Week 10 Lecture:
Programming Paradigms, Introduction to Object Oriented Programming

Introduction to Programming and Geoprocessing Using R
GEO6938-4172
GEO4938-4166

Programming Paradigms

• Procedural Programming
  – Also known as “imperative”

• Functional Programming
  – Similar to procedural, but no reliance on “shared state”

• Object Oriented Programming
  – Everything is an object
Programming Paradigms

• Object Oriented Programming (OOP)
  – Everything is an object
  – Examples:
    • Primarily OO: Java, C# (.NET)
    • Can be use as OO: C++, Python, Ruby, ... R
  – Objects have:
    • Properties
    • Methods
    • Inheritance

It’s all about nouns... and verbs...

... but really about the nouns, and *abstraction*, *polymorphism*, and *inheritance*.

<table>
<thead>
<tr>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Color</td>
</tr>
<tr>
<td>-VIN</td>
</tr>
<tr>
<td>-...</td>
</tr>
<tr>
<td>+steer()</td>
</tr>
</tbody>
</table>
Abstraction...

... and inheritance via “Is-A” relationships

Unified Modeling Language (UML) Class Diagram

“Has-A” Relationships
Inheritance From Parent/Super Class

UML Specifics: Two Types of Has-A Relationships

Aggregation:
UML Specifics: Two Types of Has-A Relationships

Composition:

Object Oriented-ness

“Object Oriented Programming puts the Nouns first and foremost. Why would you go to such lengths to put one part of speech on a pedestal? Why should one kind of concept take precedence over another? It's not as if OOP has suddenly made verbs less important in the way we actually think. It's a strangely skewed perspective.”

-- Steve Yegge, 2006
Why use OOP at all?

In R, OOP is Optional...

- But exists behind even the most basic functionality
- Implemented through the use of generic functions
Basic Concepts and Vocabulary in R

- **Class**: A description of a thing
- **Object**: An instance of a class
- **Generic function**:
  - A function which dispatches methods
  - Encapsulates a generic concept (e.g. `print()`, `summary()`, `predict()`)
  - Does not actually do anything
- **Method**:
  - Implementation of a generic function for an object of a particular class

S3 vs. S4 Systems of Objects in R

- The S language (and therefore R) has two systems
  - S3 and S4, may be used independently
- S3 objects, classes and methods have been available in R since 1992
  - More informal and interactive
- S4 system is newer (1998), more formal and rigorous from an OOP standpoint
  - Provided by the `methods` package (part of base since 1.7.0)
S3 vs. S4 generic functions

> summary   # - S3 generic function
function (x, ...)
UseMethod("summary")
<environment: namespace:base>

> show      # - S4 generic function
standardGeneric for "show" defined from package "methods"
function (object)
standardGeneric("show")
<environment: 0x8d7cdc8>
Methods may be defined for arguments: object

Following are based on an OOP in R lecture by David Perlman, 2009

The summary() function:

> summary
function (object, ...)
UseMethod("summary")
<environment: namespace:base>

Defined as:

> summary <- function (object, ...)
  UseMethod("summary")
How does it work?

```r
> x <- as.factor(rep(c("a","b"),c(7,13)))
> class(x)
[1] "factor"
> summary(x)
 a  b
 7 13

y <- rnorm(20)
> class(y)
[1] "numeric"
> summary(y)
       Min.  1st Qu.   Median     Mean   3rd Qu.     Max.
       -1.78300 -0.55330  0.06459  0.01969  0.66960  1.72900

How does it work?

```r
call <- lm(y~x)
class(call)
[1] "lm"
> summary(call)
Call:
  lm(formula = y ~ x)
Residuals:
     Min      1Q  Median      3Q     Max
-1.7500 -0.6665 -0.1008  0.5537  1.5156
Coefficients:
                     Estimate Std. Error t value Pr(>|t|)
(Intercept)         -0.3769     0.3609 -1.044  0.310
xb                  0.6102     0.4477   1.363  0.190
Residual standard error: 0.9549 on 18 degrees
of freedom
Multiple R-squared: 0.09355,  Adjusted R-squared: 0.04319
F-statistic: 1.858 on 1 and 18 DF,  p-value:
 0.1897
```
Method Dispatch: summary()

> methods(summary)

[1] summary.aov            summary.aovlist summary.aspell*
[4] summary.connection     summary.data.frame summary.Date
[7] summary.default        summary.ecdf* summary.factor
[10] summary.glm            summary.infl summary.lm
[16] summary.mlm            summary.nlm* summary.packageStatus*
[19] summary.POSIXct         summary.POSIXlt summary.ppr*
[22] summary.prcomp*        summary.princomp* summary.rxCrossTabs*
[25] summary.rxCube*        summary.rxLinMod* summary.rxLogit*
[28] summary.shingle*       summary.srcfile summary.srcref
[31] summary.stepfun        summary.stl* summary.table
[34] summary.trellis*       summary.tukeysmooth*

Non-visible functions are asterisked

Method: summary.factor()

> summary.factor

function (object, maxsum = 100, ...)
{
    nas <- is.na(object)
    ll <- levels(object)
    if (any(nas))
        ....
    }
    if (any(nas))
    c(tt, "NA's" = sum(nas))
    else tt
}

<environment: namespace:base>
Why bother with OOP in statistical programming?

Calculate mean and sd for binomial data:

\[ E(x) = np; \ Var(x) = np(1-p) \]

```r
stats <- function(x) {
  n = length(x)
  p = mean(x)
  mu = n*p
  sigma = sqrt(n*p*(1-p))
  return( list(mu = mu, sigma = sigma, n = n) )
}
```

> y1 <- rbinom(100, size=1, p=.3)
> stats(y1)

$mu
[1] 33

$sigma
[1] 4.702127

$n
[1] 100

But what if we use it with the wrong type of distribution?

```r
> y2<-rnorm(100,mean=0.3,sd=1)
> stats(y2)

$mu
[1] 42.52809

$sigma
[1] 4.943855

$n
[1] 100
```

True mean and standard deviation:

> mean(y2)
[1] 0.4252809

> sd(y2)
[1] 1.005184
How do we prevent that from happening?

- We can use OOP of course...
  - At creation, we can define a class for our object
  - We can then create a generic stats() function
  - This stats() function dispatches functionality to appropriate methods based on the class of the object

Define a class constructor (S3)

```r
> as.binomial <- function(x) {
+   class(x) <- "binomial"
+   return( x )
+ }
> y1 <- as.binomial(y1)
> class(y1)
[1] "binomial"
> y2 <- as.normal(y2)
> class(y2)
[1] "normal"
```
Generic Function and Methods

stats <- function(x) UseMethod("stats")

stats.binomial <- function(x) {
  n = length(x)
  p = mean(x)
  mu = n*p
  sigma = sqrt(n*p*(1-p))
  return( list(mu = mu, sigma = sigma, n = n) )
}

stats.normal <- function(x) {
  n = length(x)
  mu = mean(x)
  sigma = sd(x)
  return( list(mu = mu, sigma = sigma, n = n) )
}

stats.default <- function(x) {
  stop("Unknown data type!")
}

13
Using Our `stats()` Function

- With a known class:
  ```r
  > stats(y1) > stats(y2)
  $mu
  [1] 33
  [1] 0.4252809
  $sigma
  [1] 4.702127
  [1] 1.005184
  $n
  [1] 100
  [1] 100
  ```

- With an unknown class:
  ```r
  > y3 <- rnorm(100, mean=2, sd=1)
  > stats(y3)
  Error in stats.default(y3) : Unknown data type!
  ```

Inheritance in R

- A way to form new classes using classes that have already been defined.

- By the mechanism of inheritance, a new class (known as a derived class) takes over or inherits attributes and behavior of pre-existing classes (referred to as base classes or ancestor classes)
Creating a Standard Normal Class

```r
as.standardNormal <- function(x){
  x <- as.normal(x)
  class(x) <- c("standardNormal", class(x))
  return( x )
}

stats.standardNormal <- function(x){
  object <- stats.normal(x)
  object$sigma <- 1
  return(object)
}
```

Inheriting Methods

```r
> y4 <- rnorm(100)
> y4 <- as.standardNormal(y4)
> stats(y4)
$mu
[1] 0.03292102
$sigma
[1] 1
$n
[1] 100
> class(y4)
[1] "standardNormal" "normal"
```
This Week in Lab:

- Work week for your class project development
- I’ll post project guidelines and assignment on Sakai
- Assignment will be to turn in a short (less than one page) synopsis of what you intend to do

- I won’t be there!
- Not required to attend