Mécanisme des classes sous R
Modèles S3, S4, RC

Ricco Rakotomalala

http://eric.univ-lyon2.fr/~ricco/cours/cours_programmation_R.html

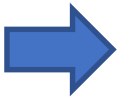
R est un « vrai » langage de programmation.

Le programmeur a la possibilité de créer ses propres classes, pour mieux organiser son code, pour faciliter l'appréhension de son travail par les utilisateurs. Ces structures obéissent aux mécanismes usuels de la programmation orientée objet (POO) (surtout les modèles S4 et RC).

- Meilleure organisation du programme, plus de lisibilité, plus de réutilisabilité,… la maintenance est facilitée.
- Meilleure cohérence des packages : objet(s) + fonctions manipulant ce(s) type(s) d'objet(s).

Traiter un problème de corrélation entre 2 vecteurs, on doit stocker plusieurs informations.

```
#vecteur x
x <- c(18,19,20,21,22,23,24,25,26,27,28,29)

#vecteur y
y <- c(76.1,77,78.1,78.2,78.8,79.7,79.9,81.1,81.2,81.8,82.8,83.5)

#effectif
n <- length(x)

#correlation
cor.p <- cor(x,y)
print(cor.p)

#t pour test de significativité
cor.t <- cor.p/sqrt((1-cor.p^2)/(n-2))
print(cor.t)

#p-value test de significativité
cor.pvalue <- 2.0*pt(abs(cor.t),n-2,lower.tail=F)
print(cor.pvalue)
```



Objectif : rassembler ces informations dans un tout cohérent.

Basées sur une structure de liste

LES CLASSES S3

Démarche : Créer une liste et attribuer un nom de classe à l'objet. C'est fait !

```
#création d'un objet de la classe correlation
objet <- list(x.values=x,y.values=y,size=n,r=cor.p,t=cor.t,pvalue=cor.pvalue)
class(objet) <- "correlation"

#ou encore – c'est equivalent - "class" est terme prédéfini
#attr(objet,"class") <- "correlation"

#vérification
print(objet)

#verification 2
print(class(objet)) # "correlation"

#liste des attributs/propriétés
print(attributes(objet))
```

Avantage : souplesse

Inconvénient : l'absence de formalisme peut être source d'erreurs.

```
> print(attributes(objet))
$names
[1] "x.values" "y.values" "size"
[4] "r"        "t"        "pvalue"

$class
[1] "correlation"
```

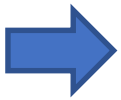
```
$x.values
 [1] 18 19 20 21 22 23 24 25 26 27 28 29

$y.values
 [1] 76.1 77.0 78.1 78.2 78.8 79.7 79.9 81.1 81.2 81.8 82.8 83.5

$size
[1] 12

$r
[1] 0.9943661

$t
[1] 29.66465

$pvalue
[1] 4.428071e-11

attr(,"class")
[1] "correlation"
```

On accède aux propriétés avec $

Exemple : objet$size

Passer par une fonction qui émule un « constructeur » est toujours mieux.

```
#"constructeur" pour classe S3
correler <- function(x,y){
  #on en profite pour introduire un contrôle
  if (length(x) != length(y) || length(x) < 2){
    stop("longueurs des vecteurs incohérentes")
  }
  #création de l'instance
  instance <- list()
  instance$x.values <- x
  instance$y.values <- y
  instance$size <- length(x)
  instance$r <- cor(x,y)
  instance$t <- instance$r/sqrt((1-instance$r^2)/(instance$size-2))
  instance$pvalue <- 2.0*pt(abs(instance$t),instance$size-2,lower.tail=F)
  class(instance) <- "correlation"
  #renvoyer le résultat
  return(instance)
}

#instanciation
obj <- correler(x,y)
print(obj)
```



Ce n'est pas un constructeur au sens usuel de la POO mais la démarche est mieux cadrée. Et on peut introduire une gestion d'erreurs !

Nous pouvons programmer nos fonctions qui manipulent nos objets, mais aussi redéfinir (*surcharger*) des méthodes existantes. Ex. la procédure print()

Méthode générique.nom de classe

```
#liste des définitions de la procédure print
methods(print)

#définir la notre
print.correlation <- function(objet){
  #affichage amélioré
  cat("n = ",objet$size,"\n")
  cat("r = ",objet$r,"\n")
  cat("test statistic = ",objet$t,"\n")
  cat("p-value = ",objet$pvalue,"\n")
}

#nouvelle liste
methods(print)

#affichage
print(obj)
```

```
[67] print.condition
[68] print.connection
[69] print.CRAN_package_reverse_dependencies_and_views*
[70] print.data.frame
[71] print.Date
```

Il y a 199 définitions de la méthode print pour ma configuration.

Définition de la méthode print() pour l'objet de type « correlation ».

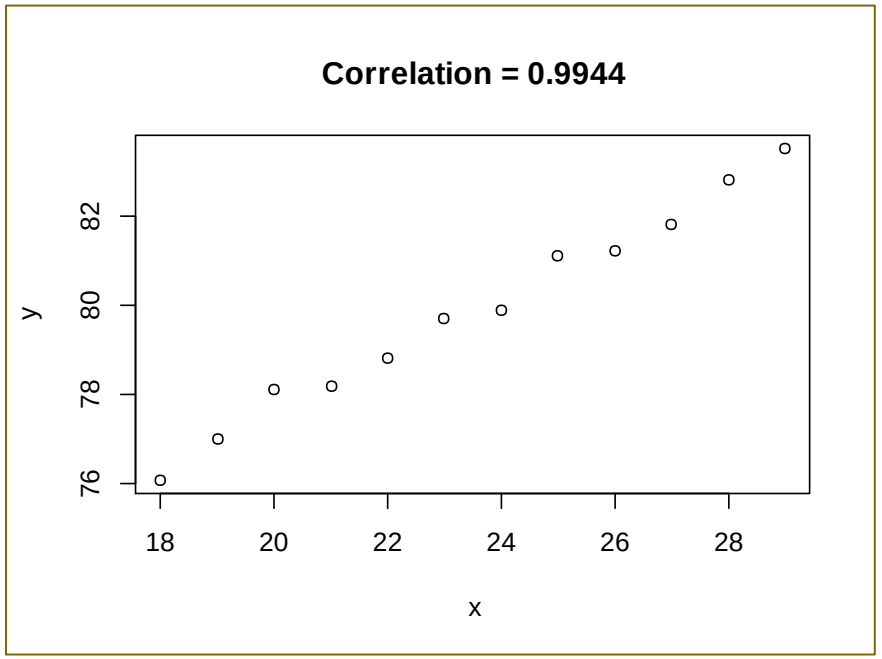
```
[67] print.condition
[68] print.connection
[69] print.correlation
[70] print.CRAN_package_reverse_dependencies_and_views*
[71] print.data.frame
[72] print.Date
```

Il y en a 200 maintenant, print() pour l'objet « correlation » est recensée.

```
> print(obj)
n =  12
r =  0.9943661
test statistic =  29.66465
p-value =  4.428071e-11
```

Ou encore surcharger la procédure générique plot()

```
#surchager la fonction plot pour la classe correlation
plot.correlation <- function(objet){
  plot(objet$x,objet$y,xlab="x",ylab="y",main=paste("Correlation =",round(objet$r,4)))
}

#appel de la méthode plot pour l'objet
plot(objet)
```



Nous pouvons créer une fonction et la rendre générique. UseMethod() se charge de la répartition des appels. Ex. définir une fonction « montrer »

```
#définir une fonction générique “montrer”
montrer <- function(objet){
  UseMethod(generic="montrer")
}

#fonction par défaut de “montrer”
#pour tout objet dont la classe n’est pas recensée
montrer.default <- function(objet){
  cat("Montrer par defaut pour classe = ",
class(objet))
}

#fonction montrer pour l'objet correlation
montrer.correlation <- function(objet){
  cat("Effectif = ",objet$size,"\n")
  cat("Correlation = ", objet$r)
}

#liste des fonctions/classes recensées pour montrer
methods(montrer)

#test pour vecteur numérique
montrer(x)

#test pour objet correlation
montrer(obj)
```

```
> methods(montrer)
[1] montrer.correlation montrer.default
```

```
> montrer(x)
Montrer par defaut pour classe =  numeric
```

```
> montrer(obj)
Effectif =  12
Correlation =  0.9943661
```

Plus de formalisme, plus de rigueur… plus de contraintes

LES CLASSES S4

Comme c'est l'usage pour les autres langages de programmation, il faut tout d'abord définir la structure : énumérer les champs et leurs types (!)

```
#objet corrélation S4
setClass("correlation",
         slots=c(x="numeric",y="numeric",
                         size="numeric",
                         r = "numeric", t="numeric",
                         pvalue="numeric")
)
```

Il est dès lors possible d'instancier un objet de type « correlation » (S4)

```
#instanciation
obj <- new("correlation",x=x,y=y,size=n,r=cor.p,t=cor.t,pvalue=cor.pvalue)

#affichage
print(obj)
```

```
> print(obj)
An object of class "correlation"
Slot "x":
 [1] 18 19 20 21 22 23 24 25 26 27 28 29

Slot "y":
 [1] 76.1 77.0 78.1 78.2 78.8 79.7 79.9 81.1
 [9] 81.2 81.8 82.8 83.5

Slot "size":
[1] 12

Slot "r":
[1] 0.9943661

Slot "t":
[1] 29.66465

Slot "pvalue":
[1] 4.428071e-11
```

Les champs/propriétés sont des « slots ».

A la définition de la classe, spécifier une fonction génératrice (une sorte de constructeur). Elle peut être appelée directement lors de l'instanciation d'un objet.

```
#objet corrélation S4 – version 2
correlation <- setClass("correlation",
                        slots=c(x="numeric",y="numeric",
                                size="numeric",
                                r = "numeric", t="numeric",
                                pvalue="numeric")
)

#instanciation bis
obj_bis <- correlation(x=x,y=y,size=n,r=cor.p,t=cor.t,pvalue=cor.pvalue)
print(obj_bis)
```

```
> print(obj_bis)
An object of class "correlation"
Slot "x":
 [1] 18 19 20 21 22 23 24 25 26 27 28 29

Slot "y":
 [1] 76.1 77.0 78.1 78.2 78.8 79.7 79.9
 [8] 81.1 81.2 81.8 82.8 83.5

Slot "size":
[1] 12

Slot "r":
[1] 0.9943661

Slot "t":
[1] 29.66465

Slot "pvalue":
[1] 4.428071e-11
```

On peut s'affranchir de l'instruction new() (qui est appelée en sous-main en réalité).

L'opérateur @ permet d'accéder aux champs d'un objet de type S4

```
#classe d'un objet
print(class(obj))

#vérification du type S4
print(isS4(obj)) #TRUE

#liste des slots d'une classe
print(getSlots("correlation"))

#liste des slots d'un objet
print(slotNames(obj))

#accès aux champs
#en lecture et en écriture
print(obj@size)

#autre approche pour accès aux champs
print(slot(obj,"size"))
```

```
> #classe d'un objet
> print(class(obj))
[1] "correlation"
attr(,"package")
[1] ".GlobalEnv"
>
> #vérification du type S4
> print(isS4(obj)) #TRUE
[1] TRUE
>
> #liste des slots d'une classe
> print(getSlots("correlation"))
        x         y      size         r
"numeric" "numeric" "numeric" "numeric"
        t    pvalue
"numeric" "numeric"
>
> #liste des slots d'un objet
> print(slotNames(obj))
[1] "x"      "y"      "size"   "r"
[5] "t"      "pvalue"
>
> #accès aux champs
> #en lecture et en écriture
> print(obj@size)
[1] 12
>
> #autre approche
> print(slot(obj,"size"))
[1] 12
```

Classe avec valeurs par défaut et vérification d'intégrité

```
#objet corrélation S4, avec valeurs par défaut et vérification d'intégrité
setClass("correlation",

         #liste des champs
         slots = c(x="numeric",y="numeric",
                   size="numeric",
                   r = "numeric", t="numeric",
                   pvalue="numeric"),

         #valeurs par défaut pour certains champs
         prototype = list(
           x=c(), y=c(),
           size = 0, r = 0, t = 0, pvalue = 1.0
         ),

         #vérification porte sur la longueur des vecteurs
         validity = function(object){
           #test
           if (length(object@x) !=  length(object@y) || length(object@x) < 2){
             return("longueurs des vecteurs incohérentes")
           }
           #validation
           return(TRUE)
         }
)

#test 1
obj1 <- new("correlation",x=x,y=y)
print(obj1)

#test 2
obj2 <- new("correlation",x=0,y=0)
print(obj2)
```

L'objet est instancié, aux champs non précisés à l'initialisation sont attribués les valeurs par défaut.

```
> print(obj1)
An object of class "correlation"
Slot "x":
 [1] 18 19 20 21 22 23 24 25 26 27 28
[12] 29

Slot "y":
 [1] 76.1 77.0 78.1 78.2 78.8 79.7 79.9
 [8] 81.1 81.2 81.8 82.8 83.5

Slot "size":
[1] 0

Slot "r":
[1] 0

Slot "t":
[1] 0

Slot "pvalue":
[1] 1
```

```
> obj2 <- new("correlation",x=0,y=0)
Error in validObject(.Object) :
  objet de classe "correlation" incorrect:
longueurs des vecteurs incohérentes
> print(obj2)
Error in print(obj2) : object 'obj2' not fo
und
```

L'objet n'est même pas instancié.

Associer une méthode à une classe passe par la **définition d'une fonction générique** à laquelle on transmet l'objet en paramètre. L'esprit est identique à celui du modèle S3.

```
#réserver le nom de la function
#en définissant une nouvelle fonction générique
setGeneric(
            #nom de la fonction
            name = "affichage",
            #définition
            def = function(objet){
                standardGeneric("affichage")
            }
)

#associer une fonction "affichage" à la classe
setMethod(
          #nom de la fonction
          f = "affichage",
          #indication de classe associée
          signature = "correlation",
          #écriture de la fonction
          definition = function(objet){
            cat("correlation = ",objet@r,"\n")
            cat("Effectif = ",objet@size)
          }
)

#appel de la fonction
affichage(obj)
```

```
> affichage(obj)
correlation =  0.9943661
Effectif =  12
```

Cette étape fait le lien entre la fonction et la classe !

En respectant la démarche (setMethod), il est possible de surcharger une méthode générique existante [ex. show()], en respectant l'en-tête de la fonction bien sûr.

```
#"surcharger" la fonction show() existante
#pour la classe correlation
setMethod(f = "show",
          signature="correlation",
          definition = function(object){
            cat("--- Show ---\n")
            cat("correlation = ",object@r,"\n")
            cat("Effectif = ",object@size,"\n")
          }
)

#appel
show(obj)
```

Respecter le format c.-à-d. utiliser **object** comme nom de paramètre.

Formal methods for `show` will usually be invoked for automatic printing (see the details).

Usage

```
show(object)
```

Arguments

`object`	Any R object

```
> show(obj)
--- Show ---
correlation =  0.9943661
Effectif =  12
```

Une méthode de classe peut recevoir des paramètres supplémentaires (en plus de l'objet lui-même).

```
#fonction générique - intervalle de confiance
setGeneric(
    name="intervalle",
    #paramètres : objet + niveau de confiance
    def = function(objet,nc=0.95){
      standardGeneric("intervalle")
    }
)

#écriture de la fonction "intervalle"
#en association avec la classe corrélation
setMethod(
  f = "intervalle",
  signature = "correlation",
  definition = function(objet,nc=0.95){
    z <- 0.5*log((1+objet@r)/(1-objet@r))
    ez <- 1.0/sqrt(objet@size-3.0)
    zb <- z - qnorm(nc/2) * ez
    zh <- z + qnorm(nc/2) * ez
    rb <- tanh(zb)
    rh <- tanh(zh)
    return(list(lower=rb,upper=rh))
  }
)

#appel de la fonction
bornes <- intervalle(obj,nc=0.90)
print(bornes)
```

$$z = \frac{1}{2}\ln\frac{1+r}{1-r}$$

$$r = \frac{e^{2z}-1}{e^{2z}+1}$$

```
> bornes <- intervalle(obj,nc=0.90)
> print(bornes)
$lower
[1] 0.9948177

$upper
[1] 0.9938753
```

Comme pour les autres langages objets, il est possible un mécanisme d'héritage pour optimiser la réutilisabilité du code. Ex. introduire la régression simple à partir de la classe corrélation.

```
#défition de la classe héritière - régression
setClass("regression",
         slots = c(sens="character"),
         validity = function(object){
           if (object@sens %in% c("gauche","droite") == FALSE){
             return("mauvais sens de la régression")
           }
         },
         #indiquer la classe ancêtre /!\
         contains = "correlation"
)

#surcharger la méthode show encore
setMethod(
  f = "show",
  signature = "regression",
  definition = function(object){
    #selon le sens de la relation
    if (object@sens == "gauche"){
      res <- lm(x ~ y)
    } else
    {
      res <- lm(y ~ x)
    }
    cat("correlation = ", object@r, "\n") #les champs de l'ancêtre sont bien accessibles /!\
    cat("regression : intercept = ", res$coefficients[1], ",  slope = ",res$coefficients[2])
  }
)

#c'est la méthode show() associée à correlation qui est appelée
obj <- new("correlation",x=x,y=y,size=n,r=cor.p,t=cor.t,pvalue=cor.pvalue)
show(obj)

#c'est la méthode show() associée à régression qui est appelée
reg <- new("regression",x=x,y=y,size=n,r=cor.p,t=cor.t,pvalue=cor.pvalue,sens="gauche")
show(reg)
```

```
> show(obj)
--- Show ---
correlation =  0.9943661
Effectif =  12
```

```
> show(reg)
correlation =  0.9943661
regression : intercept =  -100.842 ,  slope =  1.557194
```

Se rapprocher vraiment de la POO, mais changements forts des habitudes R

REFERENCE CLASSES (RC OU R5)

```
#déclaration de classe RC
correlation <- setRefClass("correlation",
                    #liste des champs
                    fields=list(x="numeric", y="numeric",
                                size="numeric",
                                r = "numeric", t="numeric",
                                pvalue="numeric")
)
```

Comme pour S4, il faut tout d'abord définir la structure (énumérer les champs et leurs types) avec setRefClass

```
#instanciation - appel de la méthode new()
obj <- correlation$new(x=x,y=y,size=n,r=cor.p,t=cor.t,pvalue=cor.pvalue)
print(obj)
```

L'instanciation passe par l'appel de la méthode new() associée à la classe

```
> print(obj)
Reference class object of class "correlation"
Field "x":
 [1] 18 19 20 21 22 23 24 25 26 27 28 29
Field "y":
 [1] 76.1 77.0 78.1 78.2 78.8 79.7 79.9 81.1 81.2 81.8 82.8
[12] 83.5
Field "size":
[1] 12
Field "r":
[1] 0.9943661
Field "t":
[1] 29.66465
Field "pvalue":
[1] 4.428071e-11
```

L'opérateur $ permet d'accéder aux champs d'un objet de type RC

```
#classe
class(obj)

#accès à un champ avec $ (lecture et écriture)
obj$size

#autre solution
#obj$field("size")

#liste des champs de la classe
#on observe des méthodes pré-programmées
correlation

#on peut obtenir la même liste pour l'objet
obj$getClass()

#accès aux méthodes avec $ aussi
obj$show()
```

```
> #classe
> class(obj)
[1] "correlation"
attr(,"package")
[1] ".GlobalEnv"
> #accès à un champ avec $ (en lecture et écriture)
> obj$size
[1] 12
> #liste des champs de la classe
> #on observe des méthodes par défaut pré-programmées
> correlation
Generator for class "correlation":

Class fields:

Name:        x       y    size       r       t  pvalue
Class: numeric numeric numeric numeric numeric numeric

Class Methods:
     "field", "trace", "getRefClass", "initFields", "copy",
     "callSuper", ".objectPackage", "export", "untrace",
     "getClass", "show", "usingMethods", ".objectParent",
     "import"

Reference Superclasses:
     "envRefClass"

> #on peut obtenir la même liste pour l'objet
> obj$getClass()
Reference Class "correlation":

Class fields:

Name:        x       y    size       r       t  pvalue
Class: numeric numeric numeric numeric numeric numeric

Class Methods:
     "field", "trace", "getRefClass", "initFields", "copy",
     "callSuper", ".objectPackage", "export", "untrace",
     "getClass", "show", "usingMethods", ".objectParent",
     "import"

Reference Superclasses:
     "envRefClass"

> #accès aux méthodes  avec $
> obj$show()
Reference class object of class "correlation"
Field "x":
 [1] 18 19 20 21 22 23 24 25 26 27 28 29
Field "y":
 [1] 76.1 77.0 78.1 78.2 78.8 79.7 79.9 81.1 81.2 81.8 82.8
[12] 83.5
Field "size":
[1] 12
Field "r":
[1] 0.9943661
Field "t":
[1] 29.66465
Field "pvalue":
[1] 4.428071e-11
```

Un objet RC est une référence, une affectation revient à faire « pointer » deux objets sur le même espace mémoire.

```
#classe personne
personne <- setRefClass("personne",
              field = list(nom="character",
                           age="numeric"))

#toto
toto <- personne$new(nom="toto",age=25)

#print toto
print(toto)

#tata
tata <- toto

#modifier le nom de tata
tata$nom <- "tata"

#affichage tata
print(tata)

#mais toto aussi est modifié
print(toto)
```

```
> #toto
> toto <- personne$new(nom="toto",age=25)
>
> #print toto
> print(toto)
Reference class object of class "personne"
Field "nom":
[1] "toto"
Field "age":
[1] 25
>
> #tata
> tata <- toto
>
> #modifier le nom de tata
> tata$nom <- "tata"
>
> #affichage tata
> print(tata)
Reference class object of class "personne"
Field "nom":
[1] "tata"
Field "age":
[1] 25
>
> #mais toto aussi est modifié
> print(toto)
Reference class object of class "personne"
Field "nom":
[1] "tata"
Field "age":
[1] 25
```

argh...

Pour obtenir une nouvelle version de l'objet, il faut le dupliquer explicitement. On a bien une nouvelle référence.

```
#toto
toto <- personne$new(nom="toto",age=32)

#print toto
print(toto)

#tata est une nouvelle référence
#le contenu de toto est copié
tata <- toto$copy()

#modifier le nom de tata
tata$nom <- "tata"

#affichage tata
print(tata)

#toto n'est pas modifié !
print(toto)
```

```
> #toto
> toto <- personne$new(nom="toto",age=32)
>
> #print toto
> print(toto)
Reference class object of class "personne"
Field "nom":
[1] "toto"
Field "age":
[1] 32
>
> #tata est une nouvelle référence
> #le contenu de toto est copié
> tata <- toto$copy()
>
> #modifier le nom de tata
> tata$nom <- "tata"
>
> #affichage tata
> print(tata)
Reference class object of class "personne"
Field "nom":
[1] "tata"
Field "age":
[1] 32
>
> #toto n'est pas modifié
> print(toto)
Reference class object of class "personne"
Field "nom":
[1] "toto"
Field "age":
[1] 32
```

toto et tata « pointent » sur deux objets différents.

Programmation des méthodes : nouveaux membres de la classe.

```
#déclaration de classe RC avec une méthode supplémentaire
correlation <- setRefClass("correlation",
                           #liste des champs
                           fields = list(x="numeric", y="numeric",
                                         size="numeric",
                                         r = "numeric", t="numeric",
                                         pvalue="numeric"),
                           methods = list(
                             #test unilatéral
                             test_unilateral = function(sens){
                               if (sens=="gauche"){
                                 return(pt(t,size-2,lower.tail=TRUE))
                               } else
                               {
                                 return(pt(t,size-2,lower.tail=FALSE))
                               }
                             }
                           )
)

#instanciation
obj <- correlation$new(x=x,y=y,size=n,r=cor.p,t=cor.t,pvalue=cor.pvalue)

#appel de la méthode test unilateral
obj$test_unilateral(sens="gauche")
```

L'accès aux champs n'est pas très top. Avec le modèle R6 (à voir), il sera possible de préciser self$champ (pour les propriétés publiques) ou private$champ (propriétés privées).

Modifier les valeurs d'un champ dans une méthode requiert un opérateur d'affectation particulier <<-

```
#classe personne
personne <- setRefClass("personne",
            field =
            list(nom="character",
                 age="numeric"),
            methods =
            list(
                 #ajouter une valeur à age
                 ajouter = function(valeur){
                     age <<- age + valeur
                 }
             )
)

#toto
toto <- personne$new(nom="toto",age=25)
print(toto)

#ajouter 5 ans à toto
toto$ajouter(valeur=5)
print(toto)
```

```
> #toto
> toto <- personne$new(nom="toto",age=25)
> print(toto)
Reference class object of class "personne"
Field "nom":
[1] "toto"
Field "age":
[1] 25
>
> #ajouter 5 ans à toto
> toto$ajouter(valeur=5)
> print(toto)
Reference class object of class "personne"
Field "nom":
[1] "toto"
Field "age":
[1] 30
```

Si on utilise <- ou = pour l'affectation, une nouvelle variable locale est créée (source d'erreur ça…).

```r
#classe personne
personne <- setRefClass("personne",
                        field = list(nom="character",
                                     age="numeric"),
                        methods = list(
                        #ajouter une valeur à age
                        ajouter = function(valeur){
                          age <<- age + valeur
                        }
                        )
)

#classe etudiant
etudiant <- setRefClass("etudiant",
                        field = list(niveau="numeric"),
                        methods = list(
                          #élever le niveau
                          elever = function(x){
                            niveau <<- niveau + x
                          }
                        ),
                         #indication de la classe ancêtre
                        contains = "personne"
)

#un étudiant
titi <- etudiant$new(nom="titi",age=24,niveau=5)

#ajouter âge
titi$ajouter(5)
print(titi)

#élever le niveau
titi$elever(3)
print(titi)
```

Le mécanisme d'héritage existe également pour le modèle RC.

Les champs et méthodes de la classe ancêtre sont exploitables dans la classe héritière.

```
> #un étudiant
> titi <- etudiant$new(nom="titi",age=24,niveau=5)
> print(titi)
Reference class object of class "etudiant"
Field "nom":
[1] "titi"
Field "age":
[1] 24
Field "niveau":
[1] 5
>
> #ajouter âge
> titi$ajouter(5)
> print(titi)
Reference class object of class "etudiant"
Field "nom":
[1] "titi"
Field "age":
[1] 29
Field "niveau":
[1] 5
>
> #élever le niveau
> titi$elever(3)
> print(titi)
Reference class object of class "etudiant"
Field "nom":
[1] "titi"
Field "age":
[1] 29
Field "niveau":
[1] 8
```

CONCLUSION

Le mécanisme des classes nous permet de programmer plus proprement, et par conséquent plus efficacement, sous R.

Le modèle **S3** est très simple mais laisse la porte ouverte à toutes les fantaisies.

Le modèle **S4** est certainement plus rigoureux, mais introduit des mécanismes assez déroutants et/ou contraignants (@ pour accéder aux champs, définir obligatoirement une fonction générique pour programmer une méthode,…).

Le modèle **RC** (reference classes) est celui qui se rapproche le plus de la POO traditionnelle. Mais les mécanismes – pourtant usuels dans les autres langages – peuvent désarçonner les afficionados de R (objet = pointeur, méthode = membre de la classe). Le modèle évolue (voir modèle de classe R6, de plus en plus conforme à la POO, portée des champs et méthodes, accès aux champs dans les méthodes avec self$…)…

Au final, le modèle S3 a tellement pris d'espace que changer les habitudes reste difficile. Il reste incontournable aujourd'hui.

PROGRAMIZ, “R Object and Class”, in R Tutorial,
https://www.programiz.com/r-programming (excellent)

K. Black, “Object Oriented Programming”, in R Tutorial,
http://www.cyclismo.org/tutorial/R/index.html

H. Wickham, “OO field Guide”, in Advanced R, http://adv-r.had.co.nz/

De la documentation à profusion (**n'achetez jamais des livres sur R**)

Site du cours

http://eric.univ-lyon2.fr/~ricco/cours/cours_programmation_R.html

Programmation R

http://www.duclert.org/

Quick-R

http://www.statmethods.net/

POLLS (Kdnuggets)

Data Mining / Analytics Tools Used

(R, 2nd ou 1er depuis 2010)

What languages you used for data mining / data analysis?

http://www.kdnuggets.com/polls/2013/languages-analytics-data-mining-data-science.html

(Août 2013, langage R en 1ère position)

Article New York Times (Janvier 2009)

"**Data Analysts Captivated by R's Power**" - http://www.nytimes.com/2009/01/07/technology/business-computing/07program.html?_r=1