# Object Oriented Programming

*Dom Owens*

*22/10/2019*

R offers three different **object-oriented programming (OOP)** models:

- S3
- S4
- Reference Classes

The three OOP models differ by the degree of encapsulation they offer. S3 offers functional OOP; S4 requires further definitions; Reference classes are mutable, meaning they can be modified in place.

OOP allows *polymorphism*, the provision of a single interface to different types. This allows us to reuse and extend ideas, as well as to separate a piece of software from its' implementation. Here, implemenatation is *encapsulated* in an object, so that the data and functionality are in the same place. Objects have *types*, and these are specified in the *class* definition. The state is in the *field* and the behaviour is in the *methods*.

## S3

**S3** is the only OO system used in the `base` and `stats` packages, and has no formal class definition. S3 objects have a base type and an attribute `class` set to the class name; we simply append a class name to an object to create one. **Constructor** functions initialise an object then return the object afterwards (consider `lm`). Validation should be performed in a separate function, and help provided by another.

We can test whether an object is an S3 object with `is.object(x) & !isS4(x)`, i.e. is this an object, and is it not S4-type.

Here is an implementation of a regression function with two regressors in S3:

```r
# set seed for reproducibility
set.seed(123)

# define sample size, coefficients, regressor and response
n  <- 50
b0 <- matrix(c(2, -1, 1), ncol = 1)
x1  <- rnorm(n)
x2  <- runif(n)
x <- cbind(x1,x2)
y  <- b0[1] + b0[2] * x1 +  b0[3] * x2 +  rnorm(n)

# Constructor for `simple_lin_regression`.
#
# This function computes the ordinary least squares estimate
# for the simple linear regression E[y|x] = b0 + b1 * x1 + b2 * x2.
#
# Params: y - a vector with n elements; observed responses
#         x - a matrix with n rows and p=2 columns; observed regressors
#
# Returns: an S3 object of type simple_lin_regression
#
simple_lin_regression <- function(y, x) {
```

```
  # define design matrix
  n <- length(y)
  D  <- matrix(c(rep(1, n), x), ncol = 3)

  # compute the OLS estimate
  b  <- solve(t(D) %*% D) %*% t(D) %*% y

  # the object will encapsulate y, X and b
  slr <- list(response = y, regressor = x, estimate = b)

  # the object `slr` is declared to be an S3 object of
  # class "simple_lin_regression", by
  class(slr) <- "simple_lin_regression"

  return(slr)
}

sl1 <- simple_lin_regression(y, x)
```

Now we can use generic methods on our object. Methods are defined according to `method_name.class_name()`, like `plot.lm()`.

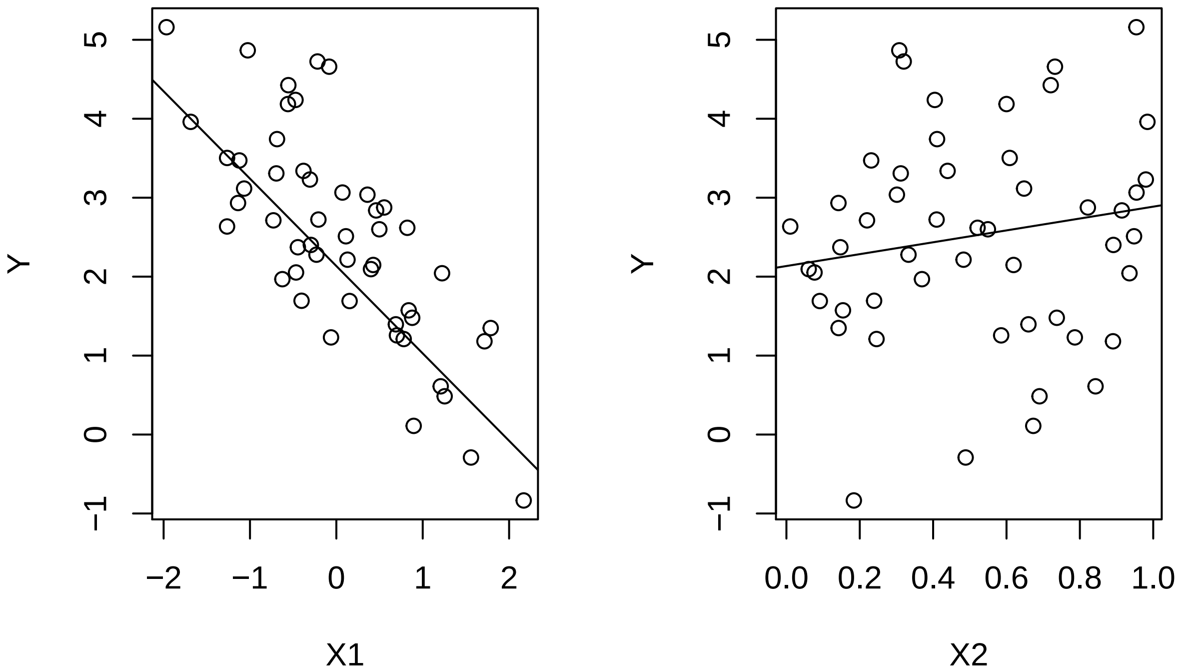
```
print.simple_lin_regression <- function(x, ...) { #print method
  cat("head(x) =", head(x$regressor), "\n") #head of x
  cat("head(y) =", head(x$response), "\n\n") #head of y
  cat("Estimated regression: E[y|x] = b0 + b1 * x1 + b2* x2\n") #model formula
  cat("b0 = ", x$estimate[1], " b1 = ", x$estimate[2],  " and b2 = ", x$estimate[3], ".\n", sep = "")
}

plot.simple_lin_regression <- function(x, y = NULL, ...) { #plot method
  par(mfrow = c(1,2)) #two panes
  plot(x = x$regressor[,1], y = x$response,
       xlab = expression(X1), ylab = expression(Y)) #plot y against x1
  abline(a = x$estimate[1], b = x$estimate[2])
  plot(x = x$regressor[,2], y = x$response,
       xlab = expression(X2), ylab = expression(Y))  #plot y against x2
  abline(a = x$estimate[1], b = x$estimate[3])
}
```

Then test:

```
print(sl1)
```

```
## head(x) = -0.5604756 -0.2301775 1.558708 0.07050839 0.1292877 1.715065 0.599989 0.3328235 0.488613 0
## head(y) = 4.186036 2.278228 -0.290813 3.065269 2.214723 1.181049
##
## Estimated regression: E[y|x] = b0 + b1 * x1 + b2* x2
## b0 = 2.134247 b1 = -1.106688 and b2 = 0.7522399.
```

```
plot(sl1)
```

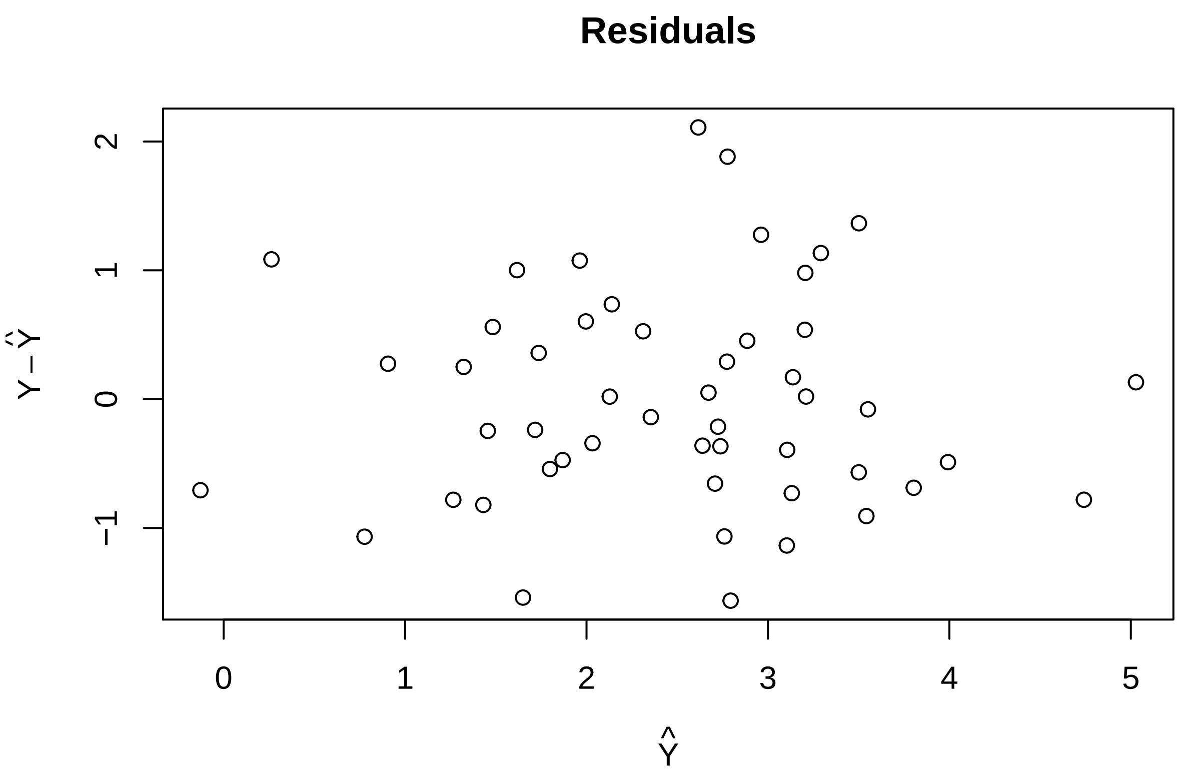



**Generic functions** are called according to `UseMethod("name_of_generic_function")`. Let's add a residual analysis function, and add it for our class:

```
residual_analysis <- function(x) {
  UseMethod("residual_analysis")
}
residual_analysis.simple_lin_regression <- function(x) {
  predictions <- x$estimate[1] + x$estimate[2] * x$regressor[,1] + x$estimate[3] * x$regressor[,2] # pr
  plot(x = predictions, y = x$response - predictions,
       xlab = expression(hat(Y)), ylab = expression(Y - hat(Y)), main = "Residuals") #plot residuals
}
```

Test it:

```
residual_analysis(sl1)
```




**Inheritance** is managed by setting the `class` attribute to a vector of class names. We demonstrate how this is transferred between objects below, using the `print` method for different, related generations.

```
# define objects of class grandparent, parent and child, respectively.
G <- structure(numeric(), class = "grandparent_class")
P <- structure(numeric(), class = c("parent_class", "grandparent_class"))
C <- structure(numeric(), class = c("child_class", "parent_class","grandparent_class"))

# now implement print, first for grandparent_class
print.grandparent_class <- function(x) {
  cat("this is print for grandparent_class objects")
}

print(G)
```

```
## this is print for grandparent_class objects
```

```
print(P)
```

```
## this is print for grandparent_class objects
```

```
print(C)
```

```
## this is print for grandparent_class objects
```

```
# now implement print for parent_class
print.parent_class <- function(x) {
  cat("this is print for parent_class objects")
}
print(G)
```

```
## this is print for grandparent_class objects
```

```r
print(P)
```

```
## this is print for parent_class objects
```

```r
print(C)
```

```
## this is print for parent_class objects
```

### S4

In S4, all elements have to be defined explicitly. We use `setClass` to create a class definition.

```r
library(methods)
setClass("simple_lin_regression",
   slots = c(
     response  = "numeric",
     regressor = "matrix",
     estimate  = "numeric"
     )
)
```

We initiate an object with `new`, usually wrapped in a **constructor** class.

```r
# Constructor for `simple_lin_regression`.
#
# This function computes the ordinary least squares estimate
# for the simple bivariate linear regression E[y|x] = b0 + b1 * x` + b2 * x2.
#
# Params: y - a vector with n elements; observed responses
#         x - a matrix with n rows and p=2 columns; observed regressors
#
# Returns: an S4 object of type simple_lin_regression
simple_lin_regression <- function(y, x) {
  slr <- new("simple_lin_regression",
             response  = y,
             regressor = x
             )
  return(slr)
}
```

Then, for the coefficients, we perform initialisation

```r
setMethod("initialize", "simple_lin_regression",
  function(.Object, response, regressor) {
    .Object@response  <- response
    .Object@regressor <- regressor

    # define design matrix
    n <- length(response)
    D  <- matrix(c(rep(1, n), regressor), ncol = 3)

    # compute the OLS estimate
    b  <- solve(t(D) %*% D) %*% t(D) %*% response
    .Object@estimate <- as.numeric(b)
    return(.Object)
```

```
  }
)
```

```
## [1] "initialize"
```

Then provide the `print` and `plot` methods

```
setMethod("print", "simple_lin_regression", #set print method
  function(x, ...) {
    cat("head(x) =", head(x@regressor), "\n")
    cat("head(y) =", head(x@response), "\n\n")
    cat("Estimated regression: E[y|x] = b0 + b1 * x1 + b2 * x2\n")
    cat("b0 = ", x@estimate[1], " b1 = ", x@estimate[2], " and b2 = ",  x@estimate[3], ".\n", sep = "")
  }
)
```

```
## [1] "print"
```

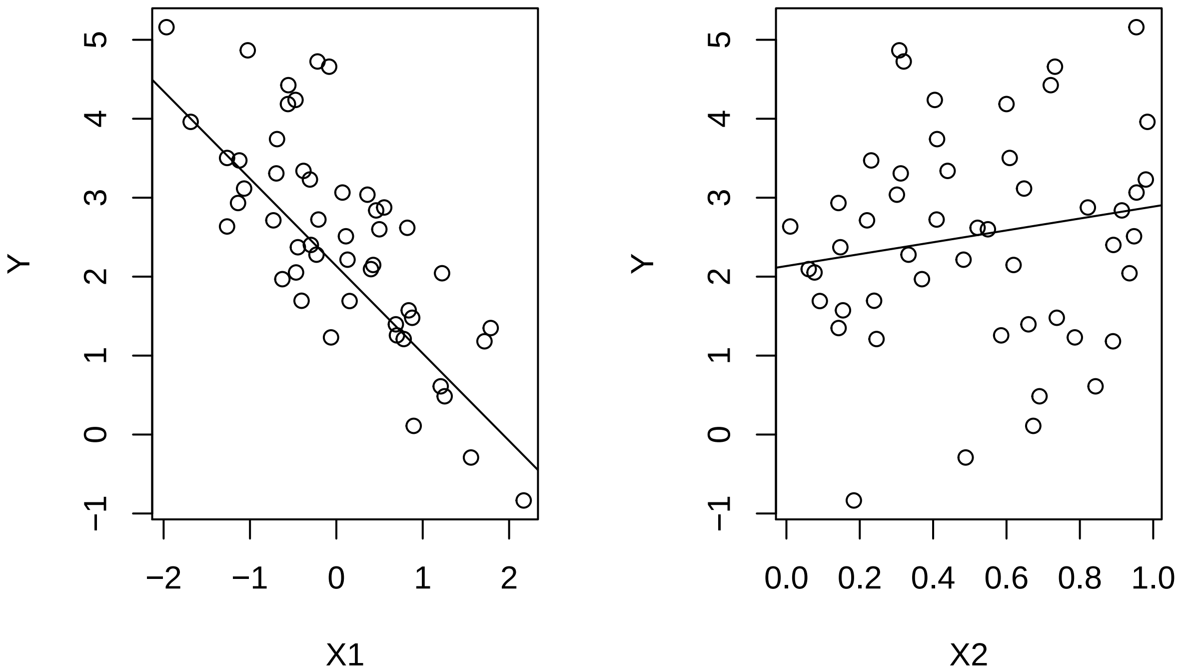
```
setMethod("plot", "simple_lin_regression", #set plot method
  function(x, y = NULL, ...) {
    par(mfrow = c(1,2))
  plot(x = x@regressor[,1], y = x@response,
       xlab = expression(X1), ylab = expression(Y))
  abline(a = x@estimate[1], b = x@estimate[2])
  plot(x = x@regressor[,2], y = x@response,
       xlab = expression(X2), ylab = expression(Y))
  abline(a = x@estimate[1], b = x@estimate[3])
  }
)
```

```
## [1] "plot"
```

```
slm1 <- simple_lin_regression(y, x)
print(slm1)
```

```
## head(x) = -0.5604756 -0.2301775 1.558708 0.07050839 0.1292877 1.715065 0.599989 0.3328235 0.488613 0
## head(y) = 4.186036 2.278228 -0.290813 3.065269 2.214723 1.181049
##
## Estimated regression: E[y|x] = b0 + b1 * x1 + b2 * x2
## b0 = 2.134247 b1 = -1.106688 and b2 = 0.7522399.
```

```
plot(slm1)
```




We can ensure *validity* using the `setValidity` method:

```
setValidity("simple_lin_regression",
            function(object) {
              if(length(object@response) == length(object@regressor)) TRUE
              else paste("Unequal lengths of regressor and response.")
            })
```

```
## Class "simple_lin_regression" [in ".GlobalEnv"]
##
## Slots:
##
## Name:   response regressor  estimate
## Class:   numeric    matrix   numeric
```

We define **getter** and **setter** methods (like `residuals(lm)`):

```
setGeneric("regressor", function(x) standardGeneric("regressor")) #getter
```

```
## [1] "regressor"
```

```
setGeneric("regressor<-", function(x, value) standardGeneric("regressor<-")) #setter
```

```
## [1] "regressor<-"
```

```
setMethod("regressor", "simple_lin_regression", function(x) x@regressor)
```

```
## [1] "regressor"
```

```
setMethod("regressor<-", "simple_lin_regression",
  function(x, value) {
    x@regressor <- value
    x <- initialize(x, response = x@response, regressor = value)
    validObject(x)
    return(x)
  }
```

```
)
```

```
## [1] "regressor<-"
```

```
setGeneric("estimate", function(x) standardGeneric("estimate"))
```

```
## [1] "estimate"
```

```
setMethod("estimate", "simple_lin_regression", function(x) x@estimate)
```

```
## [1] "estimate"
```

```
estimate(slm1) #get estimates - equivalent to slm1$estimates
```

```
## [1]  2.1342467 -1.1066876  0.7522399
```

### Reference classes (RC)

The third option for OO in R is **Reference Classes**. Here, methods are implemented as part of the class definition and belong to objects, not functions. "modify-in-place" semantics can be used.

We use a different example to illustrate this. The class `DataContainer` has the field `data` containing numbers. When the getter method for `results` is used, computation is performed. `flag` is used to denote whether new data has been supplied.

```
library(methods)

DataContainer <- setRefClass( #set class
  "DataContainer",
  fields = c(
    data   = "numeric",
    flag   = "logical",
    result = "numeric"
  )
)

dataContainer <- function(data) {
  dC <- DataContainer$new(data = data)
  return(dC)
}

DataContainer$methods(
  initialize = function(data) {
    .self$data <- data
    .self$flag <- FALSE
  },
  doComputation = function() { #perform calculation
    cat("MSG: doing computation now!\n")
    .self$result <- mean(.self$data) #calculate mean
    .self$flag   <- TRUE
  },
  show = function() {
    cat("head(data) = ", head(data), "\n", sep = "")
    cat("result     = ", result, "\n", sep = "")
  },
  getResult = function() {
```

```r
    if (!flag) {
      .self$doComputation()
    }
    return(result)
  },
  setData = function(value) {
    .self$initialize(data = value)
  }
)
```

```r
dC1 <- dataContainer(rnorm(5)) #generate data container with 5 standard normal samples
dC1 #no result
```

```
## head(data) = -0.65194990.23538660.07796085-0.9618566-0.07130809
## result     =
```

```r
dC1$getResult() #get mean
```

```
## MSG: doing computation now!
```

```
## [1] -0.2743534
```

```r
dC1 #print object with result
```

```
## head(data) = -0.65194990.23538660.07796085-0.9618566-0.07130809
## result     = -0.2743534
```

## Picking a system

How should we pick which of the three OOP systems to use?

S3 is well suited to creating simple objects and adding methods for pre-existing functions like `plot()`. This can be achieved with minimal code, making it suitable for straightforward statistical package development.

Complicated systems with interrelated objects might require S4. Consider the `Matrix` package, which stores and computes different types of sparse matrices.

RC is well-suited for programmers coming from an OOP background in other languages. The side effects of modify-in-place functionality can be unintuitive.

**This work refers to http://adv-r.had.co.nz/OO-essentials.html**