University of Glasgow

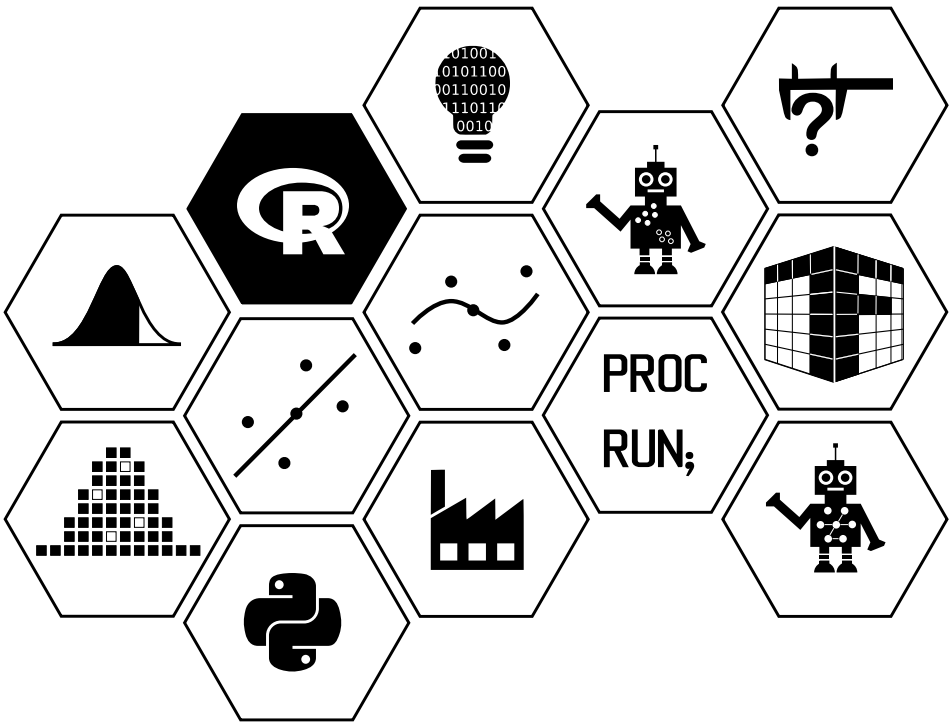
R Programming/ Statistical Computing

Craig Alexander

Academic Year 2023-24

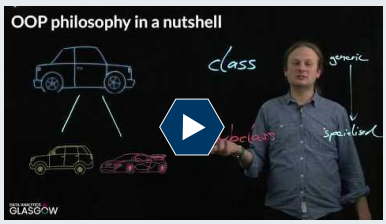
Week 9:

Object oriented programming in R



DATA ANALYTICS GLASGOW

A brief introduction to object-oriented programming



Object-oriented programming

https://youtu.be/zXMZilqO_4o

Duration: 7m43s

What you have seen so far

You have almost certainly already used aspects of object orientation in R, possibly without even realising it. The functions `summary` and `plot` appear to magically do the "right thing" for different types for arguments.

For a linear regression model `summary` shows different output …

```
model <- lm(dist~speed, data=cars)
summary(model)
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601   0.0123 *
## speed         3.9324     0.4155   9.464 1.49e-12 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

… than for a data frame …

```
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length
##  Min.   :4.300   Min.   :2.000   Min.   :1.000
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600
##  Median :5.800   Median :3.000   Median :4.350
##  Mean   :5.843   Mean   :3.057   Mean   :3.758
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100
##  Max.   :7.900   Max.   :4.400   Max.   :6.900
##   Petal.Width          Species
##  Min.   :0.100   setosa    :50
##  1st Qu.:0.300   versicolor:50
##  Median :1.300   virginica :50
##  Mean   :1.199
##  3rd Qu.:1.800
##  Max.   :2.500
```

Even when we fit a support vector machine, a method you will learn more about in *Data Mining and Machine Learning I*, `summary` knows what to do.

```
library(e1071)
```

```
model <- svm(Species~., data=iris)
summary(model)
```

```
##
## Call:
## svm(formula = Species ~ ., data = iris)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##
## Number of Support Vectors:  51
##
##  ( 8 22 21 )
##
##
## Number of Classes:  3
##
## Levels:
##  setosa versicolor virginica
```

With what we have seen so far, the only way how we could make this work would be to have a huge number of `if` statements inside the function `summary` so that the correct output is shown. This however would be a bad idea. The code implementing `summary` would be a complex mess and making `summary` work for third-party packages like `e1071`, which we have just used to fit a support vector machine, would be a challenge.

The answer as to why `summary` (almost) always appears to do the right things is down to object orientation.

What is object-oriented programming (OOP) about?

So far we have thought of programming in terms of functions. A function takes one or more inputs, manipulates them or calculates something and then returns something.

In object-oriented programming we think primarily of types of (interacting) objects. An object has properties (which we can store in fields), but, and this is the difference, the object also has methods. Methods are functions that are specific to a certain type of object.

Think of a car as an example. It has properties like the colour, registration number etc. But you can also do something with it. You can for example start or stop the engine, load something into the boot or sound the horn. These would thus be examples of methods.

Most paradigms for object-oriented programming use classes: a class defines the behaviour of a given type of object. Objects are then instances of classes.

Using object orientation has many advantages. First of all, it makes it easier to structure complex projects, which will help with maintenance of the code in the future. Most importantly it allows for better "encapsulation": the user of a class or object does not need to know the details of the implementation, they don't even need to know of what exact class an object is. Take the `summary`, `plot` and `predict` methods as an example: most objects created when fitting a model support these methods. We don't need to know how predictions are calculated for say linear regression or support vector machines, all we need to remember is to use the `predict` method, which is used in pretty much the same way for different types of models.

Do we really need this?

You might (rightly) be wondering what the benefits of OOP are for data analytics. Clearly, we benefit from R's object orientation when we use `summary` or `predict`. But does this also mean that we should code up a data analysis in terms of objects?

It makes little sense to cast a simple data analytics workflow into an OOP context. If you can store your data in a small number of data frames, then manipulating these data frames using functions (and also exploiting R's functional programming techniques) is perfectly adequate.

Sometimes however you can think of the problem you are working on in terms of a small number of types of interacting objects with very specific behaviour. These types of objects can then be represented using classes in an OOP context. Such scenarios typically occur when you use R to retrieve data from other business systems or public or private APIs or when you write user interfaces or systems that automatically generate reports or data sets. Essentially, the more your code is concerned with managing something rather than just calculating or plotting something, the stronger will be the case for OOP.

Finally, if you code up a model or algorithm in R, it is always a good idea to use objects. This makes standard methods like `summary`, `plot` or `predict` work for your method as well.

R has different systems for object orientation. The default S3 system, which we will look at first is very well suited for handling the latter, whereas the more powerful R6 system, which we will look at as well, is better suited for "generic OOP". S3 is rather different from how other programming languages go about OOP: it is very light-touch and ad-hoc. R6 is much more similar to how objects and classes are handled in programming languages like Python, Java, C# or C++.

Key terms and concepts from OOP

This section gives an overview over key terms in object-oriented programming.

Class

A class defines the data structures behind a given type of object. You can think of it as a blueprint. It also defines the methods, i.e the functions associated with the class. R's S3 system is a slight exception to this. In S3, classes do not define the data structures, they are only concerned with methods.

Objects

An object is an instance of a class.

Fields

A field (sometimes also called member) is a variable that is stored within an object (just like we can store the content of a variable in a list).

Method

A method is a function that is specific to a class, or in other words, an action an object of that class can perform (like producing a plot or computing predictions).

Inheritance

In real life there is a hierarchy of types of objects. For example, SUVs and sports cars are a special type of car. Such hierarchical relationships between types of objects are represented using what is called "inheritance" in OOP.

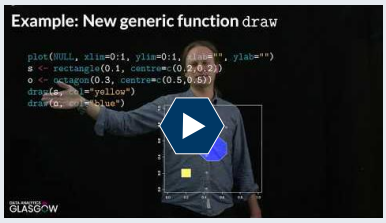
Let's go back to the example of cars. We would model the classes `SUV` or `SportsCar` as subclasses of the class `Car`, as they are specific types of cars. This way we can code up generic functionality of cars only once, without having to repeat it for different types of cars. We can then however also implement functionality that is specific to SUVs (like controlling the differential lock or 4WD) in the subclass `SUV`. (We will then say `Car` is a superclass of `SUV`.) `SUV` will inherit the fields and methods from `Car`, i.e. these will be also available for `SUV`s.

OO field guide of of Advanced R

http://adv-r.had.co.nz/OO-essentials.html

This chapter gives a systematic and more detailed overview of object-oriented programming in R.

S3 classes



S3 classes

https://youtu.be/itM2PqxyN8Q

Duration: 12m56s

Comparison to other paradigms

The standard model for OOP in R is what are called S3 classes (named after the version of S which first introduced them). The S3 model is different from how most other programming languages implement objects and classes.

S3 is based on generic functions, where most other programming languages use a member-function-based (sometimes called message-passing) approach. If you want to call the method `predict` for an object `model`, using `data=newdata` as additional argument, you would use

```
predict(model, data=newdata)
```

in R. If `model` is of the class `lm`, R translates this call into a call to the function

```
predict.lm(model, data=newdata).
```

However, in Python you would use

```
model.predict(data=newdata)
```

with C++, Java and Javascript using a similar syntax.

Another difference between S3 classes and more conventional paradigms for OOP is that R's S3 class system does not have formal class definitions and thus class methods are not defined as part of the class definition, but simply as stand-alone functions. It is just their name of the form `functionname.classname` that gives away that the function is a class method.

S3 is an extremely lightweight and ad hoc approach, which is sufficient for most of what we want to do in R. Objects are typically created when fitting a model. Once a model is fitted, the user just queries the model without modifying the object any further. S3 objects are ideally suited for this scenario. When using classes and objects in a more general context, S3 can be rather limiting. Luckily, R has alternative models for OOP, which are in that case better suited. We will look at one of them, R6, later on.

Creating objects

We can make (almost) any R object an instance of the class `classname` by setting

```
class(object) <- "classname"
```

In most cases `object` will be a list. All the above line of code does is adding an attribute `class` (set to `"classname"`) to the object, so that R can recognise the object as an instance of the class `classname`. It is our responsibility to make sure that the object behaves as one would expect of an object on this class. R will do no checking for us (we'll come back to this later on).

We can also use the function `class` to find out of what class an object is.

```
class(dplyr::starwars)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

We can see that an object can have more than one class. In that case the classes are listed from the more specific to the more generic. We'll come back to that later on.

We can use the function `inherits` to check whether an object is of a given class.

```
inherits(dplyr::starwars,"tbl_df")
```

```
## [1] TRUE
```

Example 1.

Suppose we want to create a class `rectangle` which contains the width, height and position of the centre of a rectangle.

```
#' "Constructor" function for creating rectangle objects
#' @param width width of the rectangle
#' @param height height of the rectangle (by default equal to the width)
#' @param centre vector of length 2 giving position of centre
#' @return ao object of the class rectangle
rectangle <- function(width, height=width, centre=c(0,0)) {
    object <- list(width=width, height=height, centre=centre)
    class(object) <- "rectangle"
    object
}
```

We can create a new rectangle using

```
s <- rectangle(1)             # Create a square of width and height 1
s
```

```
## $width
## [1] 1
##
## $height
## [1] 1
##
## $centre
## [1] 0 0
##
## attr(,"class")
## [1] "rectangle"
```

`s` is nothing other than a list, with an additional attribute `"class"` containing the class name.

Generic methods

We have already seen that the S3 approach to object oriented programming is based on generic functions. A generic function is a function that does not perform any operations other than what is called "method dispatch": if the first argument is of class `classname` and the function is called `fun`, then the dispatch mechanism calls the function `fun.classname` passing on all of its arguments. If the first argument is of more than one class the dispatch mechanism goes through the vector of classnames until it finds a function `fun.classname`. If it does not find one it will try `fun.generic` and, failing that, produce an error message.

Implementing existing generic methods If we want to implement the method `fun` for an object of class `classname` we have to define a function

```
fun.classname <- function(object, ...) {
  # Implementation goes here
}
```

The list of arguments should mirror the one used by the generic method.

"Standard" generic methods implemented by many S3 classes are `print`, `summary`, `plot`, `predict`, `residuals`, `coef` or `coefficients`. The `print` method is special: it controls how objects are printed in the console.

Example 2.

Suppose we want to provide a `summary` method for objects of the class rectangle we have defined in ref://s3exrect1. For this we have to define a function called `summary.rectangle`.

```
#' @method summary rectangle
summary.rectangle <- function(object) {
    cat(paste0("A rectangle of width ", object$width,
               " and height ",  object$height,
               ", located at (", object$centre[1],
               ",", object$centre[2],")\n"))
}
```

We can then run

```
summary(s)
```

```
## A rectangle of width 1 and height 1, located at (0,0)
```

Note that we just need to run `summary(s)`. We do not have to use `summary.rectangle(s)`. Because `s` is of the class `rectange` `summary(s)` will call the function `summary.rectangle` for us.

Supplementary material:
Obtaining S3 methods for classes

Sometimes the implementation of a method for a given class is not exported from packages, so `fun.classname` might not show the function implementing `fun` for class `classname`. In that case you can use `getS3method("fun", "classname")` to show the code implementing this method.

Defining new generic methods The S3 framework also allows for defining new generic methods. The key to a generic method is a call to `UseMethod("functionname")` in its body.

```
genericmethodname <- function(object, ...) {
    UseMethod("genericmethodname")
}
```

Example 3.

Suppose we want to create a new generic method called `draw` with an implementation for our class `rectangle`

```
#' Draw a shape
#' @param object shape to be drawn
#' @param ... additional details (colour etc.)
draw <- function(object, ...) {
    UseMethod("draw")
}

#' @method draw rectangle
draw.rectangle <- function(object, ...) {
    coords <- with(object, {
        cbind(centre[1]+width/2*c(-1,1,1,-1),
              centre[2]+height/2*c(-1,-1,1,1))
    })
    polygon(coords, ...)
}
```

We can now add the rectangle `s` from the previous examples to the current plot using

```
draw(s)
```

which R will translate to

```
draw.rectangle(s)
```

Inheritance

Due to its informal nature, S3 has no clean implementation of inheritance. However, if one assigns a vector of classes to an object, this can mimic the effects of inheritance. We have to list the classes from the more specific to the more general, i.e. for two classes the subclass comes before the superclass.

Example 4.

Suppose that in addition to the class `rectangle` we also have a class `octagon` with objects created using

```
#' "Constructor" function for creating octagon objects.
#' @param width with of the octagon.
#' @param height height of the octagon (default: identical to width).
#' @param corner width/height of the cut off corners that turn a rectangle
#'               into a octagon (relative to width and height). Defaults to
#'               0.2929.
#' @param centre vector of length 2 giving the centre of the octagon.
#'               Defaults to (0,0).
#' @return an object of the class octagon
octagon <- function(width, height=width, corner=.2929, centre=c(0,0)) {
    object <- list(width=width, height=height, corner=corner,
                   centre=centre)
    class(object) <- "octagon"
    object
}

#' @method draw octagon
draw.octagon <- function(object, ...) {
   coords <- with(object, {
       a <- c(-0.5, -0.5+corner, 0.5-corner, 0.5)
       cbind(centre[1]+width*c(a, -a),
             centre[2]+height*c(a[3:4],a[4:1],a[1:2]))
   })
   polygon(coords, ...)
}
```

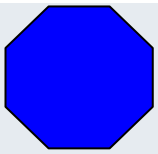
Both `draw` methods consist of two parts. The first part creates a set of coordinates for the vertices, whereas the second part draws a polygon using the coordinates. The second part is identical for rectangles and octagons suggesting that we could set up a superclass `shape` which has `rectangle` and `octagon` as subclasses. The `draw` method could then be implemented at `shape` level with the creation of the coordinates happening at `rectangle` and `octagon` level. To signal to R that rectangles and octagons are also shapes, we add `shape` to the class.

```
#' "Constructor" function for creating rectangle objects
#' @param width width of the rectangle
#' @param height height of the rectangle (by default equal to the width)
#' @param centre vector of length 2 giving position of centre
#' @return ao object of the classes rectangle and shape
rectangle <- function(width, height=width, centre=c(0,0)) {
    object <- list(width=width, height=height, centre=centre)
    class(object) <- c("rectangle", "shape")
    object
}

#' "Constructor" function for creating octagon objects.
#' @param width with of the octagon.
#' @param height height of the octagon (default: identical to width).
#' @param corner width/height of the cut off corners that turn a rectangle
#'               into a octagon (relative to width and height). Defaults to
#'               0.2929.
#' @param centre vector of length 2 giving the centre of the octagon.
```

```
#'               Defaults to (0,0).
#' @return an object of the class octagon
octagon <- function(width, height=width, corner=.2929, centre=c(0,0)) {
    object <- list(width=width, height=height, corner=corner,
                   centre=centre)
    class(object) <- c("octagon", "shape")
    object
}

#' Create a matrix of coordinates of vertices for a shape
#' @param object shape
#' @return a matrix of two columns containing the coordinates
makecoords <-  function(object) {
  UseMethod("makecoords")
}

#' @method makecoords rectangle
makecoords.rectangle <- function(object) {
    with(object, {
        cbind(centre[1]+width/2*c(-1,1,1,-1),
              centre[2]+height/2*c(-1,-1,1,1))
    })
}

#' @method makecoords octagon
makecoords.octagon <- function(object) {
   with(object, {
       a <- c(-0.5, -0.5+corner, 0.5-corner, 0.5)
       cbind(centre[1]+width*c(a, -a),
             centre[2]+height*c(a[3:4],a[4:1],a[1:2]))
   })
}

#' Draw a shape
#' @param object shape to be drawn
#' @param ... additional details passed on to polygon (colour etc.)
draw <- function(object, ...) {
    UseMethod("draw")
}

#' @method draw shape
draw.shape <- function(object) {
    coords <- makecoords(object)
    polygon(coords, ...)
}
```

We can now draw a square and an octagon on an empty plot canvas using

```
plot(NULL, xlim=0:1, ylim=0:1, xlab="", ylab="", xaxt="n", yaxt="n", bty="n")
s <- rectangle(0.1, centre=c(0.2,0.2))
o <- octagon(0.3, centre=c(0.5,0.5))
draw(s, col="yellow")
draw(o, col="blue")
```



This example illustrates one of the key strengths of object orientation. We only need to know that we can use the generic method `draw` to draw shapes. If we know that `s` and `o` are shapes we can draw them using `draw(s)` and `draw(o)`. We don't need to worry about what types of shape `s` and `o` are, as long as they are a valid shape. Object-orientation makes it very easy to conceal details of the implementation and provide users with a simple and clean interface – this is often referred to as "encapsulation".

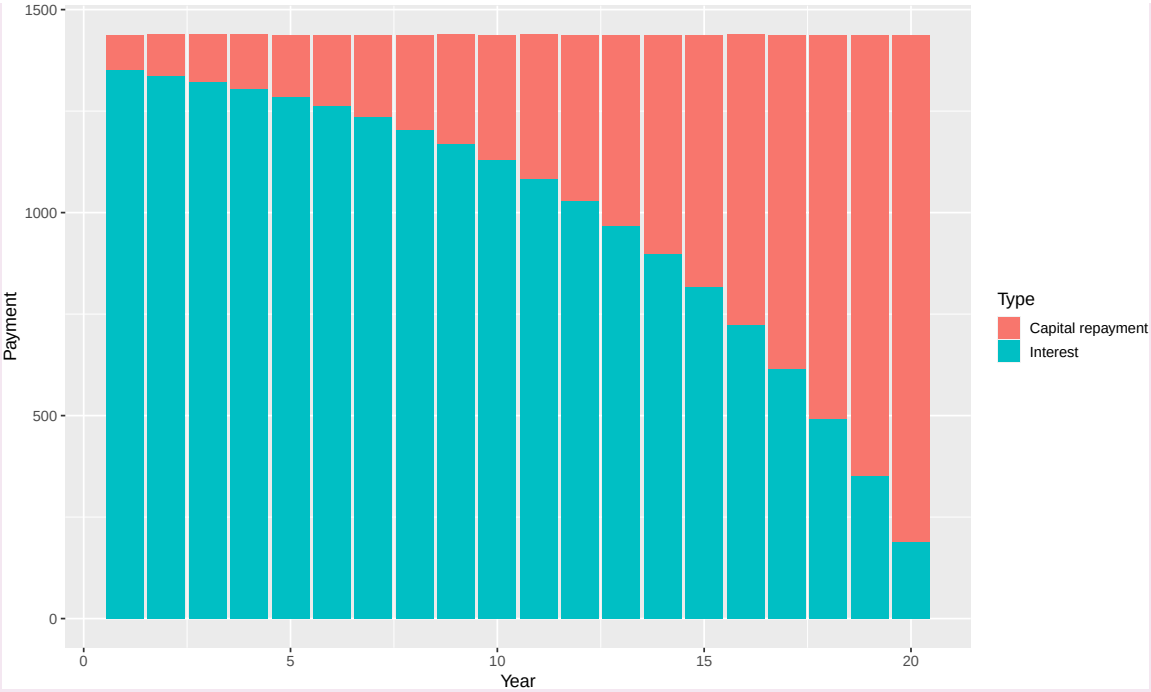
The generic method `predict`, which computes predictions for many models, is a nice example of this: It doesn't matter what exact type of model the model is and what the detailed algorithm for calculating predictions is, `predict` will do everything for us. In thew example below we will implement a simple machine learning model and code a custom predict method.

Task 1.

In week 2 we have looked at how to compute the repayments when taking out a loan of £9,000 over 20 years with an interest rate of 15%.

We have used the code below to compute the monthly payments as well as the split of the payments into capital repayment and interest.

```
n <- 20                                 # term of the loan
loan <- 9000                            # amount
interest.rate <- 0.15                   # effective annual interest rate
v <- 1 / (1+interest.rate)              # effective annual discount factor
payment <- loan * (1-v) / (v*(1-v^n))   # yearly payments
k <- 1:n                                # create vector with all possible k
alpha <- v^(n+1-k)                      # split factors
capital <- alpha * payment              # capital repayments
interest <- (1-alpha) * payment         # interest part
data <- data.frame(Year=1:n, rbind(data.frame(Type="Capital repayment",
                                              Payment=capital),
                                   data.frame(Type="Interest",
                                              Payment=interest)))
library(ggplot2)
ggplot(data=data) + geom_col(aes(Year, Payment, fill=Type))
```



Reorganise the code as set out below.

- Create a function `loan`, which takes the amount of the loan, the interest rate and the duration the loan is taken out for as arguments and which returns an object of the class `loan`.
- Create a `print` method for the class `loan`, that prints the amount, interest rate, duration and yearly repayments.
- Create a `plot` method which produces the above plot.

Supplementary material: S3 has no built-in checks and protections

R does not stop us from doing stupid things. We can for example pretend that a data frame is a model fit from a linear model:

```
test <- data.frame(x=rnorm(10))       # Create a toy data frame
class(test)
```

```
## [1] "data.frame"
```

```
class(test) <- "lm"                   # Rogue code relabelling it as lm object
class(test)
```

```
## [1] "lm"
```

R does not check whether `test` is a valid `lm` object. However trying to get predictions out of our fake linear model will fail.

```
predict(test)
```

```
## Error in x$terms %||% attr(x, "terms") %||% stop("no terms component nor attribute"): no terms componer
```

Similarly I can take a proper `lm` object and remove some of its components

```
model <- lm(dist~speed, data=cars)    # Fit linear models
model$coefficients <- NULL            # Sabotage the coefficients
```

Again R hasn't stopped me. However my `model` is now so broken that I can't use it properly any more.

```
predict(model)
```

```
## Error in X[, piv, drop = FALSE] %*% beta[piv]: requires numeric/complex matrix/vector arguments
```

Other programming languages have much stricter systems that would never ever allow something like this. R however doesn't protect you from yourself: there is almost never a reason why you should change the class of an object (other than when you create it) or meddle with one, so just don't do these things and you will be fine.

Supplementary material: S4

R also has a more advanced class system, called S4 (again named after the S version which first introduced them), implemented in the built-in package `methods`. S4 has two main advantages over S3.

- S3 methods can dispatch only based on their first argument: depending on the class of the first argument R will choose an appropriate method to call. For S3 methods, R will however not look at the class of the second argument. However, S4's dispatch mechanism can take the classes of any argument into account ("multiple dispatch"). A prominent package making use of this feature of S4 is the package `Matrix` implementing sparse matrices: there are many different ways of storing sparse matrices (implemented in different classes). When multiplying two sparse matrices we need to look at the class of both arguments and then choose the appropriate method.
- S4 classes have formal class definitions and thus offer protection against the meddling with objects we have just tried out for S3.

There is however a price to pay for this. Writing S4 classes requires much more effort, thus it is best to only use them when you cannot achieve what you want using S3 classes.

From a user's point of view, the main difference is that fields in an S4 object are accessed using `object@element` rather than `object\$element`, as used for S3 objects.

Case study

Example 5 (Naïve Bayes classifier).

In this example we will build a class for a naïve Bayes classifier for categorical data.

Consider a problem where we want to predict a categorical variable y_i from a set of categorical covariates ("features") $x_{i1}, \ldots, x_{ip}$, where training data is available for $i = 1, \ldots, n$ observations.

We want to use the training data to estimate the probability $P(Y_i = y_i | X_{i1} = x_{i1}, \ldots, X_{ip} = x_{ip})$. You have seen in *Probability and Stochastic Models* or in *Probability and Sampling Fundamentals* that we can use Bayes theorem to write

$$P(Y_i = y_i | X_{i1} = x_{i_1}, \ldots, X_{ip} = x_{ip}) \propto P(Y_i = y_i) P(X_{i1} = x_{i_1}, \ldots, X_{ip} = x_{ip} | Y_i = y_i)$$

We have omitted the denominator as it is just a normalisation constant, which we can calculate numerically later on.

You have seen in *Probability and Stochastic Models* or in *Probability and Sampling Fundamentals* that

$$\begin{aligned} P(X_{i1} = x_{i1}, \ldots, X_{ip} = x_{ip} | Y_i = y_i) &= P(X_{i1} = x_{i1} | Y_i = y_i) P(X_{i2} = x_{i2} | X_{i1} = x_{i1}, Y_i = y_i) \cdots \\ &\quad \cdot P(X_{ip} = x_{ip} | X_{i1} = x_{i1}, \ldots X_{i,p-1} = x_{i,p-1}, Y_i = y_i) \end{aligned}$$

The naïve Bayes classifier, sometimes also called idiot's Bayes, makes the (typically incorrect) assumption that given Y_i the X_{ij} are conditionally independent and thus

$$P(X_{i1} = x_{i1}, \ldots, X_{ip} = x_{ip} | Y_i = y_i) = P(X_{i1} = x_{i1} | Y_i = y_i) P(X_{i2} = x_{i2} | Y_i = y_i) \cdots P(X_{ip} = x_{ip} | Y_i = y_i)$$

This way we do not have to model how the different X_{ij} relate to each other within one class. This simplifies the modelling task substantially.

Exploiting this assumption we obtain

$$P(Y_i = y_i | X_{i1} = x_{i_1}, \ldots, X_{ip} = x_{ip}) \propto P(Y_i = y_i) P(X_{i1} = x_{i1} | Y_i = y_i) P(X_{i2} = x_{i2} | Y_i = y_i) \cdots P(X_{ip} = x_{ip} | Y_i = y_i)$$

we thus need to estimate the "prior" probability of class y_i as well as the probabilities $P(X_{ij} = x_{ij}|Y_i = y_i)$.

If the response and all covariates are categorical, we can estimate these probabilities simply by the proportions in the training data.

In most cases the above assumption of conditional independence is not justified. In that case one can show that the naïve Bayes classifier estimates more extreme probabilities, i.e. it is more confident in its predictions than what it should be.

You will learn more about naïve Bayes classifiers in *Predictive Modelling* and *Data Mining and Machine Learning I*.

We will start by writing the function that estimates the naïve Bayes model

```
naivebayes <- function(x, y, prior, smooth=1) {
    # Make sure x is a data frame
    x <- as.data.frame(x)
    # Make sure response and all covariates are factors
    if (!is.factor(y) || !all(vapply(x,is.factor,logical(1))))
        stop("x and y must be factors")
    # Check x and y have same number of observations
    if (length(y)!=nrow(x))
        stop("x and y do not have the same number of observations")
    # Use marginal as prior of y if none specified
    if (missing(prior)) {
        # Get marginal distribution of y
        ydist <- as.numeric(table(y))
        prior <- as.numeric(table(y)/length(y))
    }
    if (is.null(names(prior)))
        names(prior) <- levels(y)
    # Get conditional distributions of x given y
    # proceeds through x column by column
    # We use lapply as lapply is guaranteed to return a list
    xdists <- lapply(x, function(xcol) {
        # Get joint distribution of column xcol and y
        # Add smooth to avoid estimated probabilities of 0
        xtab <- xtabs(~y+xcol) + smooth
        # Get marginal distribution of y (+smooth)
        ymarg <- rowSums(xtab)
        # Conditional is joint divided by marginal
        sweep(xtabs(~y+xcol),1,ymarg,"/")
    })
    # Assemble fitted model
    result <- list(xdists=xdists, prior=prior, y=y, x=x)
    # Make object of class "naivebayes"
    class(result) <- "naivebayes"
    result
}
```

We will illustrate the model using spam detection as an example. Many systems for spam detection (SpamAssassin, the spam detection within Thunderbird etc.) are based on naïve Bayes classifiers, as these are quick to train and can cope with a very large number of features.

If you load

```
load(url("https://github.com/UofGAnalyticsData/R/raw/main/Week%2011/spam.RData"))
```

you have a data frame `spam` in your workspace. The first column indicates whether the text message is spam or ham (i.e. not spam). The second column contains the message itself. The remaining columns contain indicators whether the text message contains some key words.

We can now train the naïve Bayes classifier using

```
model <- naivebayes(spam[,-(1:2)], spam$spam)
```

When we enter `model` into R, it floods the screen by printing the entire object. We can avoid this by providing a custom print method.

```
print.naivebayes <- function(model) {
    if (!inherits(model, "naivebayes"))
        stop("model must be of the class naivebayes.")
    cat(paste0("A naïve Bayes model with ",length(model$xdists),
               " covariates (features) trained on ",length(model$y),
               " observations.\n"))
}
```

Now the object is printed in a more concise way

```
model
```

```
## A naïve Bayes model with 100 covariates (features) trained on 4574 observations.
```

We can still access all the elements (even though they are not shown when printing). For example, we still can access the prior using

```
model$prior
```

```
##       ham      spam
## 0.8664189 0.1335811
```

Next, we will implement a `predict` method, so that we can decide whether we should flag text messages as spam.

The predict method takes the fitted model as first argument and allows the user to optionally specify new data. The argument `probabilities` lets the user choose whether to return the probabilities for each outcome class, or just the name of the most likely class.

```
predict.naivebayes <- function(model, newdata, probabilities=FALSE) {
    if (!inherits(model, "naivebayes"))
        stop("model must be of the class naivebayes.")
    # If the user does not provide newdata set it to model$x
    if (missing(newdata))
        newdata <- model$x
    if (!is.data.frame(newdata))
        newdata <- as.data.frame(newdata)
    # Start with prior probabilities
    prob <- matrix(rep(model$prior, each=nrow(newdata)), nrow=nrow(newdata))
    # Copy in names of outcomes
    colnames(prob) <- names(model$prior)
    # Multiply by p(x_j|y)
    for (j in names(model$xdists)) {
        prob <- prob * t(model$xdists[[j]][,newdata[,j]])
    }
    probs <- sweep(prob, 1, rowSums(prob), "/")
    # Return probabilities if user has requested this
    if (probabilities)
        return(probs)
    # Otherwise return most likely class
    class.idx <- apply(probs, 1, which.max)
    # Translate indices to labels
    ylabels <- names(model$prior)
    ylabels[class.idx]
}
```

We can now predict from our model.

```
predictions <- predict(model)
```

To see how good our model is we look at the so-called confusion matrix, which is just a cross classification table of the predictions and the actual observations.

```
actual <- model$y
xtabs(~actual+predictions)
```

```
##       predictions
## actual  ham spam
##   ham  3940   23
##   spam  103  508
```

We want to see large numbers on the diagonal, where predicted class and actual class agree. Our classifier gave the correct results for 97.2% of the messages, which is, given how simple the model was, not bad.

We can wrap up this code as a `summary` method.

```
summary.naivebayes <- function(model) {
    if (!inherits(model, "naivebayes"))
        stop("model must be of the class naivebayes.")
    cat("Confusion matrix (training data):\n")
    actual <- model$y
    predictions <- predict(model)
    xtab <- xtabs(~actual+predictions)
    print(xtab)
    misclass <- 1-sum(diag(xtab))/sum(xtab)
    cat("\nMisclassification rate (training data): ",
        signif(100*misclass, 3),"%\n")
    for (i in 1:nrow(xtab)) {
        misclassi <- sum(xtab[i,-i])/sum(xtab[i,])
        cat(paste0(signif(100*misclassi,3),"% of ",rownames(xtab)[i],
                   " incorrectly classified.\n"))
    }
}
```

We use the function `signif` to print the numbers more nicely.

We can then run `summary`.

```
summary(model)
```

```
## Confusion matrix (training data):
##       predictions
## actual  ham spam
##   ham  3940   23
##   spam  103  508
##
## Misclassification rate (training data):  2.75 %
## 0.58% of ham incorrectly classified.
## 16.9% of spam incorrectly classified.
```

Ideally we should not assess the method on the data we have trained it with. The data frame `spam_test` contains 1,000 unseen test cases, which allows us to assess the performance of our model more honestly.

```
predictions <- predict(model,spam_test)
actual <- spam_test$spam
xtabs(~actual+predictions)
```

```
##       predictions
## actual ham spam
##   ham  860    4
##   spam  28  108
```

On the test data, the naïve Bayes classifier gave the correct results for 96.8% of the messages.

Naïve Bayes classifiers are however not ideal for spam detection. A message containing "hot girls" is almost certainly spam, whereas the words "hot" and "girls" on their own are more innocent. However this goes against the conditional independence assumption we have made above to simplify calculations. Thus the naïve Bayes classifier cannot take such considerations into account.

R6 classes

The shortcomings of S3

If you are familiar with the classical approach to OOP (Java, C++, C#, Python, ...) you have probably found R's S3 classes rather unsatisfactory. Though the S3 approach works quite well for implementing statistical models (which is what most classes in R are about), S3 isn't that well suited to projects involving several interacting classes (like for example complex simulation studies or for developing user interfaces).

The main drawback of S3 objects is how they are passed on to functions. S3 objects are like all other R objects (except for environments) passed on "by value": as soon an object is changed inside a function, a local copy is created and all changes made to the object are effectively rolled-back at the end of the function. So, in other words, S3 objects are not mutable. A work-around in S3 is to return the modified object, but this is not always feasible.

In the standard use case of S3 objects, an S3 object is created (say by fitting a model). This object is then never modified, as methods like `print`, `summary`, `plot` and `predict` just query the object without modifying it.

However, most classical approaches on OOP are centred around mutable objects and functions modifying these.

Example 6.

This example will illustrate the challenges that arise from S3 objects not being mutable.

Suppose you want to create a class `s3person`, which has a name and a list of friends. We can write a function that creates such an S3 object.

```
#' Create a person
#' @param name of the person
#' @param friends list of friends (optional)
#' @return an object of the class s3person.
s3person <- function(name, friends=list()) {
    person <- list(name=name, friends=friends)
    class(person) <- "s3person"
    person
}

#' @method print person
print.s3person <-  function(person) {
    cat(person$name, "- a person with",length(person$friends),"friend(s):\n")
    # Extract names of friends
    friend.names <- vapply(person$friends, function(friend)
        friend$name, character(1))
    # Print them (separated by a ,)
    cat(paste0(friend.names,collapse=", "),"\n")
}
```

Suppose we want to now write a function `befriend` that makes two persons friends.

```
#' Store two persons as each others' friends
#' @param person1 a person
#' @param person2 another person
befriend <- function(person1, person2) {
    person1$friends <- c(person1$friends, person2)
    person2$friends <- c(person2$friends, person1)
}
```

Unfortunately this won't work in R.

```
jack <- s3person("Jack")
jill <- s3person("Jill")
befriend(jack, jill)
jack
```

```
## $name
## [1] "Jack"
```

```
##
## $friends
## list()
##
## attr(,"class")
## [1] "s3person"
```

```
jill
```

```
## $name
## [1] "Jill"
##
## $friends
## list()
##
## attr(,"class")
## [1] "s3person"
```

Though Jack and Jill became friends inside the function `befriend`, they aren't friends outside that function. The two lines which add the other person as a friend, modify the objects `person1` and `person2` and thus R creates a local copy for these modifications. These local copies will then get deleted at the end of the function.

A similar issue arisen when storing a person in the list of friends. As soon as a change is made to `jack` after he has become friends with `jill`, we will have two different Jacks, one who is the friend of `jill` (not updated) and the person referred to as `jack`.

Other programming languages avoid this issue by passing objects "by reference" or by using pointers, but R does not support this (though environments can be used to mimic some of this).

R6

The R package R6 provides an alternative framework for object orientation, which is much more similar to how object orientation is handled in other programming languages.

The main difference between R6 and S3 are:

- R6 objects are mutable as they are passed by reference. This makes R6 classes particularly useful for representing "stateful" object, i.e. objects that change over time. For this reason Shiny is based on R6.
- R6 classes have formal definitions and class methods are defined in the class definition.
- S3 methods are called like regular functions, `methodname(object, ...)`. R6 class methods are using `object\$methodname(...)`.

Class definitions The syntax for declaring an R6 class is

```
ClassName <- R6Class("ClassName",
#                     ^^^^^^^^^ should be the same as the class name
                    public=list(
                        field=value,
                        ...
                        initialize=function(...) {
                            # Code for object initialisation ("constructor")
                        },
                        methodname=function(...) {
                            # Function implementation ...
                        },
                        print=function() {
                            # Print implementation
                        }
                        ...
                    ))
```

An R6 object of the class `ClassName` can then be created using

```
object <- ClassName$new(...)
```

R then creates the object and runs the method `initialize(...)`. The arguments given to `new(...)` will be passed on to `initialize(...)`. The fields and functions of the list given as argument `public` are available as `object\$field` and `object\$methodname(...)`. Inside an R6 method the object is available as a `self`.

By default, R6 objects are locked and you cannot create a new field `object\$newfield`. All fields need to be "declared" by adding them to the list `public` (with some initial value, which can be `NULL`).

Just like for S3 objects, the `print` method is getting called when R prints the object to the screen (for example when its name is being entered in the console).

Example 7.

We struggled implementing our friendship model in S3. It is straightforward to implement in R6.

```
library(R6)
Person <- R6Class("Person",
                  public=list(
                      name=NULL,
                      friends=list(),

                      initialize = function(name, friends=list()) {
                          self$name <- name
                          self$friends<- friends
                      },

                      befriend = function(person) {
                          self$friends <- c(self$friends, person)
                          person$friends <- c(person$friends, self)
                      },

                      print  = function() {
                          cat(self$name, "- a person with",length(self$friends),
                              "friend(s):\n")
                          # Extract names of friends
                          friend.names <- vapply(self$friends,
                                                 function(elt) elt$name, character(1))
                          # Print them (separated by a ,)
                          cat(paste0(friend.names,collapse=", "),"\n")
                      }

                  )
)
```

We can then create Jack and Jill using

```
jack <-  Person$new("Jack")
jill <-  Person$new("Jill")
jack$befriend(jill)
jack
```

```
## Jack - a person with 1 friend(s):
## Jill
```

```
jill
```

```
## Jill - a person with 1 friend(s):
## Jack
```

Private members and functions In addition to the argument `public`, we can provide a list as argument `private` when we define a class using `R6Class`. The fields and functions of the list given as argument `private` not externally visible, i.e. the user cannot directly manipulate these fields or run these methods. The private fields and functions are available as a list `private`.

Example 8.

Let's return to example 7. At the moment the user can simply change the name of a person.

```
jill$name <- "Jennifer"
jill
```

```
## Jennifer - a person with 1 friend(s):
```

```
## Jack
```

in which case Jill will now be called Jennifer. If we want to prevent the user from changing the name of a person, we have to hide the name from the user. We can do this by making it private. We need to do three things for this:

- We need to remove the initialisation `name=NULL` from the list provided as argument `public` to a new list provided as argument `private`.
- We need to introduce a new method `getName` which will return the name of person (without letting the user change the name).
- Finally we need to replace `self\$name` by `private\$name`.

```
library(R6)
Person <- R6Class("Person",
                  public=list(
                      friends = list(),

                      initialize = function(name, friends=list()) {
                          private$name <- name
                          self$friends<- friends
                      },

                      befriend = function(person) {
                          self$friends <- c(self$friends, person)
                          person$friends <- c(person$friends, self)
                      },

                      getName = function() {
                          private$name
                      },

                      print  = function() {
                          cat(self$getName(), "- a person with",
                              length(self$friends),"friend(s):\n")
                          # Extract names of friends
                          friend.names <- vapply(self$friends,
                                                 function(elt) elt$getName(), character(1))
                          # Print them (separated by a ,)
                          cat(paste0(friend.names,collapse=", "),"\n")
                      }

                  ),
                  private=list(
                      name=NULL
                  )
)
```

Now we cannot change Jill's name any more.

```
jill <-  Person$new("Jill")
jill$name <- "Jennfier"
```

```
## Error in jill$name <- "Jennfier": cannot add bindings to a locked environment
```

This is not 100% bullet-proof though. R6 classes are based on S3 classes and environments, so a user can (with some effort) use the internal S3 representation of the R6 object to access the environments and modify private fields.

We could have also made the list of friends private, but this is a bit more complicated, because our method `befriend` currently modifies the friend's list of friends, which would not be accessible if it was private.

Task 2.

Add a method `setName` to the class `Person`, so that the name can be set by the user. Produce an error message if the user tries to set a name of length 0.

Inheritance R6 classes support inheritance in the form of a single superclass being specified as argument `inherit` when creating the class using `R6Class`. Fields and methods from the superclass are then available in the subclass, as if they were defined in the subclass.

Example 9.

Think of a class model for pets, which we assume to be dogs or cats. We can formalise this as a class `Pet` with two subclasses `Dog` and `Cat`,

```
Pet <- R6Class("Pet",
               public=list(
                   name=NULL,

                   initialize = function(name) {
                       self$name <- name
                   }
               )
)

Dog <- R6Class("Dog",
               inherit=Pet,
               public=list(
                   initialize = function(name) {
                       super$initialize(name)   # Call initialize from superclass Pet
                   },
                   makeSound = function() {
                       cat("Woof!\n")
                   }
               )
)

Cat <- R6Class("Cat",
               inherit=Pet,
               public=list(
                   initialize = function(name) {
                       super$initialize(name)   # Call initialize from superclass
                   },
                   makeSound = function() {
                       cat("Meow!\n")
                   }
               )
)

milly <- Cat$new("Milly")
max <- Dog$new("Max")
milly$makeSound()
```

```
## Meow!
```

```
max$makeSound()
```

```
## Woof!
```

We can now access

```
milly$name
```

```
## [1] "Milly"
```

`name` is a field of objects of class `Animal`. However, `milly` is a object of the class `Cat`, which inherits from `Animal` and thus the fields from `Animal` are also available.

Task 3 (Shapes).

Recreate the shape example from example ref://s3exrect4 using R6 classes.

Supplementary material:
Reference classes

R also has a built-in a class system which is very similar to R6 classes: reference classes (RC). Compared to reference classes R6 is the simpler, more lightweight and slightly better performing system. The syntax for R6 classes is also closer to the syntax for defining classes in other OOP-based programming languages (Java, C++, C#, Python, …)

Answers to tasks

Answer to Task 1. We start by defining the "constructor" function and implement a print and plot method.

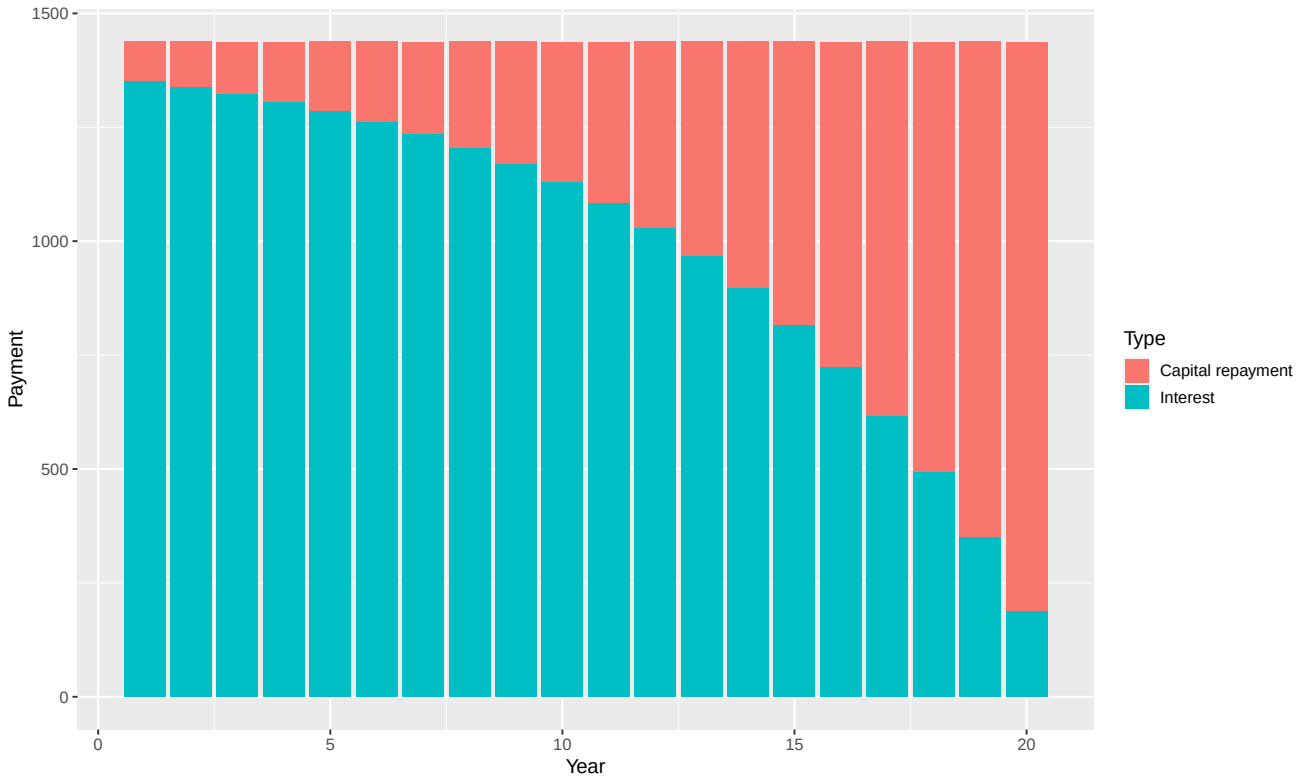
```
#' Calculate repayments for a loan
#' @param loan amount of the loan
#' @param n duration until loan should be repaid
#' @param interest.rate interest rate (not as a percentage)
#' @return object of the class loan
loan <- function(loan, n, interest.rate) {
    v <- 1 / (1+interest.rate)             # effective annual discount factor
    payment <- loan * (1-v) / (v*(1-v^n))  # yearly payments
    object <- list(loan=loan, n=n, interest.rate=interest.rate,
                   v=v, payment=payment)
    class(object) <- "loan"
    object
}

#' @method print loan
print.loan <- function(object) {
    if (!inherits(object, "loan"))
        stop("Function only works for an object of class loan")
    cat(paste0("A loan of £",round(object$loan, 2)," taken out over ",object$n,
               " years at an interest rate of ",signif(object$interest.rate*100, 3),
               "%.\nThe resulting yearly repayments are £",round(object$payment,2),
               ".\n"))
}

#' @method plot loan
plot.loan <- function(object) {
     if (!inherits(object, "loan"))
        stop("Function only works for an object of class lob")
     k <- 1:object$n                         # create vector with all possible k
     alpha <- object$v^(object$n+1-k)        # split factors
     capital <- alpha * object$payment       # capital repayments
     interest <- (1-alpha) * object$payment # interest part
     data <- data.frame(Year=1:object$n, rbind(data.frame(Type="Capital repayment",
                                                          Payment=capital),
                                               data.frame(Type="Interest",
                                                          Payment=interest)))
     require(ggplot2)
     ggplot(data=data) + geom_col(aes(Year, Payment, fill=Type))
}
```

```
l <- loan(9000, 20, 0.15)
```

```
l
```

```
## A loan of £9000 taken out over 20 years at an interest rate of 15%.
## The resulting yearly repayments are £1437.85.
```

```
plot(l)
```



Answer to Task 2. Starting with the class definition from above we add a class method “setName”

```
library(R6)
Person <- R6Class("Person",
                  public=list(
                      friends = list(),

                      initialize = function(name, friends=list()) {
                          private$name <- name
                          self$friends<- friends
                      },

                      befriend = function(person) {
                          self$friends <- c(self$friends, person)
                          person$friends <- c(person$friends, self)
                      },

                      getName = function() {
                          private$name
                      },

                      setName = function(name) {
                          if (nchar(name)>0) {
                              private$name <- name
                          } else {
                              stop("Blank names not allowed")
                          }
                      },

                      print  = function() {
                          cat(self$getName(), "- a person with",length(self$friends),"friend(s):\n")
                          # Extract names of friends
                          friend.names <- vapply(self$friends, function(elt) elt$getName(),
                                                 character(1))
                          # Print them (separated by a ,)
```

```
                        cat(paste0(friend.names,collapse=", "),"\n")
                    }

                ),
                private=list(
                    name=NULL
                )
)
```

Answer to Task 3 (Shapes). We start by defining the R6 class.

```
library(R6)
Shape <- R6Class("Shape",
                 public = list(
                     draw = function(...) {
                         coords <- self$makecoords()
                         polygon(coords, ...)
                     }
                 )
)

Rectangle <- R6Class("Rectangle",
                     inherit=Shape,
                     public=list(
                         centre=NULL,
                         width=NULL,
                         height=NULL,

                         initialize = function(width, height=width,
                                               centre=c(0,0))  {
                             self$width <- width
                             self$height <- height
                             self$centre <- centre
                         },

                         makecoords =function() {
                             cbind(self$centre[1]+self$width/2*c(-1,1,1,-1),
                                   self$centre[2]+self$height/2*c(-1,-1,1,1))
                         }
                     )
)


Octagon <- R6Class("Octagon",
                     inherit=Shape,
                     public=list(
                         centre=NULL,
                         width=NULL,
                         height=NULL,
                         corner=NULL,

                         initialize = function(width, height=width,
                                               corner=.2929, centre=c(0,0)) {
                             self$width <- width
                             self$height <- height
                             self$corner <- corner
                             self$centre <- centre
                         },

                         makecoords =function() {
                             a <- c(-0.5, -0.5+self$corner, 0.5-self$corner, 0.5)
```

```
                        cbind(self$centre[1]+self$width*c(a, -a),
                              self$centre[2]+self$height*c(a[3:4],a[4:1],
                                                           a[1:2]))
                    }
                )
)
```

We can then draw same the picture as we have drawn using S3 classes.

```
plot(NULL, xlim=0:1, ylim=0:1, xlab="", ylab="", xaxt="n", yaxt="n", bty="n")
s <- Rectangle$new(0.1, centre=c(0.2,0.2))
o <- Octagon$new(0.3, centre=c(0.5,0.5))
s$draw(col="yellow")
o$draw(col="blue")
```