# Exercises and solutions for chapter 'Object-Oriented Programming in R'

August 11, 2008

**Exercise 1**
*Define a class for passenger names that has slots for the first name, middle initial and last name. Change the definition of the Passenger class to reflect your new class. Does this change the inheritance properties of the Passenger class or the FreqFlyer class?*

**Solutions:** None available, yet.

**Exercise 2**
*Write a simple* `show` *method for the Passenger class. Write a* `show` *method for theFreqFlyer class that makes use of the* `show` *method for passengers. For S4 you will want to use* `setMethod` *and* `callNextMethod`*, while for an S3 implementation you will need to use* `NextMethod` *and name the print methods* `print.Passenger` *and* `print.FreqFlyer`*.*

**Solutions:** We describe an S4 solution here.

```
> setMethod("show", "Passenger", function(object) {
+     cat("Name: ", object@name, "\n")
+     cat("Origin: ", object@origin, "\n")
+     cat("Destination:", object@destination,
+         "\n")
+ })
[1] "show"
> p1 = new("Passenger", name = "J. Biologist",
+     origin = "YXY", destination = "TGL")
> p1
Name:  J. Biologist
Origin:  YXY
Destination: TGL
```

And now we can add a show method for frequent flyers that reuses all the code for passengers.

```
> setMethod("show", "FreqFlyer", function(object) {
+     callNextMethod()
+     cat("Freq Flyer no: ", object@ffnumber,
+         "\n")
+ })
[1] "show"
> p2 = new("FreqFlyer", name = "K. Biologist",
+     origin = "YVR", destination = "LAX",
+     ffnumber = 1)
> p2
Name:  K. Biologist
Origin:  YVR
Destination: LAX
Freq Flyer no:  1
```

**Exercise 3**
*The S3 system has been used for some years and a very extensive set of tools for statistical modeling has been developed based on this system (**?**). Among the builtin classes is glm. Fit a simple generalized linear model (using an example from the help page for* `glm` *is the easiest way) and examine its structure. What classes does glm extend? What are the slots in a glm instance?*

**Solutions:**

```
> counts = c(18, 17, 15, 20, 10, 20, 25, 13,
+     12)
> outcome = gl(3, 1, 9)
> treatment = gl(3, 3)
> d.AD = data.frame(treatment, outcome, counts)
> glm.D93 = glm(counts ~ outcome + treatment,
+     family = poisson())
> class(glm.D93)
[1] "glm" "lm"
> is.list(glm.D93)
[1] TRUE
> attr(glm.D93, "class")
[1] "glm" "lm"
```

And we see that instances of the class *glm* inherit from *lm* and that instances of this class do have a `class` attribute. We can also see that the class is implemented as a list and hence we can determine the slot names by simply calling the names function.

```
> names(glm.D93)
 [1] "coefficients"      "residuals"
 [3] "fitted.values"     "effects"
 [5] "R"                 "rank"
 [7] "qr"                "family"
 [9] "linear.predictors" "deviance"
[11] "aic"               "null.deviance"
[13] "iter"              "weights"
[15] "prior.weights"     "df.residual"
[17] "df.null"           "y"
[19] "converged"         "boundary"
[21] "model"             "call"
[23] "formula"           "terms"
[25] "data"              "offset"
[27] "control"           "method"
[29] "contrasts"         "xlevels"
```

**Exercise 4**
*Returning to our ExpressionSet example, Section* **??**, *instances of EXPRS3 can be very large and we want to control the default information that is printed by R. Write S3* `print` *methods for the PHENODS3 and EXPRS3 classes.*

**Solutions:**

```
> print.PHENODS3 = function(object) {
+     dm = dim(object$pData)
+     cat("instance of PHENODS3 with", dm[2],
+         "variables")
+     cat("and", dm[1], "cases\n")
+     vL = object$varLabels
+     cat("\t varLabels\n")
+     nm = names(vL)
+     for (i in seq(along = vL)) cat("\t\t",
+         nm[[i]], ": ", vL[[i]], "\n", sep = "")
+ }
> print.EXPRS3 = function(object) {
+     dm = dim(object$exprs)
+     cat("instance of EXPRS3\n")
+     cat("number of genes:", dm[1], "\n")
+     cat("number of samples:", dm[2], "\n")
+     print(object$phenoData)
+ }
```

**Exercise 5**
*Write a replacement method for the following problem. Let* `x` *be a matrix with named rows. Define* `x$a = y` *to mean that the row of* `x` *named* `a` *be set to* `y`*. Because* `$` *is an internal generic, it will only dispatch on objects for which* `is.object` *is* `TRUE`*, so you will need to set the* `oldClass`*.*

**Solutions:**

```
> "$<-.matrix" = function(x, name, value) {
+     if (!name %in% row.names(x))
+         stop("bad name")
+     x[name, ] = value
+     x
+ }
```

And then we can test it.

```
> x = matrix(1:10, nr = 5)
> rownames(x) = letters[1:5]
> oldClass(x) = "matrix"
> x$c = c(100, 100)
> x
  [,1] [,2]
a    1    6
b    2    7
c  100  100
d    4    9
e    5   10
attr(,"class")
[1] "matrix"
```

**Exercise 6**
*What happens if you generate a second instance of the Ex1 class? Why might this not be desirable? Examine the prototype for the class and see if you can understand what has happened. Will changing the prototype to* `list(s1=quote(rnorm(10)))` *fix the problem?*

**Solutions:** The issue is that the function call to `rnorm` has been executed and its value is stored, not the call.

```
> b2 = new("Ex1")
```

So if the user hoped to get a different set of $N(0, 1)$ values for each instance that is not possible. No, using `quote` does not really help. There is no simple way to store a function that will generate the value for a slot in

the prototype. However, this is trivial if you use a constructor function, as we do below. Some good examples of constructor functions can be found in the **GSEABase** package.

```
> makeex = function() {
      obj = new("Ex1")
      obj@s1 = rnorm(10)
  }
> b2 = makeex()
```

**Exercise 7**
*Return to the first representation of the Rectangle class example of Section* **??** *and write a validity method that ensures that the value placed in the* `area` *slot is indeed the product of the width and the height.*

**Solutions:** No solution, currently.

**Exercise 8**
*Write a function that searches every package on the search path for any class that extends oldClass.*

**Solutions:** None available yet.

**Exercise 9**
*Define another method for the generic function* `foo` *defined above, with a different signature. Test that the correct method is dispatched to for different arguments.*

**Exercise 10**
*Write different methods for the generic function* `foo` *defined above, that make use of ANY, and missing in the signature. Test these methods to be sure they behave as you expect.*

**Exercise 11**
*Show that for S4 classes,* `is` *gets the inheritance correctly while* `inherits` *does not.*

**Solutions:**

```
> setClass("A")
[1] "A"
> setClass("B", representation(s = "numeric"), contains = "A")
[1] "B"
> y = new("B")
> is(y, "A")
[1] TRUE
> inherits(y, "A")
[1] FALSE
```

**Exercise 12**
*Plot the graph that corresponds to the second largest connected component. What classes does it contain?*

**Solutions:** Once you find which connected component, then the code for rendering, from above can be reused.

```
> whichcc = as.integer(names(sort(complens, decr = T))[2])
> subGN = ccomp[[whichcc]]
> subG = subGraph(subGN, graphClassgraph)
> x = layoutGraph(subG, attrs = list(node = list(shape = "ellipse",
      fixedsize = FALSE)))
> renderGraph(x)
```

**Exercise 13**
*Write a function to compute the has-a relationships between all classes in a package. You will probably want to also include classes that are not defined in the package, but appear in slot specifications. You might not want to worry too much about* `classUnions` *at this point, but a comprehensive solution would need to deal with them.*

**Solutions:** None available, yet.