "numeric",
                e2 = "vcoords"),
     function(e1, e2) {
         if (length(e1) > length(values(e2)))
             stop("incompatible lengths")
         vcoords(xcoords(e2),
                 ycoords(e2),
                 callGeneric(as.vector(e1),
                             values(e2)))
            })
[1] "Arith"
```

Example: Scaling `vcoords`

```
> vpts
      x     y value
1  0.85 -1.11    76
2 -0.51  2.22    27
3 -1.46 -0.22    24
4 -1.38 -1.30    36
5  1.82 -0.29    12
```

```
> 3 * vpts
      x     y value
1  0.85 -1.11   228
2 -0.51  2.22    81
3 -1.46 -0.22    72
4 -1.38 -1.30   108
5  1.82 -0.29    36
```

Additional Methods

The following method will make expressions like `1 + x` and `3 * y` work correctly.

```
> setMethod("Arith",
      signature(e1 = "vcoords",
                e2 = "numeric"),
     function(e1, e2) {
         if (length(values(e1)) < length(e2))
             stop("incompatible lengths")
         vcoords(xcoords(e1),
                 ycoords(e1),
                 callGeneric(values(e1),
                             as.vector(e2)))
     })
[1] "Arith"
```

Example: Scaling `vcoords`

```
> vpts
      x     y value
1  0.85 -1.11    76
2 -0.51  2.22    27
3 -1.46 -0.22    24
4 -1.38 -1.30    36
5  1.82 -0.29    12

> vpts / 2
      x     y value
1  0.85 -1.11  38.0
2 -0.51  2.22  13.5
3 -1.46 -0.22  12.0
4 -1.38 -1.30  18.0
5  1.82 -0.29   6.0
```

Example: `vcoords` Powers

```
> vpts
      x     y value
1  0.85 -1.11    76
2 -0.51  2.22    27
3 -1.46 -0.22    24
4 -1.38 -1.30    36
5  1.82 -0.29    12

> vpts^2
      x     y value
1  0.85 -1.11  5776
2 -0.51  2.22   729
3 -1.46 -0.22   576
4 -1.38 -1.30  1296
5  1.82 -0.29   144
```

Testing Class Membership

The function `is` allows us to test whether an object belongs to a particular class.

```
> is(vpts, "vcoords")
[1] TRUE
```

Remember that the `vcoords` class is defined as inheriting from the `coords` class. So every `vcoords` object is also a `coords` object.

```
> is(vpts, "coords")
[1] TRUE
```

Coercion of Objects

Class inheritance provides a natural way of coercing objects from one class to another. The function `as` can be used to do this.

```
> as(vpts, "coords")
      x     y
1  0.85 -1.11
2 -0.51  2.22
3 -1.46 -0.22
4 -1.38 -1.30
5  1.82 -0.29
```

Coercion only works in the direction of inheritance (it is easy to discard slots).

Subsetting

It is likely that we will want to take subsets of `coords` and `vcoords` objects. We can do this by defining methods for the `[` generic.

```
> setMethod("[",
            signature(x = "vcoords",
                      i = "ANY",
                      j = "missing",
                      drop = "missing"),
            function(x, i, j)
            vcoords(xcoords(x)[i],
                    ycoords(x)[i],
                    values(x)[i]))
[1] "["
```

Example

```
> vpts[1:3]
      x     y value
1  0.85 -1.11    76
2 -0.51  2.22    27
3 -1.46 -0.22    24

> vpts[values(vpts) > 50]
     x     y value
1 0.85 -1.11    76
```