Object Oriented Programming

- Object-oriented programming concepts are in many programming languages
- R has a different model than C++/Java/Python/JavaScript
  - But still two fundamental concepts in common
    - polymorphism/generic functions
    - inheritance

R's OOP

- R has 2 OOP models
  - S3 – older, simpler, more dynamic, less structured version
  - S4 – newer, more structured, more powerful
- S3 is the basis for most of the modeling software in R
  - so important to know to understand how R works
  - powerful enough to produce good, widely used software.
  - easy enough to explain in undergrad class.

S3 OOP

- Every object has a type and also a class, sometimes the same
- The type is the fundamental data structure
- The class is a character string or vector that is a label giving a more abstract notion of what the object represents, e.g. a linear model, a generalized linear model, a description of a package, a grid of moving cars
- The classes are R programmer-made labels, not necessarily built into the core R language

Linear & Generalized Linear Models

- As an example, let's look at linear models
  - glm.mtcars = glm(mpg ~ ., data = mtcars)
- class(glm.mtcars)
  - This gives c("glm", "lm") which indicates that it is an object of class glm first, but also "lm"
- Let's fit just a linear model
  - lm.mtcars = lm(mpg ~ wt + cyl, mtcars)
- This has class just "lm" (since it just a linear model and not a generalized linear model)

Generic Functions

- One key purpose of OOP in R is to allow users call a function such as plot() or summary() and to have the function behave differently and appropriately for the given argument(s)
- In other words, plot() or summary() is a generic verb that has methods for different types of inputs.

Generic dispatch

- How might we implement this?
  - We could define the plot() function as a large sequence of if() statements that said what to do for each type of class, e.g.
    - plot = function(x, y, …) {
      if(is(x, "glm"))
      return(callPlotCodeForGLM(x, y, …))
      else if(is(x, "lm"))
      return(callPlotCodeForLM(x, y, …))
      else if(is(x, "matrix"))
      return(callPlotCodeForMatrix(x, y, …))
      else # and so on for all classes we know about.

- This clearly is tedious, error-prone and very limited
  - it can only take into account the classes we currently know about
  - not those that others introduce in the future
- To add support for a new class, I would have to define a new version of plot and my code.
  - I could copy the original one and add my own if(is(x, "myClass")) …. somewhere
  - or just handle the myClass case and then call the original one, e.g. using graphics::plot(x, y, …), i.e. delegate back to the original one.
- The latter approach is best as if the graphics::plot() function is changed, I will still get those changes, and not the version I copied.
- Also, do I add my if(is(x, "myClass")) at the top or bottom of the sequence of if() tests for class?
  - the order matters!

Support for two or more new classes

- What if two packages want to add support for two or more classes?
  - They would both need to make a new plot() function and add if(is(x, "class-name")) … to their new version
  - But only one would be called and that would depend on the order the packages were loaded and added to the search path
  - This makes for very volatile, non-robust code!

A better way – S3 dispatch

- R provides a much better approach than the sequence of if() statements and making copies of functions.
- We define a generic function such as plot() as follows:
  plot = function(x, y, …)
  UseMethod("plot")
- The body calls UseMethod() with the name of the generic.
- This means that when the generic plot() function is actually called (not defined), it will look for the appopriate method based on the type of x.
- (It can use a different argument, e.g. y, but typically we "dispatch" on the type of the first argument.)

Finding methods

- So how does UseMethod() find the appropriate method for a given class
- We use a very simple convention:
  - find a function named
    <generic name>.<class name>
    e.g. plot.glm or plot.lm or plot.function
- Algorithmically, you can think of it this way:
  - find(paste("plot", c(class(x), "default"), sep = "."))
  - i.e. paste the generic name to each of the elements in the class vector, including the word "default" and find functions with these names
    - use the first of these, or if none exists, raise an error
    - by adding default to the names of the classes, we can have a catch-all case, which may or may not make sense for the particular generic.

Defining a method

- So suppose we introduce a new class, say "Normal" to represent a Normal distribution.
- dist = c(mean = 4, sd = 5)
  class(dist) = "Normal"
- (or structure(c(mean = 4, sd = 5), class = "Normal")))
- So we haven't formally defined a class "Normal", but just put a label on the object dist to identify it as an object of class "Normal"

- To be able to allow users to draw the density of the particular "Normal" object, we can define a method for plot() as
  - plot.Normal =

```
function(x, y, ...) {
  lims = x["mean"] + c(- 3* x["sd"],  3* x["sd"])
  h = seq(lims[1], lims[2], length = 1000)
  plot(h, dnorm(h, x["mean"], x["sd"]), type = "l", …)
}
```

- Now when we call
  plot(dist)
  the generic plot() function will look for plot.Normal and use our new method.
- We have not changed any code, just added a new class and method.
- So we haven't broken code or needed to retest the original.
- And others can define new classes in exactly the same, isolated manner, e.g. Exponential, Poisson, Cauchy, or for totally different types of objects.

Inheritance

- The second important benefit of OOP is inheritance.
- Let's look at our glm object again
  - It has a class vector of c("glm", "lm"), i.e. two labels.
- plot(glm.mtcars) causes the generic to look for a method named plot.glm
  - there is none (check)
  - so it uses plot.lm which does exist.
- Our glm object inherits the methods for the "parent" classes when there is not a method that is specific to the glm class.
- A glm object is an lm object, but we can override or specialize certain methods.

Methods that use the inherited methods

- We might define a method for plot for a glm class that does something different from an lm object.
  - we can do this by defining a function plot.glm()
- In many cases, such a method might want to do some pre-computations and then call the method inherited from the lm class.
- To do this, we would write our method something like

```
plot.glm = function(x, y, …) {
        # here we do some precomputations
      cat("in plot.glm\n")
      NextMethod("plot") # call the inherited method from lm.
  }
```

- Here we are passing control back to the next method that would have been called by the generic if plot.glm had not been defined.

- We could have written the previous method as

```
plot.glm = function(x, y, …) {
        cat("in plot.glm\n")
        plot.lm(x, y, …)
    }
```

- This will appear to do the same thing in most cases, but it is quite different
  - it says always call plot.lm
  - but what if our glm object 'x' had a class c("glm", "OtherClass", "lm")
  - NextMethod() would have called the plot.OtherClass() method, not plot.lm()
  - So better to use NextMethod() as it respects the actual class hierarchy on the object at run-time rather than when the plot.glm function was written.

S4 classes & methods

- (see the file S4.R)

Issues with S3

- S3 is very powerful and convenient
- However,
  - there is no formal structure to the class definitions and I can label the value 3 as a glm object

    x = 3

    class(x) = "glm"
    - So there are no checks on the contents of the objects of a particular class, or consistency about the class vector (e.g. missing the "lm" here as a parent class.)
    - Without a formal structure, relying on programmers to be consistent amongst each other and across time!!!!

Dispatching on more than one argument

- Also, what happens if we want to write a method for the case

  x is a glm

  y is a numeric vector

  and another case for

  x is an lm

  y is a matrix
- Since an S3 generic only looks at the class of a single parameter, we have to go back to a sequence of if() statements within each of the methods – bad news!

S4

- The S4 class & method system addresses both issues
  - formal definitions of classes
  - validation that the contents of an object of a class are appropriate
  - define methods for combinations of types of one or more parameters.

S4 Example - Shapes

- Let's define a set of classes representing shapes, i.e. circles, squares, rectangles, triangles, octagons, polygons, …
- We'll start with a class representing the basic concept of a shape, but ensure that users cannot create an instance of this "abstract" class, but only the "concrete" or real classes.
  - The Shape class is a virtual class – no instances can be created.
- setClass("Shape", representation("VIRTUAL"))

Circle

- We define a Circle class as a sub-class of Shape (via contains = "Shape") and specify that it has a single slot named "radius"
- We also want new instances of Circle to have a radius of 1 unit, so we specify a prototype.
- setClass("Circle",
  representation(radius = "numeric"),
  contains = "Shape",
  prototype = list(radius = 1))
- Now we can create instances of the class Circle
  - cr = new("Circle")
    cr2 = new("Circle", radius = 10)

Accessing slots in S4 objects

- While you might think of S4 objects as a list containing the values of the slots, this is not the case
- You cannot use cr$radius
- Instead, we use @ to access slots
  cr@radius
  cr@radius = 20
- This ensures that we correctly
  - access slots inherited from parent/ancestor classes
  - and validate assignments to the slots so that the right hand side are consistent with the definition of the type of the slot.

Finding the names of slots

- It is hard to remember the set of slots in all of the classes one is working with.
- We use slotNames() to find these on an object or class
  - slotNames(cr)
    slotNames("Circle")
- We can find the definition of a class, i.e the slots, their types, the ancestor classes, etc. with
    getClass("Circle")
- This ability to query the class & definition of an object is often called "reflection".

Validity methods

- Of course, a circle cannot have a negative radius, and nor can it have two or more radii or an empty numeric vector as a radius.
- Our class definition hasn't protected us against a user doing
    new("Circle", radius = c(-10, 100))
- So we want to specify a validity method that will be used to ensure that the contents of a Circle object are legitimate, both individually and collectively (i.e. that all the potential slots are jointly meaningful)

Validity method

- So we write a function that takes a potential Circle object and then perform the relevant checks.
- If it is okay, we return TRUE; otherwise we return a string describing the problem
- We use setValidity() to associate the validity function with the class for which it is intended
- setValidity("Circle",

```
  function(object) {
    r = object@radius
    if(length(r) == 1 && r >= 0)
      TRUE
    else if(length(r) == 0)
      "no radius in Circle"
    else if(length(r) > 1)
      "more than one radius in Circle"
    else if(r < 0)
      "negative radius in Circle"
  })
```

Inherited slots

- We defined Shape as being a virtual class with no slots.
- However, we might have added slots such as border color and interior color, border width, etc.
- The class would still be VIRTUAL, but all sub-classes would then inherit the set of slots from Shape.
- We access the inherited slots in the same way as regular slots, i.e. cr@borderWidth

Methods

- Let's define a generic function area() and define methods for it to act on different classes of Shape objects.
- We define a generic function in S4 via setGeneric()
- setGeneric("area",
  function(obj, …)
  standardGeneric("area"))
- We give the name of the generic function and the skeleton function implementing the generic.
- That skeleton calls standardGeneric() with the name of the generic and this is analogous to UseMethod(), i.e. when the generic is called, it will find the appropriate method given the actual arguments.

… in the signature of a generic

- It is a good idea to specify a signature for the generic that is most representative of the methods that we will define for it, i.e.
  - if there will usually be 2 arguments, explicitly specify this function(x, y, …)
- Add … at the end (if not already present) so that methods can explicitly add their own named parameters.

Defining methods

- Instead of using the simple naming convention of generic.className to identify a method, we use setMethod() to define and register a method in S4.
- We specify
  - the name of the generic for which the method is being defined, e.g. "area"
  - the class signature for which it is to be used, i.e. the parameters and their corresponding classes for which the method applies
  - and the actual function implementing the method.

Method definition

- setMethod("area", "Circle",
  function(obj, …)
  pi * obj@radius^2)
- Now when we call
  area(cr)
  our method will be found

area for Ellipse

- To define an area method for an Ellipse, we might use

```
setMethod("area", "Ellipse",
          function(obj, ...)
              pi * prod(obj@radius))
```

Inheritance & 'is a'

- A Circle is a special type of Ellipse, i.e. one with the two radii constrained to be equal.
- We might use this to define a Circle as
  setClass("Circle", contains = "Ellipse")
- We could define a validity method that ensured the Circle had a radius slot of length 2 and that the elements were the same.
- We could then inherit the area() method, and most likely other methods too.
- This reduces the code we need to write and provide – always a good thing, at least initially!

- Drawing a circle might be faster using graphical primitives for circles rather than those for ellipses.
- In this case, we might want to override the method for draw() for an Ellipse to provide a specialized version for the sub-class Circle,
  - setMethod("draw", "Circle",
    function(x, at, …)
    drawCircle(at[1], at[2],
    radius = x@radius[1]))

- The choice to have sibling classes or sub-classes can be difficult and take both experience and careful consideration of how the classes will be used.
- There is a tradeoff between reducing the amount of code via inheriting methods, having to provide specialized methods for too many methods, and having less efficient representation of the object, i.e. using a vector of length 2 when we only need one value for the radius of a Circle.

- We might even go as far as to use a virtual class 2DShape which maintains a vector of two giving the length dimensions of the object.
  We could use this for both Ellipses and Rectangles and use an IsoLength class to get Circle and Square from these.
- The complexity and indirectness is probably not worth it unless we want to exploit the common ancestor class extensively.

setClassUnion

- If we add characteristics such as color and border width to the Shape class, we have to think about representing color.
- Colors can be specified in various different ways
  - by name, e.g., "red", "pink",
  - by an #RGB string, e.g. #FF0000,
  - or by a vector of values c(R, G, B) for the Red, Green and Blue components.
  - and we might even have an alpha level for a color.
- If we define a slot color in our Shape class, we have to specify a type and we don't seem to be able to conveniently accommodate all of these different types in a single slot.

- We want a general class Color that can accommodate all of these, i.e. that can be any of a name string, a RGB string, or a numeric vector
- setClassUnion() is used for this, to define a new class that can have values from any of the specified other classes.

- setClass("ColorName", contains = "character")
- setClass("RGBString", contains = "character")
- setClass("RGBColor", contains = "numeric")
- setClassUnion("Color",
  c("ColorName", "RGBString", "RGBColor"))
- Now, to specify the color in a Shape object, we might use
  new("Circle",
  borderColor = new("ColorName", "red"))

Helper functions

- To avoid having to use new("ColorName", …) and new("RGBString", …), we probably would want to write an R function that converted a string to the appropriate Color object, i.e. ColorName or RGBString
- The function would examine the string and figure out if it was in the RGBString format or not.

- Note also that we might not want to bother with the RGBString class in the Color definition.
- We can always process that string and convert it to an RGBColor object
- This is where we would like to have a method to coerce from RGBString to RGBColor.
- We also want to have helper/constructor functions that create Colors and Shape objects and perform the relevant coercions to the expected types.

setAs()

- If we define Circle and Ellipse as separate, unrelated classes both derived from Shape, we might want to define a method that turns a Circle into an Ellipse
  - such an object would have the 2 radii equal to the radius of the circle
  - We can define a coercion method from one class to another using setAs()
  - setAs("Circle", "Ellipse",
    function(from)
    new("Ellipse", radius = rep(from@radius, 2)))

coercion from RGBString to RGBColor

- We can use col2rgb to do this
- setAs("RGBString", "RGBColor"
  function(from)
  new("RGBColor",
  as.numeric(col2rgb(from))))

Packages & OOP

- When we put our code into an R package (a good thing), there are some issues to be aware of relating to OOP
- If you don't use a NAMESPACE file (booo!), things are pretty straightforward and work as one might expect.
- If you do use a NAMESPACE (congratulations!), you have to export methods and S4 classes.

Exporting S3

- When we want specific S3 methods to be available outside of a package, we export them via the NAMESPACE file with the directive S3method() call
- This takes the name of the generic and the S3 class
  - S3method(plot, myClass)

Exporting S4 classes & methods