Programmation orienté objet en R

Sophie Baillargeon, Université Laval

2018-03-19

Table des matières

La programmation orientée objet est un paradigme de programmation basé sur le concept d'objets, qui peuvent contenir des données et métadonnées (attributs), et qui sont associés à des procédures, souvent appelées méthodes. R propose plusieurs systèmes orientés objet en R :

- **S3** : Il s'agit du système le moins formel, mais le plus utilisé, en particulier dans les packages `base` et `stats`. Ces notes traitent principalement de ce système.
- **S4** : Ce système fonctionne de façon similaire au système S3, mais il est plus formel. La majorité des packages sur Bioconductor utilisent ce système. Nous verrons ici comment manipuler des objets de classe S4, mais pas comment en créer.
- Autres : Parmi les autres systèmes orientés objet en R, il y a eu **RC** (aussi nommé **R**), le système « Reference Classes ». Celui-ci n'a cependant jamais été très utilisé. Récemment, le système **R6** a vu le jour, afin de remplacer le système RC. Comparativement aux systèmes S3 et S4, ce système se rapproche davantage du paradigme orienté objet des langages informatiques Python et Java notamment. Comparativement au système RC, il est plus simple et plus rapide. Cette année, nous n'approfondirons pas ce système dans le cours. Cependant, si son utilisation se répand, il sera peut-être couvert dans des éditions futures du cours.

Système S3

À chaque fois que nous avons effleuré le sujet de la programmation orientée objet dans ce cours, nous parlions toujours du système S3. Nous l'avons mentionné dans les notes sur :

- les concepts de base ;
- les graphiques ;
- les calculs statistiques et mathématiques en R.

Le fonctionnement de ce système est très simple. Il est possible d'attribuer des *classes* S3 aux objets R. Ces classes déterminent comment les *fonctions génériques* se comportent en recevant en entrée un certain objet. Une fonction générique est dite *polymorphe*. Elle possède plusieurs définitions, appelées *méthodes*, pour des objets de différentes classes. Techniquement, une fonction générique R ne fait que rediriger les arguments qui lui sont fournis en entrée vers la méthode associée à la classe des objets donnés comme premiers arguments (souvent seulement la classe du premier argument importe).

Utilisation de classes S3

Nous avons déjà mentionné que les fonctions suivantes sont génériques : `mean`, `plot`, `print`, `summary`, `coef` et plusieurs autres fonctions génériques d'extraction d'information de la sortie d'une fonction d'ajustement de modèle. La fonction `print` est probablement la fonction de cette liste que nous utilisons le plus souvent, puisqu'elle est appelée à chaque fois que nous affichons un objet dans la console. La définition de cette fonction est la suivante :

```
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x0000000016eb8c78>
## <environment: namespace:base>
```

Il n'y a qu'une seule instruction dans le corps de cette fonction : un appel à la fonction `UseMethod`. La fonction `UseMethod` vérifie d'abord la classe des objets fournis aux arguments nommés dans la définition de la fonction générique, ici uniquement `x`. Ensuite, elle appelle la méthode correspondant à la classe obtenue, si elle existe, en lui fournissant en entrée les arguments qui ont été fournis dans l'appel à la fonction générique.

Par exemple, considérons le data frame suivant (il s'agit d'un jeu de données du package `datasets`).

```
str(women)
```

```
## 'data.frame':    15 obs. of  2 variables:
##  $ height: num  58 59 60 61 62 63 64 65 66 67 ...
##  $ weight: num  115 117 120 123 126 129 132 135 139 142 ...
```

Les data frames possèdent un attribut `class`.

```
attributes(women)
```

```
## $names
## [1] "height" "weight"
##
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
##
## $class
## [1] "data.frame"
```

Lors de l'évaluation de l'appel à la fonction `print` suivant :

```
print(women)
```

R va d'abord vérifier la classe du premier argument fourni :

```
class(women)
```

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

Puis R va vérifier si une méthode `print` est définie pour cette classe. La fonction `methods` nous permet de connaître toutes les méthodes définies pour une fonction générique.

```
methods(print)
```

Le résultat n'est pas affiché en entier, car le nombre de méthodes pour la fonction générique `print` est trop grand. Voici cependant un extrait de la sortie obtenue.

```
## . . .
## [67] print.condition                                      print.connection
## [69] print.CRAN_package_reverse_dependencies_and_views* print.data.frame
```

```
## [71] print.Date                                          print.default
## [73] print.dendrogram*                                   print.density*
## . . .
## see '?methods' for accessing help and source code
```

Pour une classe spécifique, une méthode S3 portera le nom : `nomFonctionGenerique.nomClasse`. Nous voyons ici que la méthode `print.data.frame` existe. Celle-ci est une fonction, dont la définition est la suivante :

```
print.data.frame
```

```
## function (x, ..., digits = NULL, quote = FALSE, right = TRUE,
##     row.names = TRUE)
## {
##     n <- length(row.names(x))
##     if (length(x) == 0L) {
##         cat(sprintf(ngettext(n, "data frame with 0 columns and %d row",
##             "data frame with 0 columns and %d rows"), n), "\n",
##             sep = "")
##     }
##     else if (n == 0L) {
##         print.default(names(x), quote = FALSE)
##         cat(gettext("<0 rows> (or 0-length row.names)\n"))
##     }
##     else {
##         m <- as.matrix(format.data.frame(x, digits = digits,
##             na.encode = FALSE))
##         if (!isTRUE(row.names))
##             dimnames(m)[[1L]] <- if (identical(row.names, FALSE))
##                 rep.int("", n)
##             else row.names
##         print(m, ..., quote = quote, right = right)
##     }
##     invisible(x)
## }
## <bytecode: 0x00000000174aabb0>
## <environment: namespace:base>
```

R va donc finalement appeler cette fonction.

Donc, la commande

```
women
```

qui revient en fait à la commande

```
print(women)
```

cache l'évaluation de la commande suivante

```
print.data.frame(women)
```

qui produit le résultat suivant :

```
##    height weight
## 1      58    115
## 2      59    117
## 3      60    120
## 4      61    123
## 5      62    126
```

```
## 6      63    129
## 7      64    132
## 8      65    135
## 9      66    139
## 10     67    142
## 11     68    146
## 12     69    150
## 13     70    154
## 14     71    159
## 15     72    164
```

Qu'arrive-t-il lorsque la méthode `nomFonctionGenerique.nomClasse` n'existe pas ?

Dans ce cas, R utilise la méthode `nomFonctionGenerique.default`, si elle existe.

Par exemple, créons une copie de `women` à laquelle nous allons retirer l'attribut `class` avec la fonction `unclass`.

```
women2 <- unclass(women)
attributes(women2)
```

```
## $names
## [1] "height" "weight"
##
## $row.names
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

Comment agit la fonction générique `print` avec cet objet ?

```
print(women2)
```

```
## $height
##  [1] 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
##
## $weight
##  [1] 115 117 120 123 126 129 132 135 139 142 146 150 154 159 164
##
## attr(,"row.names")
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

Voici l'explication de ce qui se passe. `women2` n'a pas d'attribut `class`, mais tout objet R possède une classe implicite. La fonction `class` vérifie d'abord si l'objet possède un attribut nommé `"class"`. Si c'est le cas, elle retourne cet attribut.

```
class(women)
```

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

Sinon, elle retourne la classe implicite de l'objet.

```
class(women2)
```

```
## [1] "list"
```

`women2` possède la classe implicite `list`. Ce résultat est cohérent avec le fait qu'un data frame est un type particulier de liste. Ainsi, lors de l'évaluation de la commande `print(women2)`, R cherche la méthode `print.list`, mais celle-ci n'existe pas.

```
any(methods(print) == "print.list")
```

```
## [1] FALSE
```

Il se rabat donc sur la méthode `print` par défaut et évalue l'appel de fonction suivant :

```
print.default(women2)
```

```
## $height
##  [1] 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
##
## $weight
##  [1] 115 117 120 123 126 129 132 135 139 142 146 150 154 159 164
##
## attr(,"row.names")
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

Certaines fonctions génériques ne possèdent pas de méthode par défaut. Par exemple, c'est le cas de la fonction générique `anova`.

```
methods(anova)
```

```
## [1] anova.glm*     anova.glmlist* anova.lm*      anova.lmlist*  anova.loess*
## [6] anova.mlm*     anova.nls*
## see '?methods' for accessing help and source code
```

Si nous donnons en entrée à la fonction `anova` un objet qui n'est pas de classe `glm`, `glmlist`, `lm`, `lmlist`, `loess`, `mlm` ou `nls`, une erreur est retournée.

```
anova(women2)
```

```
## Error in UseMethod("anova") :
##   no applicable method for 'anova' applied to an object of class "list"
```

Qu'arrive-t-il lorsque `class` retourne plus d'une classe ?

Les objets R peuvent posséder plus d'une classe. Par exemple, les objets R retournés par la fonction `aov` ont deux classes : `aov` et `lm`.

```
model <- aov(Sepal.Length ~ Species, data = iris)
attributes(model)
```

```
## $names
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "contrasts"     "xlevels"       "call"          "terms"
## [13] "model"
##
## $class
## [1] "aov" "lm"
```

```
class(model)
```

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

Si un objet possédant plus d'une classe est fourni en entrée à une fonction générique, R cherche d'abord à utiliser la méthode associée à la première classe de la liste. Si celle-ci n'existe pas, R utilise la méthode associée à le seconde classe. Si celle-ci n'existe pas, R continue à parcourir le vecteur des noms de classe jusqu'à ce qu'il trouve une méthode pour une classe. S'il n'existe pas de méthode, pour aucune classe de la liste, c'est la méthode par défaut qui est employée. Et, comme nous venons de le voir, s'il n'y a même pas de méthode par défaut, alors une erreur est générée.

Remarques

Notons que, alors que les listes ont la classe implicite `list`, les arrays ont la classe implicite `array`, les matrices ont la classe implicite `matrix` et les vecteurs ont une classe implicite correspondant au type de données qu'ils contiennent, par exemple :

```
x <- 1:5
x
```

```
## [1] 1 2 3 4 5
```

```
class(x)
```

```
## [1] "integer"
```

Les facteurs, pour leur part, ont un attribut `class`, tout comme les data frames.

```
x_factor <- factor(x)
x_factor
```

```
## [1] 1 2 3 4 5
## Levels: 1 2 3 4 5
```

```
attributes(x_factor)
```

```
## $levels
## [1] "1" "2" "3" "4" "5"
##
## $class
## [1] "factor"
```

```
class(x_factor)
```

```
## [1] "factor"
```

Notons également que la fonction `methods` permet aussi d'énumérer toutes les fonctions génériques possédant une méthode associée à une classe en particulier, par exemple :

```
methods(class = "data.frame")
```

```
##  [1] $             $<-           [             [[            [[<-          [<-
##  [7] aggregate     anyDuplicated as.data.frame as.list       as.matrix     by
## [13] cbind         coerce        dim           dimnames      dimnames<-    droplevels
## [19] duplicated    edit          format        formula       head          initialize
## [25] is.na         Math          merge         na.exclude    na.omit       Ops
## [31] plot          print         prompt        rbind         row.names     row.names<-
## [37] rowsum        show          slotsFromS3   split         split<-       stack
## [43] str           subset        summary       Summary       t             tail
## [49] transform     unique        unstack       within
## see '?methods' for accessing help and source code
```

Création de méthodes et de classes S3

Il est facile de créer de nouvelles méthodes, associées à de nouvelles classes, pour des fonctions génériques existantes (ex. `print`, `summary`, `plot`, `coef`, etc.). Nous ne verrons pas comment créer de nouvelles fonctions génériques, mais plutôt comment créer de nouvelles méthodes (versions) de ces fonctions.

Pour assigner une classe à un objet R, il suffit de l'encadrer d'un appel à la fonction `class`, suivi d'un opérateur d'assignation et du nom de la classe. Voici un exemple.

```r
unObjet <- rnorm(5)
print(unObjet)
```

```
## [1]  0.4616122 -0.8055189 -0.9698873  0.9082013 -0.1793219
```

```r
class(unObjet) <- "nouveauVecteur"
print(unObjet)
```

```
## [1]  0.4616122 -0.8055189 -0.9698873  0.9082013 -0.1793219
## attr(,"class")
## [1] "nouveauVecteur"
```

Maintenant, pour créer une méthode associée à une fonction générique existante pour un objet d'une nouvelle classe, il faut créer une fonction nommée : `nomFonctionGenerique.nomClasse`. Cette fonction comporte typiquement les arguments suivants :

- mêmes arguments que ceux de la fonction générique ;
- l'argument `...`, même s'il n'est pas utilisé ;
- arguments supplémentaires au besoin.

Créons une méthode `print` pour la classe `nouveauVecteur`.

```r
print.nouveauVecteur <- function(x, ..., intro = TRUE){
  if (intro) {
    cat("Voici le vecteur :\n")
  }
  print.default(unclass(x), ...)
  invisible(x)
}
```

```r
print(unObjet)
```

```
## Voici le vecteur :
## [1]  0.4616122 -0.8055189 -0.9698873  0.9082013 -0.1793219
```

```r
print(unObjet, intro = FALSE)
```

```
## [1]  0.4616122 -0.8055189 -0.9698873  0.9082013 -0.1793219
```

Il est courant pour une méthode `print` de terminer par la commande `invisible(x)`. La fonction `invisible` provoque le retour d'une valeur par la fonction, tout comme la fonction `return`. Cependant, contrairement à `return`, `invisible` ne provoque pas un affichage lorsque l'appel de la fonction n'est pas assigné à un nom, évitant ainsi un affichage double suite à une commande comme la suivante :

```r
sortie <- print(unObjet)
```

```
## Voici le vecteur :
## [1]  0.4616122 -0.8055189 -0.9698873  0.9082013 -0.1793219
```

Tentons maintenant de créer une méthode `plot` pour un objet de cette classe. La fonction générique `plot` a deux arguments :

```r
plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x0000000016029a20>
## <environment: namespace:graphics>
```

mais notre méthode ne possédera qu'un seul argument. Elle générera un histogramme.

```r
plot.nouveauVecteur <- function(x, ..., main = "Nouveau vecteur"){
  hist(x, main = main)
}
```

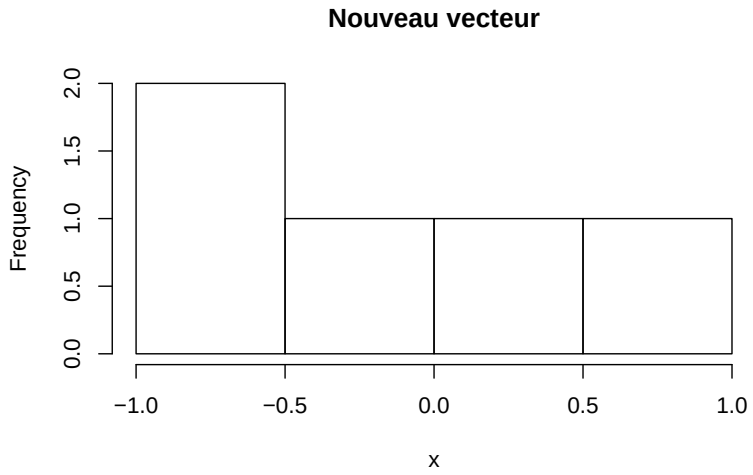
Nous avons maintenant défini deux méthodes pour la classe `nouveauVecteur`.

```r
methods(class = "nouveauVecteur")
```

```
## [1] plot  print
## see '?methods' for accessing help and source code
```

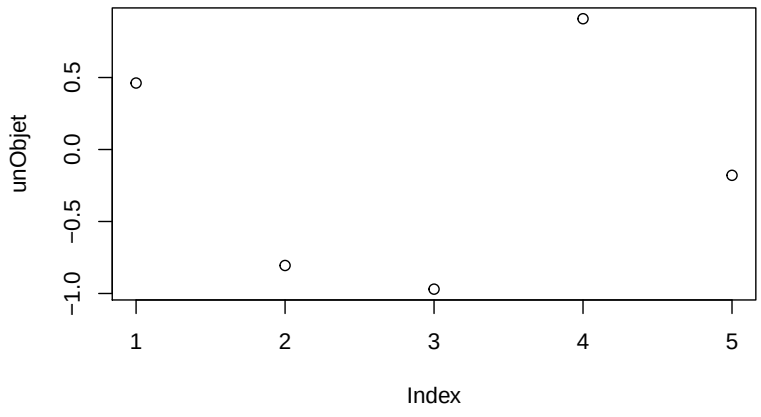
La méthode pour la fonction générique `plot` produit le résultat suivant.

```r
plot(unObjet)
```



Si `unObjet` n'était pas de classe `nouveauVecteur`, nous obtiendrions plutôt ce graphique.

```r
unObjet <- unclass(unObjet)
plot(unObjet)
```



Cet exemple illustre le fait qu'une méthode ne doit pas obligatoirement posséder les mêmes arguments que

la fonction générique associée. De plus, les arguments supplémentaires ne doivent pas être forcément après l'argument `...` . Par exemple, la fonction générique `aggregate` :

```
aggregate
```

```
## function (x, ...)
## UseMethod("aggregate")
## <bytecode: 0x0000000015c023f8>
## <environment: namespace:stats>
```

possède 4 méthodes :

```
methods(aggregate)
```

```
## [1] aggregate.data.frame aggregate.default*   aggregate.formula*   aggregate.ts
## see '?methods' for accessing help and source code
```

Ces méthodes possèdent les arguments suivants :

```
args(aggregate.data.frame)
```

```
## function (x, by, FUN, ..., simplify = TRUE, drop = TRUE)
## NULL
```

```
args(getS3method("aggregate", "default"))
```

```
## function (x, ...)
## NULL
```

```
args(getS3method("aggregate", "formula"))
```

```
## function (formula, data, FUN, ..., subset, na.action = na.omit)
## NULL
```

```
args(aggregate.ts)
```

```
## function (x, nfrequency = 1, FUN = sum, ndeltat = 1, ts.eps = getOption("ts.eps"),
##     ...)
## NULL
```

Une de ces méthodes (`aggregate.formula`) ne possède pas d'argument nommé `x` et l'argument `...` est après l'unique argument de la fonction générique (`x`) seulement dans la méthode par défaut.

Méthode `print` pour formater l'affichage des sorties de nos fonctions

Il est souvent utile de créer des méthodes `print` pour formater l'affichage des sorties de nos fonctions. Pour ce faire, il suffit de compléter les deux étapes suivantes.

1. D'abord, dans le corps de la fonction, il **faut attribuer une nouvelle classe** (souvent le nom de la fonction est utilisé) avec la fonction `class` à l'objet retourné en sortie.
2. Il faut ensuite créer une fonction nommée : `print.nomClasse`.

Pour illustrer ces étapes, créons une méthode `print` pour un objet retourné par la fonction `statDesc` créée dans le notes de cours sur les fonctions. Tout d'abord, attribuons une nouvelle classe à la sortie de `statDesc`.

```
statDesc <- function (x, formatSortie = c("vecteur", "matrice", "liste"), ...) {
  # Calcul
  if (is.numeric(x)) {
    stats <- c(min = min(x, ...), moy = mean(x, ...), max = max(x, ...))
  } else if (is.character(x) || is.factor(x)) {
```

```
    stats <- table(x)
  } else {
    stats <- NA
  }
  # Production de la sortie
  formatSortie <- match.arg(formatSortie)
  if (formatSortie == "matrice"){
    stats <- as.matrix(stats)
    colnames(stats) <- if (is.character(x) || is.factor(x)) "frequence" else "stat"
  } else if (formatSortie == "liste") {
    stats <- as.list(stats)
  }
  out <- list(stats = stats)
  class(out) <- "statDesc"
  out
}
```

Dans cet exemple, en plus de l'instruction `class(out) <- "statDesc"` ajoutée pour attribuer une classe à la sortie de la fonction, l'objet retourné en sortie a été formaté en liste contenant tout ce qu'il y a à retourner (ici un seul objet). Il n'est pas obligatoire qu'une sortie de fonction qui possède une classe soit une liste, mais c'est une pratique très courante.

Maintenant, écrivons le code de notre nouvelle méthode `print`, pour un objet de classe `statDesc`.

```
print.statDesc <- function(x, ...){
  cat("Statistiques descriptives :\n")
  print(x$stats, ...)
  invisible(x)
}
```

Le résultat de la fonction `statDesc` sera maintenant toujours affiché en utilisant la méthode `print.statDesc`.

```
statDesc(x = iris$Species, formatSortie = "matrice")
```

```
## Statistiques descriptives :
##            frequence
## setosa            50
## versicolor        50
## virginica         50
```

Pour une fonction qui retourne une très longue liste, attribuer une classe à sa sortie et écrire une méthode `print` pour cette classe permet d'éviter l'affichage dans la console de la liste entière retournée en sortie.

Notons que dans le corps de la fonction `statDesc`, les instructions

```
  out <- list(stats = stats)
  class(out) <- "statDesc"
  out
```

auraient pu être remplacées par ce qui suit.

```
  structure(list(stats = stats),  class = "statDesc")
```

Système S4

Utilisation de classes S4

Même si nous n'illustrons pas ici comment créer des classes S4, il est bon de savoir comment utiliser ce type de classes qui est assez courant, particulièrement dans les packages distribués sur Bioconductor. Ces classes sont utilisables en R grâce au package `methods`, inclus dans l'installation de base.

Pour illustrer les classes S4, installons le package `sp`, qui exploite ce type de classe.

```
install.packages("sp")
```

Voici un exemple d'utilisation d'une fonction de ce package, tiré d'une fiche d'aide du package.

```
library(sp)
x = c(1,2,3,4,5)
y = c(3,2,5,1,4)
S <- SpatialPoints(cbind(x,y))
S
```

```
## SpatialPoints:
##      x y
## [1,] 1 3
## [2,] 2 2
## [3,] 3 5
## [4,] 4 1
## [5,] 5 4
## Coordinate Reference System (CRS) arguments: NA
```

```
str(S)
```

```
## Formal class 'SpatialPoints' [package "sp"] with 3 slots
##   ..@ coords     : num [1:5, 1:2] 1 2 3 4 5 3 2 5 1 4
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : NULL
##   .. .. ..$ : chr [1:2] "x" "y"
##   ..@ bbox       : num [1:2, 1:2] 1 1 5 5
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:2] "x" "y"
##   .. .. ..$ : chr [1:2] "min" "max"
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
##   .. .. ..@ projargs: chr NA
```

L'objet retourné par la fonction `SpatialPoints` n'est pas une liste. C'est un objet appartenant à une classe S4, définie dans le package `sp`.

```
isS4(S)
```

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

```
class(S)
```

```
## [1] "SpatialPoints"
## attr(,"package")
## [1] "sp"
```

Pour atteindre les éléments dans l'objet, il est possible d'utiliser une méthode conçue à cet effet. Par exemple, la fiche d'aide ouverte par la commande `help("SpatialPoints-class")` nous informe qu'une méthode `coordinates` est définie pour les objets de la classe `"SpatialPoints"`. Nous pouvons aussi énumérer toutes

les fonction génériques possédant une méthode définie pour une certaine classe S4 avec la fonctions `methods` comme suit.

```
methods(class = "SpatialPoints")
```

```
##  [1] $             $<-           [             [[            [[<-          [<-
##  [7] addAttrToGeom as.data.frame bbox          coerce        coordinates   coordinates<-
## [13] coordnames    coordnames<-  dimensions    fullgrid      geometry      geometry<-
## [19] gridded       gridded<-     is.projected  length        merge         over
## [25] plot          points        polygons      print         proj4string   proj4string<-
## [31] rbind         row.names     row.names<-   show          spChFIDs<-    split
## [37] sppanel       spsample      spTransform   summary
## see '?methods' for accessing help and source code
```

En fait, des méthodes S3 et S4 peuvent être définies pour des objets de classe S4. La fonction `methods` retourne les méthodes des deux types. Pour se limiter à un seul type, il faut utiliser les fonctions `.S3methods` et `.S4methods`.

```
.S3methods(class = "SpatialPoints")
```

```
## [1] as.data.frame length        points        print         rbind         row.names
## [7] row.names<-   split
## see '?methods' for accessing help and source code
```

```
.S4methods(class = "SpatialPoints")
```

```
##  [1] $             $<-           [             [[            [[<-          [<-
##  [7] addAttrToGeom bbox          coerce        coordinates   coordinates<- coordnames
## [13] coordnames<-  dimensions    fullgrid      geometry      geometry<-    gridded
## [19] gridded<-     is.projected  merge         over          plot          polygons
## [25] proj4string   proj4string<- show          spChFIDs<-    sppanel       spsample
## [31] spTransform   summary
## see '?methods' for accessing help and source code
```

Une fonction générique dans le système S4 n'a pas la même allure que dans le système S3.

```
coordinates
```

```
## standardGeneric for "coordinates" defined from package "sp"
##
## function (obj, ...)
## standardGeneric("coordinates")
## <environment: 0x0000000018fc4d08>
## Methods may be defined for arguments: obj
## Use  showMethods("coordinates")  for currently available ones.
```

La méthode `coordinates` pour un objet de classe `"SpatialPoints"` extrait l'élément de l'objet `S` nommé `coords`.

```
coordinates(S)
```

```
##      x y
## [1,] 1 3
## [2,] 2 2
## [3,] 3 5
## [4,] 4 1
## [5,] 5 4
```

Nous pouvons accéder à la définition de cette méthode grâce à la fonction `getMethod` comme suit.

```
getMethod(coordinates, signature = "SpatialPoints")
```

```
## Method Definition:
##
## function (obj, ...)
## {
##     .local <- function (obj)
##     obj@coords
##     .local(obj, ...)
## }
## <environment: namespace:sp>
##
## Signatures:
##         obj
## target  "SpatialPoints"
## defined "SpatialPoints"
```

Cette définition n'est pas aussi simple que celle d'une méthode S3. Je ne vais pas l'approfondir ici.

Pour extraire des éléments d'un objet de classe S4, il est aussi possible d'utiliser l'opérateur `@` (et non `$` puisqu'il ne s'agit pas d'une liste).

```
S@coords
```

```
##      x y
## [1,] 1 3
## [2,] 2 2
## [3,] 3 5
## [4,] 4 1
## [5,] 5 4
```

ou encore la fonction `slot`.

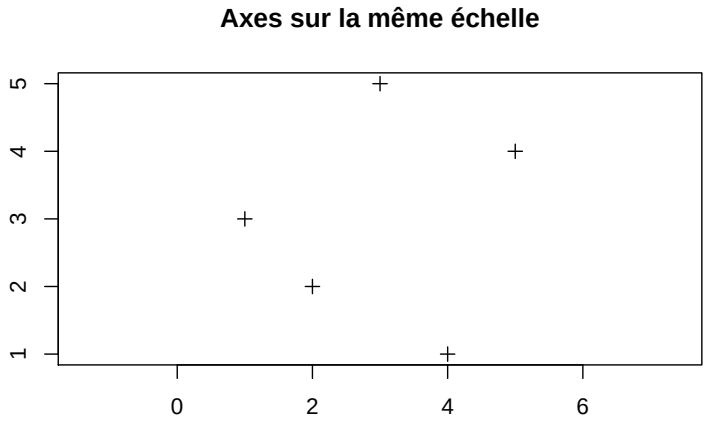
```
slot(S, "coords")
```

```
##      x y
## [1,] 1 3
## [2,] 2 2
## [3,] 3 5
## [4,] 4 1
## [5,] 5 4
```

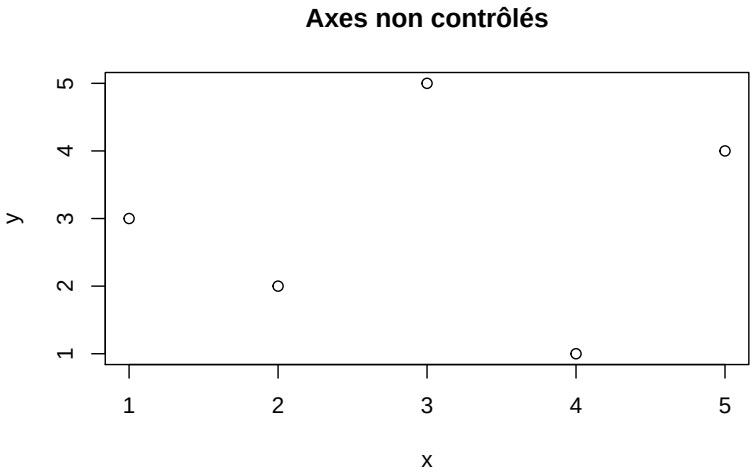
Ainsi, utiliser des classes S4 est simple. Il suffit de d'abord bien identifier qu'il s'agit d'un objet de classe S4. Le texte `Formal class` dans la sortie de `str` nous l'indique. La fonction `isS4` peut aussi nous le confirmer. Ensuite, nous pouvons manipuler les objets avec les fonctions génériques possédants des méthodes pour cette classe et nous pouvons extraire des éléments des objets avec l'opérateur `@` ou la fonction `slot`.

Notons qu'un des intérêts du package `sp` est la production facilitée de graphiques représentant des données spatiales, par exemple des coordonnées géographiques, en s'assurant d'utiliser des axes sur la même échelle.

```
plot(S, main = "Axes sur la même échelle", axes=TRUE)
```




```r
plot(x, y, main = "Axes non contrôlés")
```



Références

- Matloff, N. (2011). *The Art of R Programming : A Tour of Statistical Software Design*. No Starch Press, chapitre 9.
- Wickham, H. (2014). Advanced R. CRC Press. URL de la deuxième édition en développement : https://adv-r.hadley.nz/oo.html