oject-oriented programming in R

R differs from other **object-oriented programming** languages like C++, Java, and Python, but possesses a number of properties unique to those of **OOP**:

… Everything in R is an object—from numbers to character strings, etc.
… R utilizes ***encapsulation***—where separate, but related data are packaged into single class instances. ***Encapsulation*** enhances clarity within code.
… R classes are ***polymorphic***—where the same function call executes different operations on different object classes. ***Polymorphism*** enhances reuse and reproduction.
… R allows ***inheritance***—extending a given class to a more specialized and distinct class.

s3 classes **S3** in R

The original class structures in R are known as **S3** classes. An **S3** class consists of a list, with a class name attribute and ***dispatch*** capability; allowing generic function use. **S4** classes were developed with the intent of added safety and security to the coding environment; preventing access to a nonexistent class component.

The concept of ***polymorphism*** play a key role in allowing ***generic functions*** and code to be written, regardless of object classes. The following example runs R's regression function **lm( )**:

```
> lmout <- lm(y ~ x)
> class(lmout)          #because the object class is lm ...
[1] "lm"

> lmout                 #the generic print() function dispatches to print.lm() method

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
       -3.0          3.5
```

The above illustration of ***generic functions*** is can be explained by looking at the **print( )** function:

```
> print
function (x, ...)
UseMethod("print")
<bytecode: 0x00000000049d2d38>
<environment: namespace:base>
```

The single call to **UseMethod( )** signifies a ***dispatcher*** function to the appropriate class methods. If the class is removed from the variable **lmout** in the above illustration, the output is unaltered and superfluous:

```
> unclass(lmout)
$coefficients
(Intercept)           x
       -3.0         3.5

$residuals
   1    2    3
 0.5 -1.0  0.5

$effects
(Intercept)           x
  -6.928203   -4.949747    1.224745
$rank
[1] 2

$fitted.values
  1   2   3
0.5 4.0 7.5

$assign
[1] 0 1

$qr
$qr
  (Intercept)          x
1  -1.7320508 -3.4641016
2   0.5773503 -1.4142136
3   0.5773503  0.9659258
attr(,"assign")
[1] 0 1
```

implementations of generic methods in R

All implementations of a given generic method can be found using the **methods( )** functions:

```
> methods(print)    # *asterisks* denote nonvisible functions; not in default namespace
  [1] print.acf*
  [2] print.anova*
  ...
```

The ***nonvisible*** functions in the default namespace can be accessed through the **getAnywhere( )** function:

Illustrating the latter by using the **print.aspell( )** method; **aspell( )** performs spellcheck on a file argument.

Example file ***wrd*** contains the following text: `[1] "Which word is mispelled?"` , applied as follows:

```
> aspell("wrds")
mispelled
  wrds:1:15
```

The point of focus is what mechanism was used to print the output. The **aspell( )** function returns an object of class **"aspell"**. R does not have a generic **print.aspell( )** method. Instead, R called **UseMethod( )** on the object of class **"aspell"**. Furthermore, if the print method is called directly, R will **<u>not</u>** recognize it:

```
> print.aspell(wrds)
Error: could not find function "print.aspell"
```

The solution to the above error is to access the method by calling the **getAnywhere( )** function:

```
> getAnywhere(print.aspell)
A single object matching ‘print.aspell’ was found
It was found in the following places
  registered S3 method for print from namespace utils
  namespace:utils
with value

function (x, ...)
{
    if (nrow(x))
        writeLines(paste(format(x, ...), collapse = "\n\n"))
    invisible(x)
}
<bytecode: 0x000000000309e2a0>
<environment: namespace:utils>
```

As seen above, the function belongs to the **utils** namespace, and thus can be accessed as such:

```
> utils:::print.aspell("wrds")
mispelled
  wrds:1:15
```

Additionally, all of the generic methods can be printed through the **methods( )** function:

```
> methods(class = "default")
  ...
```

writing s3 classes in R

A class is created by forming a list, with the list components representing member variables of the class. The “class” attribute is set manually by calling the **attr( )** or **class( )** function; various implementations of the generic functions are then defined. The latter can be seen in the **lm( )** function as applied earlier:

```
> lm
  ...
      z <- list(coefficients = if (is.matrix(y)) matrix(, 0,
            3) else numeric(), residuals = y, fitted.values = 0 *
            y, weights = w, rank = 0L, df.residual = if (!is.null(w)) sum(w !=
            0) else if (is.matrix(y)) nrow(y) else length(y))
  ...
      class(z) <- c(if (is.matrix(y)) "mlm", "lm")
  ...
```

A listed is created, assigned to **z**, serving as the **“lm”** class instance framework (eventually a function return value). Some components of the latter list were previously assigned (**residuals**). Additionally, the class attribute was set to **“lm”** (the **“mlm”** discussed later).

Constructing a class for the employee example used previously is illustrated as follows:

```
> j <- list(name = "Joe", salary = 50000, union = TRUE)
> class(j) <- "employee"
> attributes(j)
$names
[1] "name"   "salary" "union"

$class
[1] "employee"
```

The **“employee”** class is created with variable **j** as its formal argument; printed with default method below:

```
> j
$name
[1] "Joe"

$salary
[1] 50000

$union
[1] TRUE

attr(,"class")
[1] "employee"
```

The call to print **j** resulted in treatment as a list for printing purposes; print method created below:

```
> print.employee <- function(wrkr) {
+     cat(wrkr$name, "\n")
+     cat("salary", wrkr$salary, "\n")
+     cat("union member", wrkr$union, "\n")
+ }

> methods(,"employee")         #any call to print() on a object class "employee" applies
[1] print
```

The above creation of an **“employee”** class is illustrated and proven by printing variable **j** as follows:

```
> j
Joe
salary 50000
union member TRUE
```

using inheritance in R

The notion of ***inheritance*** forms new classes as specialized versions of old classes. Following the employee example, a new class can be created for hourly employees **"hrlyemployee"**; a new subclass of **"employee"**.

```
> k <- list(name = "Kate", salary = 68000, union = FALSE, hrsthismonth = 2)
> class(k) <- c("hrlyemployee", "employee")
```

Note that the new class has an additional variable: **hrsthismonth**. The name of the created class consists of two character strings (the new class, the old class). The ***inheritance*** is seen when variable **k** is printed:

```
> k
Kate
salary 68000
union member FALSE
```

Concretely, the execution of variable **k** from the command line called the **print( )** function; which in turn called the **UseMethod( )** function to search for the **first** of **k**'s two class names (**"hrlyemployee"**). Because there was no method for the **"hrylyemployee"** class, the **UseMethod( )** function searched **employee"**.

Recalling the **"lm"** class from earlier, the following line of code makes sense, as **"mlm"** is a subclass of **"lm"**:

```
> lm
  ...
      class(z) <- c(if (is.matrix(y)) "mlm", "lm")
  ...
```

s4 classes **S4** in R

Some programming schools of thought find S3 classes leave more exposure to the code in R, opposed the expected safety that exists in other **OOP** languages. For example, consider the **employee** example from earlier; three fields names – **name salary union**. The following is a few examples of S3 class exposure:

… Union status is left NULL upon entry
… *Union* is misspelled as *onion*
… A object is created of some class other than the **"employee"** class, but mistakenly assigns the object's class attribute to **"employee"**.

S3 classes will produce no warnings to the above instances; in theory, **S4** classes will warn of the mishaps. **S4** structures are known to be more robust than **S3** classes, some of the basic differences are as follows:

Operation	S3 class	S4 class
Define class	Implicit in constructor code	setClass()
Create object	Build list, set class attribute	New()
Reference member variable	$	@
Implement generic f()	Define f.classname()	setMethod()
Declare generic	UseMethod()	setGeneric()

writing s4 classes in R

S4 classes are defined by calling the **setClass( )** function. The following continues the **employee** problem:

```
> setClass("employee",
+         representation(
+             name = "character",
+             salary = "numeric",
+             union = "logical")
+         )
> employee
```

the S4 class **"employee"** is created. An instance of the class is created by the S4 **new( )** *constructor* function.

```
> joe <- new("employee", name = "Joe", salary = 55000, union = TRUE)
> joe
An object of class "employee"
Slot "name":
[1] "Joe"

Slot "salary":
[1] 55000

Slot "union":
[1] TRUE
```

Member variables now exist in **"slots"** and are referenced with the **@** symbol; opposed to **$** with S3 classes:

```
> joe@salary
[1] 55000

> slot(joe,"salary")
[1] 55000

> joe@salary <- 88000
> joe@salary
[1] 88000
```

Equally, new assignments can be made to the slots through the **@** symbol. Lastly, to note the improved control around S4 classes, consider the following assignment to a misspelled slot:

```
> joe@salry <- 88000
Error in (function (cl, name, valueClass)  :
  'salry' is not a slot in class "employee"
```

Conversely, an S3 class would not restrict the latter example.

implementing a generic function on an S4 class in R

The **setMethod( )** function is used to define an implementation of a generic function on an S4 class. The following illustration will perform the latter with the **show( )** function; the S4 *analog* of S3's generic **print( )**.

```
> joe
An object of class "employee"
Slot "name":
[1] "Joe"

Slot "salary":
[1] 88000

Slot "union":
[1] TRUE
```

```
> show(joe)
An object of class "employee"
Slot "name":
[1] "Joe"

Slot "salary":
[1] 88000

Slot "union":
[1] TRUE
```

Note that the above default print of **joe** actually calls the **show( )** function for **S4 class objects**. The following code redefines the **show( )** function for the **"employee"** class created previously:

```
> setMethod("show", "employee",
+           function(object) {
+               inorout <- ifelse(object@union, "is", "is not")
+               cat(object@name, "has a salary of", object@salary,
+                   "and", inorout, "in the union", "\n")
+           }
+           )
[1] "show"
```

The initial argument in the **setMethod( )** function gives the name of the generic function for class-specific definition; the second argument specifies the class name; the third argument defines the new function:

```
> joe
Joe has a salary of 88000 and is in the union
```

class comparisons **S3** *VS* **S4** in R

The general tradeoff between classes—the convenience of S3 classes, or the security of S4 classes.

A concrete comparison between the S3 and S4 classes is given by **various contributors**.

managing objects in R

As a programmer increases the amount of objects accumulated over time, there are various tools to help:

… The **ls( )** function
… The **rm( )** function
… The **save( )** function
… The **class( )** and **mode( )** function, among others that provide information on object structure
… The **exists( )** function

The **ls( )** command lists all current objects in the workspace. A useful named argument in the **ls( )** functions is **pattern**; enabling *wildcards*:

```
> ls()
 [1] "%subsetof%" "a"          "b"          "c32"
 [5] "d"          "dm"         "exactlyone" "f"
 [9] "inc"        "m"          "r"          "symdiff"
[13] "t"          "two"        "u"          "w"
[17] "x"          "y"          "z"

> ls(pattern = "a")
[1] "a"          "exactlyone"
```

The **rm( )** function is used to remove objects that are no longer needed from the workspace.

```
> rm("%subsetof%")
> ls()
 [1] "a"          "b"          "c32"        "d"
 [5] "dm"         "exactlyone" "f"          "inc"
 [9] "m"          "r"          "symdiff"    "t"
[13] "two"        "u"          "w"          "x"
[17] "y"          "z"
```

A call to **rm(list = ls( ))** will clear the entire workspace of all objects.

```
> rm(list = ls())
> ls()
character(0)
```

Additionally, the above arguments **list** and **pattern** can be combined to effectively manage an environment:

```
> rm(list=ls(pattern = "a"))
> ls()
 [1] "b"          "c32"        "d"         "dm"
 [5] "f"          "inc"        "m"         "r"
 [9] "symdiff"    "t"          "two"        "u"
[13] "w"          "x"          "y"          "z"
```

The **browseEnv( )** is also helpful; it opens a web browser to display the global objects and various details.

A collection of objects can be written to disk with the **save( )** function; retrieved with **load( )** or **attach( )**:

```
save(..., list = character(),
     file = stop("'file' must be specified"),
     ascii = FALSE, version = NULL, envir = parent.frame(),
     compress = isTRUE(!ascii), compression_level,
     eval.promises = TRUE, precheck = TRUE)

load(file, envir = parent.frame(), verbose = FALSE)

attach(what, pos = 2L, name = deparse(substitute(what)),
       warn.conflicts = TRUE)
```

When the exact **structure** of an object returns from a library functions is needed, beyond documentation:

The following functions can be utilized to examine object structures in R:						
class()	mode()	names()	attributes()	unclass()	str()	edit()

R contains facilities for constructing *contingency tables*, **discussed previously:**

```
> ct <- table(ct)
> ct
          Voted.for.X.Last.Time
Vote.for.X No Yes
  No        2   0
  Not Sure  0   1
  Yes       1   1
```

```
> ctu <- unclass(ct)
> ctu
          Voted.for.X.Last.Time
Vote.for.X No Yes
  No        2   0
  Not Sure  0   1
  Yes       1   1

> class(ctu)
[1] "matrix"
```

The object **cttab** is returned by the function **table( )**, of class **"table"**. The documentation can be referenced via **?table**, or the class can be explored through the available functions in R

Illustrated on the left, the counts portion of the object is a matrix. Note the names of the object are displayed as belonging to the matrix. If the **unclass( )** function as not applied in this context, the output would be dependent on the default **print( )** method for the assigned class of the object (no difference in this case). The **str( )** functions operates similarly in a more compact form. Note that **unclass( )** still applies a class.

Expanding on the above table to examine the underlying code of the **table( )** function used above:

```
> edit(table)   #opens a window display of code, page() does the same through export
```

The **edit( )** function opens the function passed as an argument into another widow for adjustment. The code can then be browsed through a text editor, finding the following code at the end of the **table( )** function:

```
  y <- array(tabulate(bin, pd), dims, dimnames = dn)              #line 77 of table()
  class(y) <- "table"                                             #line 78 of table()
  y                                                               #line 79 of table()
```

The above reveals that the **table( )** function is a wrapper for another; the **tabulate( )** function. More importantly, the code reveals the structure of a **"table"** object is simply an ***array*** created from the ***counts***.

The **names( )** functions prints the components in an object; the **attributes( )** function provides more detail.

The **exists( )** function returns a logical value TRUE or FALSE, indicating the argument object's existence.

```
> exists("ct")
[1] TRUE

> exists("ctu")
[1] TRUE
```

This function is particularly useful when developing functions, loading packages, and running object-dependent logical statements. For example, if general-purpose code is being written, the code may need to determine the existence of certain objects necessary to proper execution; if the object does not exist, then the code will have to create it.

Additionally, objects can be saved to disk through the **save( )** function and reloaded through the **load( )** function; the latter general-code might load previously saved code if the specified object does not exist.