# Petit Manuel de S4

***Programmation Orienté Objet sous R***

**Christophe Genolini**

# Qu'allez-vous trouver dans ce manuel ?

Ce livre est un guide à la programmation objet sous R (ou S4). Il ne nécessite pas de connaître la programmation orientée objet. Par contre, un minimum de connaissances sur R et sur la programmation en général est requis. Pour ceux qui sont totalement débutants en R, nous vous recommandons "R pour les Débutants", d'Emmanuel Paradis [4].

Le présent manuel est découpé en quatre parties. Après une introduction générale sur la programmation objet (section 1) et une définition un peu plus formelle (section 2), un exemple est présenté (section 3). Grave problème que celui du choix de l'exemple : les exemples réels sont trop compliqués. Les artificiels sont inintéressants. Nous allons donc utiliser une simplification d'un problème réel ayant donné lieu à la construction d'un package (package `kml`). Il nous accompagnera tout au long de ce livre pour finir sous forme d'un package appelé `miniKml` disponible sur le site du CRAN.

La deuxième partie présente les bases de la programmation objet. Chaque élément nouvellement introduit est ajouté aux précédents et permet au package de prendre corps. C'est là que sont présentés les éléments fondamentaux de la programmation S4. Cette partie est simple, rédigée pour les débutants, mais indispensable pour comprendre la programmation objet en S. Les sections 4, 5, 6, 7 doivent se lire préférentiellement dans l'ordre.

Dans la troisième partie sont abordés des sujets plus subtils, comme les signatures complexes (section 8), l'héritable (section 9) ou la super assignation pour modification interne d'un objet (section 10). Elles ne sont pas indispensables à la programmation objet, on peut lire l'une sans l'autre et surtout les omettre en première lecture. Cela dépend ensuite des besoins de chacun.

En espérant que ce livre fera naitre de nombreux package S4,
bonne lecture !

# Table des matières

**I Préliminaires** **9**

**1 Introduction** **11**
1.1 Qu'est-ce que "S4" ? . . . 11
1.2 Qu'est-ce que la programmation objet ? . . . 11
1.3 Pourquoi faire de l'objet ? . . . 11
1.3.1 Programmation classique . . . 12
1.3.2 Programmation objet . . . 13
1.4 Pour résumer . . . 15
1.5 Devez-vous vous mettre à l'objet ? . . . 15
1.6 Le côté obscur de la programmation . . . 16
1.6.1 D'accord, pas d'accord . . . 16
1.6.2 Les fusées, la bourse, les bugs... et les coupables ! . . . 16
1.6.3 R, langage propre ? . . . 17

**2 Généralités sur les objets** **19**
2.1 Définition formelle . . . 19
2.1.1 Les attributs . . . 19
2.1.2 Les méthodes . . . 19
2.2 Dessiner, c'est gagner ! . . . 20

**3 Exemple de travail** **21**
3.1 Analyse du problème . . . 22
3.2 L'objet `Trajectoires` . . . 22
3.3 L'objet `Partition` . . . 23
3.4 L'objet `TrajDecoupees` . . . 23
3.5 Plan d'analyse . . . 24
3.6 Application à R . . . 25

**II Les bases de l'objet** **27**

**4 Déclaration des classes** **29**
4.1 Définition des attributs . . . 29

4.2 Constructeur par défaut . . . 30
4.3 Accéder aux attributs . . . 31
4.4 Valeurs par défaut . . . 32
4.5 Supprimer un objet . . . 32
4.6 L'objet vide . . . 33
4.7 Voir l'objet . . . 34

**5 Les méthodes 35**
5.1 `setMethod` . . . 35
5.2 `show` et `print` . . . 37
5.3 `setGeneric` . . . 39
5.3.1 Générique versus Spécifique . . . 39
5.3.2 Définition formelle . . . 40
5.3.3 `lockBinding` . . . 40
5.3.4 Déclaration des génériques . . . 41
5.4 Voir les méthodes . . . 41

**6 Construction 43**
6.1 Vérificateur . . . 43
6.2 L'initiateur . . . 45
6.3 Constructeur grand public . . . 48
6.4 Petit bilan . . . 50

**7 Accesseur 53**
7.1 Les getteurs . . . 53
7.2 Les setteurs . . . 55
7.3 Les opérateurs `[` et `[<-` . . . 56
7.4 `[`, `@` ou get ? . . . 58

**III Pour aller plus loin 59**

**8 Méthodes utilisant plusieurs arguments 63**
8.1 Le problème . . . 63
8.2 `signature` . . . 64
8.3 `missing` . . . 67
8.4 Nombre d'arguments d'une signature . . . 68
8.5 `ANY` . . . 68

**9 Héritage 69**
9.1 Principe . . . 69
9.2 Père, grand-père et `ANY` . . . 70
9.3 `contains` . . . 70
9.4 `unclass` . . . 71

9.5 Arbre d'héritage . . . . . 72
9.6 Voir la méthode en autorisant l'héritage . . . . . 72
9.7 `callNextMethod` . . . . . 75
9.8 `is`, `as` et `as<-` . . . . . 77
9.9 `setIs` . . . . . 78
9.10 Les classes virtuelles . . . . . 80
9.11 Pour les dyslexiques... . . . . 82

**10 Modification interne d'un objet** **83**
10.1 Fonctionnement interne de R : les `environnements` . . . . . 83
10.2 Méthode pour modifier un attribut . . . . . 84

**IV Annexes** **87**

**A Remerciements** **89**
A.1 Nous vivons une époque formidable . . . . . 89
A.2 Ceux par qui ce tutorial existe... . . . . 89

**B Mémo** **91**
B.1 Création . . . . . 91
B.2 Validation . . . . . 91
B.3 Accesseur . . . . . 92
B.4 Méthodes . . . . . 92
B.5 Quelques fonctions incontournables . . . . . 92
B.6 Voir la structure des objets . . . . . 92

**C Pour aller plus loin** **95**

**Liste des tables et figures** **96**

**Bibliographie** **97**

**Index** **98**

# Première partie

# Préliminaires

# Chapitre 1

# Introduction

## 1.1 Qu'est-ce que "S4" ?

S4 est la quatrième version de S. S est un langage qui a deux implémentations : S-plus est commerciale, R est gratuite. La particularité de S4 par rapport au S3 est l'apparition de fonctions qui permettent de considérer S-plus comme un langage objet [1]. Par extension, S4 désigne la programmation orientée objet sous S. Et donc sous R et sous S-plus.

## 1.2 Qu'est-ce que la programmation objet ?

Un *objet* est un ensemble de variables et de fonctions qui concernent toutes le même thème : l'objet lui-même. Pas très clair ? Prenons un exemple : un objet `image` contiendra :

- Les variables qui permettent de définir une image (comme la taille de l'image, son mode de compression, l'image proprement dite) ;
- les fonctions utilisées pour la manipulation de l'image (comme `noirEtBlanc()` ou `redimentionner()`).

Si vous êtes complètement débutant en objet et que tout ça n'est pas limpide, ne vous inquiétez pas, de nombreux exemples vont suivre.

## 1.3 Pourquoi faire de l'objet ?

Pour le néophyte, la programmation objet est quelque chose de lourd et les avantages ne semblent pas évidents : il faut penser son programme à l'avance, modéliser le problème, choisir ses types, penser aux liens qui lieront les objets entre eux... Que des inconvénients. D'où la question légitime : pourquoi faire de l'objet ?

1. *permettent de considérer* et non pas *transforment en*. En tout état de cause, R N'est PAS un langage objet, il reste un langage interprété classique avec une surcouche possible. A quand R++ ?

### 1.3.1 Programmation classique

Le plus simple est de prendre un exemple et de comparer la programmation classique à la programmation objet. L'IMC, l'Indice de Masse Corporelle est une mesure de maigreur ou d'obésité. On le calcule en divisant le poids (en kilo) par la taille au carré. Ensuite, on conclut :

- **20 < IMC < 25** : tout roule pour vous
- **25 < IMC < 30** : Nounours
- **30 < IMC < 40** : Nounours confortable
- **40 < IMC** : Méga nounours, avec effet double douceur, mais qui devrait tout de même aller voir un médecin très vite...
- **18 < IMC < 20** : Barbie
- **16 < IMC < 18** : Barbie mannequin
- **IMC < 16** : Barbie squelette, même diagnostic que le Méga nounours, attention danger...

On veut donc calculer l'IMC. En programmation classique, rien de plus simple :

```
> ### Programmation classique, IMC
> poids <- 85
> taille <- 1.84
> (IMC <- poids / taille^2)
```

```
[1] 25.10633
```

Jusqu'à là, rien de très mystérieux. Si vous voulez calculer pour deux personnes `Moi` et `Elle`, vous aurez

```
> ### Programmation classique, mon IMC
> poidsMoi <- 85
> tailleMoi <- 1.84
> (IMCmoi <- poidsMoi / tailleMoi^2)
```

```
[1] 25.10633
```

```
> ### Programmation classique, son IMC
> poidsElle <- 62
> tailleElle <- 1.60
> (IMCelle <- poidsMoi / tailleElle^2)
```

```
[1] 33.20312
```

Ça marche... sauf qu'`Elle` est qualifiée de "Nounours confortable" (ou "Nounoursette" dans son cas) alors que son poids ne parait pas spécialement excessif. Un petit coup d'oeil au code révèle assez vite une erreur : le calcul de `IMCelle` est faux, nous avons divisé `poidsMoi` par `tailleElle` au lieu de `poidsElle` par `tailleElle`. Naturellement, R n'a pas détecté d'erreur : de son point de vue, il a juste effectué une division entre deux `numeric`.

### 1.3.2 Programmation objet

. En langage objet, la démarche est tout autre. Il faut commencer par définir un objet `IMC` qui contiendra deux valeurs, `poids` et `taille`. Ensuite, il faut définir la fonction `print` qui affichera l'IMC [2]

```
> ### Définition d'un objet IMC
> setClass("IMC",representation(poids="numeric",taille="numeric"))
```

```
[1] "IMC"
```

```
> setMethod("show","IMC",
+     function(object){
+         cat("IMC =",object@poids/(object@taille^2),"\n")
+     }
+ )
```

```
[1] "show"
```

Le code équivalent à ce qui a été fait dans la section 1.3.1 page 12 est alors :

```
> ### Création d'un objet pour moi, et affichage de mon IMC
> (monIMC <- new("IMC",poids=85,taille=1.84))
```

```
IMC = 25.10633
```

```
> ### Création d'un objet pour elle, et affichage de son IMC
> (sonIMC <- new("IMC",poids=62,taille=1.60))
```

```
IMC = 24.21875
```

Á partir du moment où l'initialisation est correcte (problème qui se posait également dans la programmation classique), il n'y a plus d'erreur possible. Là est toute la force de l'objet : la conception même du programme interdit un certain nombre de bug.

**Typage :** l'objet protège également des erreurs de typage, c'est à dire les erreurs qui consistent à utiliser un type là où il faudrait en utiliser un autre. Une erreur de typage, c'est un peu comme additionner des pommes et des kilomètres au lieu d'additionner des pommes et des pommes.

Concrètement, une variable nommée `poids` est conçue pour contenir un poids, c'est à dire un nombre, et non une chaîne de caractères. En programmation classique (pas de typage), `poids` peut contenir n'importe quoi :

```
> ### Programmation classique, pas de typage
> (poids  <- "Bonjour")
```

```
[1] "Bonjour"
```

2. Pour que notre exemple illustratif soit reproductible immédiatement, nous avons besoin de définir ici un objet. Nous le faisons sans explication, mais naturellement, tout cela sera repris ensuite avec moults détails.

En programmation objet, affecter `"bonjour"` à une variable qui devrait contenir un nombre provoquera une erreur :

```
> new("IMC",poids="Bonjour",taille=1.84)
```

```
Error in validObject(.Object) :
  invalid class "IMC" object:
invalid object for slot "poids" in class "IMC": got class
"character", should be or extend class "numeric"
```

**Vérificateurs :** L'objet permet de construire des vérificateurs de cohérence pour, par exemple, interdire certaines valeurs. En programmation classique, une taille peut être négative :

```
> ### Programmation classique, pas de contrôle
> (TailleMoi <- -1.84)
```

```
[1] -1.84
```

En programmation objet, on peut préciser que les tailles négatives n'existent pas :

```
> ### Programmation objet, contrôle
> setValidity("IMC",
+     function(object){
+         if(object@taille<0){
+             return("Taille négative")
+         }else{
+             return(TRUE)
+         }
+     }
+ )
```

```
Class "IMC" [in ".GlobalEnv"]

Slots:

Name:    poids  taille
Class: numeric numeric
```

```
> new("IMC",poids=85,taille=-1.84)
```

```
Error in validObject(.Object) :
  invalid class "IMC" object:
Taille négative
```

**Héritage :** la programmation objet permet de définir un objet comme *héritier* des propriétés d'un autre objet, devenant ainsi son *fils*. L'objet fils bénéficie ainsi de tout ce qui existe pour l'objet . Par exemple, on souhaite affiner un peu nos diagnostics en fonction du sexe de la personne. Nous allons donc définir un nouvel objet, `IMCplus`, qui contiendra trois valeurs : `poids`, `taille` et `sexe`. Les deux premières variables sont les

mêmes que celle de l'objet `IMC`. Nous allons donc définir l'objet `IMCplus` comme héritier de l'objet `IMC`. Il pourra ainsi bénéficier de la fonction `show` telle que nous l'avons définie pour `IMC` et cela, sans aucun travail supplémentaire, puisque `IMCplus` *hérite* de `IMC` :

```
> ### Définition de l'héritier
> setClass("IMCplus",
+     representation(sexe="character"),
+     contains="IMC"
+ )
```

```
[1] "IMCplus"
```

```
> ### Création d'un objet
> lui <- new("IMCplus",taille=1.76,poids=84,sexe="Homme")
> ### Affichage qui utilise ce qui a été défini pour `IMC'
> lui
```

```
IMC = 27.11777
```

La puissance de cette caractéristique apparaitra plus clairement section 9 page 69.

**Encapsulation :** enfin, la programmation objet permet de définir tous les outils composant un objet et de les enfermer dans une boite, puis de ne plus avoir à s'en occuper. Cela s'appelle *l'encapsulation*. Les voitures fournissent un bon exemple d'encapsulation : une fois le capot refermé, on n'a plus besoin de connaître les détails de la mécanique pour rouler. De même, une fois l'objet terminé et refermé, un utilisateur n'a pas à s'inquiéter de son fonctionnement interne. Mieux, concernant les voitures, il n'est pas possible de se tromper et de mettre de l'essence dans le radiateur puisque le radiateur n'est pas accessible. De même, l'encapsulation permet de protéger ce qui doit l'être en ne laissant accessible que ce qui ne risque rien.

## 1.4 Pour résumer

La programmation objet "force" le programmeur à avoir une réflexion préliminaire. On a moins la possibilité de programmer "à la va-vite" ; un plan de programme est quasi indispensable. En particulier :

- Les objets doivent être déclarés et typés.
- Des mécanismes de contrôle permettent de vérifier la cohérence interne des objets.
- Un objet peut hériter des propriétés qui ont été définies pour un autre objet.
- Enfin, l'objet permet une encapsulation du programme : une fois qu'on a défini un objet et les fonctions qui s'y rattachent, on n'a plus besoin de s'occuper de la cuisine interne de l'objet.

## 1.5 Devez-vous vous mettre à l'objet ?

Nous venons de le voir, la programmation objet a des avantages, mais elle a aussi des inconvénients. En premier lieu, elle n'est pas facile à apprendre et on trouve assez

peu de tutoriaux disponibles (c'est d'ailleurs ce vide qui a donné naissance au présent ouvrage). Ensuite, la mise en route d'un projet est passablement plus lourde à gérer qu'en programmation classique où tout peut se faire petit à petit. Se pose donc la légitime question : doit-on ou ne doit-on pas faire de l'objet ?

- Pour la petite programmation de tous les jours, un nettoyage des données, une analyse univariée/bivariée, une régression, tous les outils existent déjà, pas besoin de S4.
- Pour les projets un peu plus importants, quand vous travaillez sur un nouveau concept, sur des données complexes dépassant le simple cadre du `numeric` et `factor`, alors l'objet devient une force : plus difficile au départ, mais bien moins buggué et plus facilement manipulable à l'arrivée.
- Enfin, la construction d'un package devrait toujours être réfléchie et structurée. Dans ce cadre, la programmation objet est conseillée.

## 1.6 Le côté obscur de la programmation

Pour terminer avec cette longue introduction, une petite digression.

### 1.6.1 D'accord, pas d'accord

Vous êtes en train de lire un manuel de programmation objet. Vous êtes en train d'apprendre une nouvelle méthode. Sachez qu'il n'existe pas "une" mais "des" manières de programmer : à ce sujet, les informaticiens ne sont pas toujours d'accord entre eux.

Ce livre suit une certaine vision, celle de son auteur. Des gens très compétents sont d'accord, d'autres le sont moins... Y a-t-il une vérité et si oui, où se trouve-t-elle ? Mystère. Néanmoins, pour essayer de donner une vision la plus large possible, il arrivera que ce livre vous présente plusieurs points de vue, celui de l'auteur, mais aussi celui de lecteurs qui ont réagi à ce qu'ils lisaient et qui ont fait valoir une autre vision des choses.

Ainsi, lecteur avisé, vous aurez tous les éléments en main,
vous pourrez peser le pour et le contre
et vous choisirez votre voie en toute liberté,
à la lumière de la connaissance...

### 1.6.2 Les fusées, la bourse, les bugs... et les coupables !

Quand les hommes ont commencé à envoyer des fusées dans l'espace [3] et qu'elles ont explosé en plein vol, ils ont écrasé une petite larme et ont cherché les causes de l'échec. Comme il fallait bien brûler quelqu'un, ils ont cherché un coupable. Et ils ont trouvé... les informaticiens. "C'est pas d'not' faute, ont déclaré les informaticiens tout marris, c'est un fait avéré intrinsèque aux ordinateurs : tous les programmes sont buggués !" Sauf que dans le cas présent, la facture du bug était plutôt salée...

3. Ou quand ils ont confié la bourse, les hôpitaux, les fiches de paye...aux ordinateurs.

Des gens très forts et très intelligents ont donc cherché des moyens de rendre la programmation moins bugguée. Ils ont fabriqué des nouveaux langages et défini des règles de programmation. On appelle ça la *programmation propre* ou les *bonnes pratiques*. Vous trouverez une proposition de bonnes pratiques en annexe section **??** page **??**. Certaines opérations, certaines pratiques, certaines manières de programmer sont donc qualifiées de *bonnes*, *belles* ou *propres* ; d'autres au rebours sont qualifiés de *mauvaises*, *dangereuses*, *impropres* voire *sales* (Le terme exact consacré par les informaticiens est *crade*...) Il ne s'agit pas là de jugements de valeur, ce sont simplement des qualificatifs qui indiquent que ce type de programmation favorise l'apparition d'erreurs dans le code. Donc à éviter d'urgence.

### 1.6.3 R, langage propre ?

R n'est pas un langage très propre. Quand un informaticien veut ajouter un package, il est libre de faire à peut près ce qu'il veut (ce qui est une grande force) y compris la définition d'instructions "surprenantes" (ce qui est une faiblesse). Une instruction "surprenante" est une instruction dont le fonctionnement est contraire à l'intuition. Par exemple, sachant que `numeric()` désigne un `numeric` vide et que `integer()` est en entier vide, comment désigner une matrice vide ? Toute définition autre que `matrix()` serait surprenante. De nombreuses surprises existent dans R... (Voir la section 4.6 page 33 pour plus de détails sur les objets vides). R permet donc de faire tout un tas d'opérations dangereuses. Il sera néanmoins parfois possible de le rendre propre en nous auto-interdisant des manipulations. Mais ça ne sera pas toujours le cas, nous serons amenés à utiliser des outils impropres. En tout état de cause, les passages dangereux seront signalés par le petit logo qui orne le début de ce paragraphe.

Mais tout le monde ne partage pas ce point de vue : *"On a néanmoins besoin des langages ultra-flexibles et peu contraints*, précise un relecteur, *parce qu'ils rendent certaines applications faciles à programmer alors que les langages propres ne les gèrent pas ou mal. Dans ce genre de cas, utiliser un langage propre revient un peu à prendre un marteau-piqueur pour écraser une noix. En particulier, écrire le coeur d'un programme en C (au lieu de R) peut accroitre la rapidité d'exécution par un facteur 10 ou 20."*

C'est vrai. Mais tout dépend du niveau du programmeur : à haut niveau, pour ceux qui ne font pas d'erreur, l'efficacité peut primer sur la propreté. A notre niveau, il me parait surtout important de programmer proprement pour résoudre le problème principal : programmer sans bug.

En vous souhaitant de ne jamais tomber
du côté obscur de la programmation...

# Chapitre 2

# Généralités sur les objets

## 2.1 Définition formelle

Un objet est un ensemble cohérent de variables et de fonctions qui tournent autour d'un concept central. Formellement, un objet est défini par trois éléments :

- La *classe* est le nom de l'objet. C'est aussi son architecture, la liste de variables et de fonctions qui le compose.
- Les variables de l'objet sont appelées *attributs*.
- Les fonctions de l'objet sont appelées *méthodes*.

Dans l'introduction, nous avions défini un objet de classe `IMC`. Il avait pour attributs `poids` et `taille`, pour méthode `show`.

### 2.1.1 Les attributs

Les attributs sont simplement des variables typées. Une variable typée est une variable dont on a fixé la nature une bonne fois pour toutes. Dans R, on peut écrire `poids <- 62` (à ce stade, `poids` est un `numeric`) puis `poids <- "bonjour"` (`poids` est devenu un `character`). Le type de la variable `poids` peut changer.

En programmation objet, il ne sera pas possible de changer le type des attributs en cours de programmation [1]. Cela semble être une contrainte, c'est en fait une sécurité. En effet, si une variable a été créée pour contenir un poids, elle n'a aucune raison de recevoir une chaîne de caractères... sauf erreur de programmation.

### 2.1.2 Les méthodes

On distingue 4 types d'opérations à faire sur les objets :

- **Méthodes de création** : entrent dans cette catégorie toutes les méthodes permettant de créer un objet. La plus importante s'appelle le constructeur. Mais son utilisation est un peu rugueuse pour l'utilisateur. Le programmeur écrit donc des

1. Hélas, avec R et la S4, c'est possible. Mais comme c'est sans doute l'opération la plus impropre de toute l'histoire de l'informatique, nous préférons "oublier" et faire comme si ça ne l'était pas.

méthodes faciles d'accès pour créer un objet à partir de données, d'un fichier ou à partir d'un autre objet (par exemple `numeric` ou `read.csv2`).
- **Validation** : en programmation objet, il est possible de vérifier que les attributs respectent certaines contraintes. Il est également possible de créer des attributs à partir de calculs faits sur les autres attributs. Tout cela rentre dans la validation d'objet.
- **Manipulation des attributs** : modifier et lire les attributs n'est pas aussi anodin que dans la programmation classique. Aussi, dédie-t-on des méthodes à la manipulation des attributs (par exemple `names` et `names<-`).
- **Autres** : tout ce qui précède constitue une sorte de "minimum légal" à programmer pour chaque objet. Reste ensuite les méthodes spécifiques à l'objet, en particulier les méthodes d'affichage et les méthodes effectuant des calculs.

Les méthodes sont des fonctions typées. Le type d'une fonction est la juxtaposition de deux types : son type *entrée* et son type *sortie*.
- Le type-entrée est simplement l'ensemble des types des arguments de la fonction. Par exemple, la fonction `trace` prend pour argument une matrice, son type-entrée est donc `matrix`.
- Le type-sortie est le type de ce que la fonction retourne. Par exemple, la fonction `trace` retourne un nombre, son type-sortie est donc `numeric`.

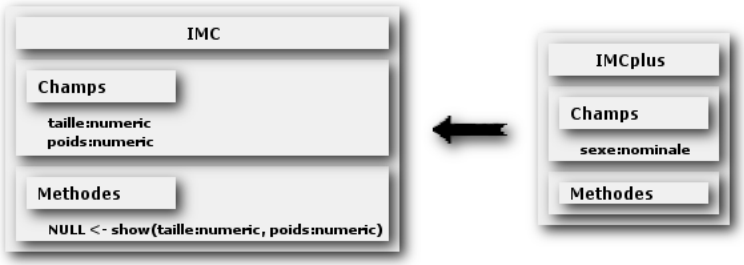
Au final, le type d'une fonction est `(type-entree,type-sortie)`. On le note `Type-Sortie <- fonction(TypeEntree)`.

Par exemple, le type de la fonction `trace` est `numeric <- trace(matrix)`.

## 2.2 Dessiner, c'est gagner !

*Il vaut mieux un bon dessin qu'un long discours.* La maxime s'applique particulièrement bien à la programmation objet. Chaque classe est représentée par un rectangle dans lequel sont notés les attributs et les méthodes avec leur type.

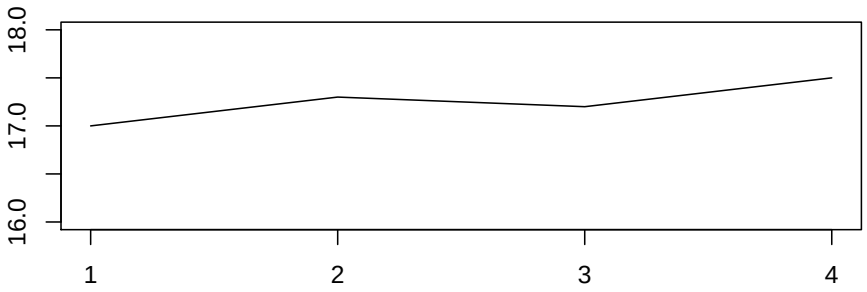
L'héritage (voir section 9 page 69) entre deux classes est noté par une flèche, du fils vers le père.



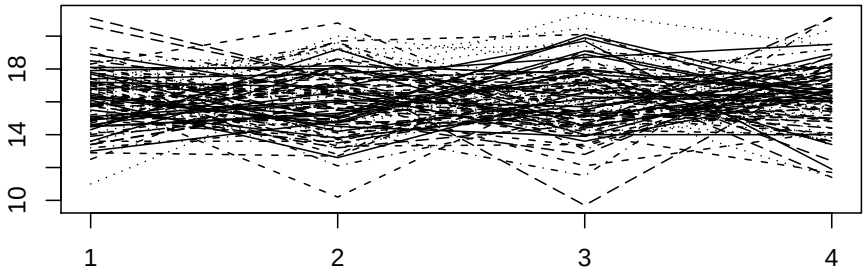
# Chapitre 3

# Exemple de travail

Plutôt que d'inventer un exemple avec des 'wiz', des 'spoun' ou des concepts mathématiques complexes truffés d'équations, voilà un cas réel (fortement simplifié, mais réel) : la doctoresse Tam travaille sur des patientes anorexiques. Semaine après semaine, elle mesure leur IMC (exemple `17`, puis `17.2`, puis `17.1`, puis `17.4`). Pour une patiente, la suite obtenue forme une `trajectoire` (exemple `(17, 17.3, 17.2, 17.4)`). Graphiquement, on obtient :




L'IMC de la demoiselle augmente progressivement. Lorsqu'on dispose de plusieurs trajectoires, les graphes deviennent illisibles :




D'où la nécessité, pour y voir plus clair, de regrouper les trajectoires.

Elle veut donc classer ses patientes en groupe selon des critères bien précis : `[AGrandi] (Oui)/(Non)`, `[DemandeAVoirSesParents] (Oui)/(Non)/(Refuse)`, `[RemangeRapidement] (Oui)/(Non)`. Au final, son but est de comparer différentes manières de regrouper les trajectoires.

## 3.1 Analyse du problème

Ce problème est découpable en trois objets.
- Le première sera celui qui contiendra les trajectoires des patientes.
- Le deuxième représentera un découpage en groupe (que nous appellerons une *partition*).
- Le troisième sera un mélange des deux : les trajectoires partitionnées en groupe.

## 3.2 L'objet `Trajectoires`

Tam nous prévient que pour un groupe donné, les mesures sont faites soit toutes les semaines, soit tous les quinze jours. L'objet `Trajectoires` doit en tenir compte. Il sera donc défini par deux attributs :
- `temps` : numéro de la semaine où les mesures sont effectuées. Pour simplifier, la semaine où commence le suivi du groupe aura le numéro 1.
- `traj` : les trajectoires de poids des patientes

Exemple :

$$\texttt{temps} = (1, 2, 4, 5)$$

$$\texttt{traj} = \begin{pmatrix} 15 & 15.1 & 15.2 & 15.2 \\ 16 & 15.9 & 16 & 16.4 \\ 15.2 & 15.2 & 15.3 & 15.3 \\ 15.7 & 15.6 & 15.8 & 16 \end{pmatrix}$$

Quelle(s) méthode(s) pour cet objet ? Tam nous dit que dans ce genre de recherche, il y a souvent des données manquantes. Il est important de savoir combien il y en a. Nous définirons donc une première méthode qui comptera le nombre de valeurs manquantes.

La deuxième méthode sera à cheval entre les objets `Trajectoires` et `Partition`[1] : une manière classique de construire une partition de groupes est de distinguer les patients selon leur IMC initial : dans notre exemple, on pourrait considérer deux groupes, les "IMC initial faible" et "IMC initial haut". Il faudra donc définir une méthode de découpage selon l'IMC initial et le nombre de groupes souhaité.

1. En R, ça n'a pas beaucoup d'importance mais dans un langage objet classique, chaque méthode doit obligatoirement appartenir à une classe et une seule. Pour nous conformer aux règles de l'objet, nous inclurons donc cette méthode dans `Trajectoires`.

Enfin, la présence de valeurs manquantes peut interdire l'utilisation d'outils statistiques. Il sera donc intéressant de pouvoir les imputer [2]. La troisième méthode imputera les manquantes.

## 3.3 L'objet `Partition`

Une partition est un ensemble de groupe. Par exemple les groupes pourraient être A ={T1,T3} et B={T2,T4}. Il sera sans doute plus simple de les considérer comme un vecteur ayant pour longueur le nombre de patientes : {A,B,A,B}. Dans un certain nombre de cas, il faudra également connaître le nombre de groupe, en particulier quand un groupe est manquant : si Tam classe ses ados en trois groupes et qu'elle obtient la partition {A,C,C,A}, il est important de garder en mémoire que le groupe B existe, même s'il ne contient personne. D'où deux attributs pour l'objet `Partition` :

- `nbGroupes` : donne le nombre de groupes.
- `part` : la suite des groupes auxquels les trajectoires appartiennent.

Exemple :

`nbGroupe = 3`

$$\texttt{part} = \begin{pmatrix} A \\ B \\ A \\ C \end{pmatrix}$$

Certains statisticiens (et même certains logiciels) transforment les variables nominales en chiffres. Nous pourrions définir `part` comme un vecteur d'entier. "C'est beaucoup plus pratique", s'entend-on dire régulièrement. Sauf que ce faisant, on trompe R sur la nature de nos variables : R est conçu pour savoir calculer la moyenne d'une variable numérique, mais refuser de calculer la moyenne d'une variable nominale (un `factor`). Si nous codons `part` avec des chiffres, R acceptera de calculer la moyenne d'une partition, ce qui n'a aucun sens. Ça serait un peu comme faire la moyenne de Tortue, Girafe et Tigre... Une variable nominale doit donc être codée par un `factor`, une numérique par un `numeric`. Oui, nous savons, dans certains cas, il serait un peu plus *pratique* de coder `part` par des nombres, mais c'est beaucoup plus dangereux. Donc *à éviter d'urgence* !

## 3.4 L'objet `TrajDecoupees`

`TrajDecoupees` sera l'objet regroupant un objet `Trajectoires` et plusieurs `Partition`. En effet, il est possible que pour un ensemble de trajectoires, plusieurs partitionnements soient intéressants. Il sera donc héritier de `Trajectoires`. Par contre, il ne

2. L'imputation d'une valeur manquante est son remplacement par une valeur que l'on "devine" grâce aux autres valeurs... avec plus ou moins de bonheur !

sera pas héritier de `Partition` parce que les deux classes ne partagent pas vraiment de propriétés. Pour plus de détail, voir section 9 page 69.

- `temps` : le temps auquel les mesures sont effectuées (comme dans 'trajectoires')
- `traj` : les trajectoires de poids des patientes (comme dans 'trajectoires')
- `listePartitions` : un ensemble de partitions.

Exemple :

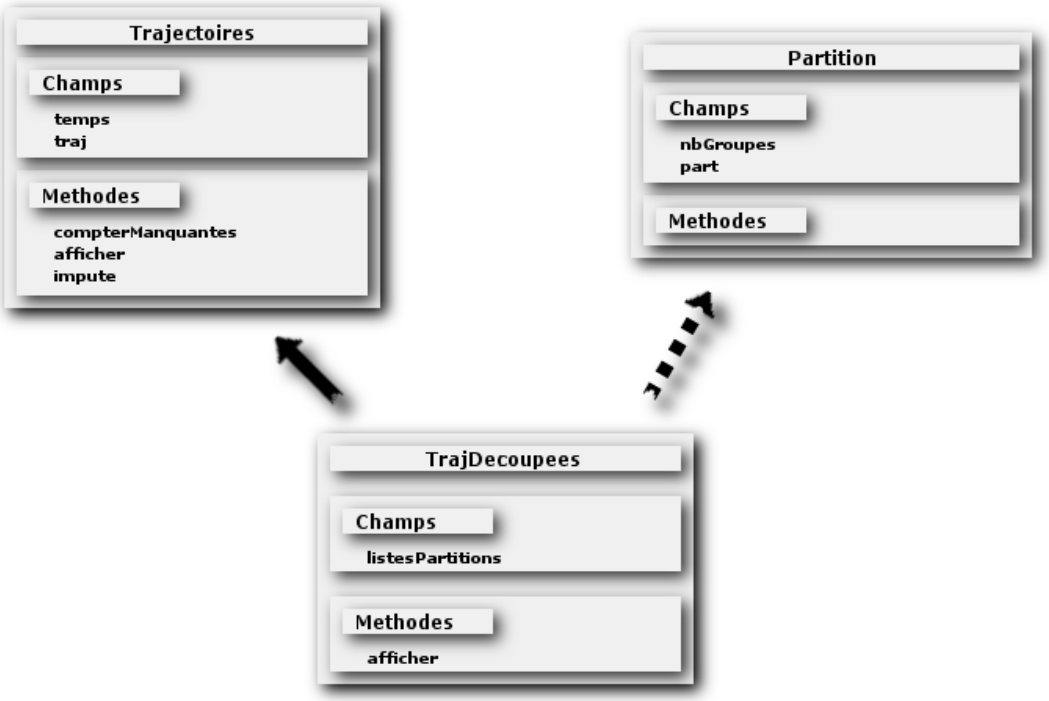
$$\texttt{temps} = (1, 2, 4, 5)$$

$$\texttt{traj} = \begin{pmatrix} 15 & 15.1 & 15.2 & 15.2 \\ 16 & 15.9 & 16 & 16.4 \\ 15.2 & 15.2 & 15.3 & 15.3 \\ 15.5 & 15.6 & 15.8 & 16 \end{pmatrix}$$

$$\texttt{listePartitions} = \left[\texttt{nbGroupe} = 3,\ \texttt{part} = \begin{pmatrix} A \\ B \\ A \\ C \end{pmatrix}\right] \left[\texttt{nbGroupe} = 2,\ \texttt{part} = \begin{pmatrix} A \\ B \\ A \\ A \end{pmatrix}\right]$$

## 3.5 Plan d'analyse

Pour résumer, nous avons donc trois classes :



La flèche pleine (du fils vers le père) indique l'héritage, la flèche en pointillé désigne l'inclusion d'un objet dans l'autre sans héritage.

## 3.6 Application à R

Et R dans tout ça, seriez-vous en droit de réclamer ? C'est également une des caractéristiques des langages objets, nous avons fait une analyse relativement poussée (poussée pour un problème aussi simple que le nôtre) et il n'y a toujours pas la moindre ligne de code R... Théoriquement, on pourrait même choisir de coder dans un autre langage. Mais bon, là n'est pas le sujet du jour.

Appliquées à R, les méthodes définies au 2.1.2 page 19 deviennent :

- **Méthodes de création** : la méthode de création principale porte le nom de la classe.
- **Validation** : Cela dépend de l'objet.
- **Manipulation des attributs** : pour chaque attribut, il faudra une méthode permettant d'accéder à sa valeur et une méthode permettant de la modifier.
- **Autres** : les méthodes "autres" dépendent des particularités de l'objet, à l'exception toutefois des méthodes d'affichage. Pour l'affichage, `show` permet un affichage sommaire de l'objet quand on tape son nom dans la console. `print` donne un affichage plus complet. `plot` est l'affichage graphique.

Pour finir avec les particularités de R, la majorité des langages objets forcent le programmeur à grouper tout ce qui concerne un objet dans un même endroit. Cela s'appelle l'encapsulation. R n'a pas cette propriété : vous pouvez très bien déclarer un objet, puis un autre, puis une méthode pour le premier objet, puis déclarer un troisième objet, et ainsi de suite. C'est fortement déconseillé, et de surcroit facilement évitable en suivant une règle toute simple :

**À chaque objet son fichier.**

Tout ce qui concerne un objet doit être dans un unique ficher, un fichier ne doit contenir que des informations relatives à un unique objet. Cela sera repris plus en détail lors de la création de package section **??** page **??**.

# Deuxième partie

# Les bases de l'objet

# Chapitre 4

# Déclaration des classes

Nous l'avons évoqué dans les chapitres précédents, une classe est un ensemble d'attributs et de méthodes. Nous allons maintenant définir les uns et les autres.

Dans la majorité des langages objets, la définition de l'objet contient les attributs et les méthodes. En R, la définition ne contient que les attributs. Les méthodes sont précisées ensuite. C'est dommage, ça atténue la puissance de l'encapsulation, mais c'est comme ça. L'utilisateur averti (c'est à dire vous) peut compenser "manuellement", par exemple en se forçant à définir attributs et méthodes en bloc dans un même fichier, et en utilisant un fichier par objet.

Plus de conseils sur l'art de bien encapsuler section **??** page **??**.

## 4.1 Définition des attributs

La première étape est de définir les attributs de l'objet proprement dit. Cela se fait à l'aide de l'instruction `setClass`. `setClass` est une fonction qui prend deux arguments (et quelques autres ingrédients que nous verrons plus tard).

- `Class` (avec une majuscule) est le nom de la classe que nous sommes en train de définir.
- `representation` est la liste des attributs de la classe.

Comme nous l'avons vu dans l'exemple introductif, la programmation objet fait du contrôle de type, c'est à dire qu'elle ne permettra pas à une chaîne de caractères d'être rangée là où devrait se trouver en entier. Chaque attribut doit être déclaré avec son type.

```
> setClass(
+     Class="Trajectoires",
+     representation=representation(
+         temps = "numeric",
+         traj = "matrix"
+     )
+ )
```

```
[1] "Trajectoires"
```

Malheureusement, il est tout de même possible de ne pas typer (ou de mal typer) en utilisant les listes. Le non typage est fortement déconseillé, mais lors de l'utilisation de listes, il n'y a pas d'autre option.

## 4.2 Constructeur par défaut

On peut ensuite créer un objet `trajectoire` grâce au constructeur `new` :

```
> new(Class="Trajectoires")
```

```
An object of class "Trajectoires"
Slot "temps":
numeric(0)

Slot "traj":
<0 x 0 matrix>
```

Comme vous pouvez le constater, l'affichage n'est pas extraordinaire. Il sera important de définir une méthode pour l'améliorer.

En général, on définit un objet en spécifiant les valeurs de ses attributs. On doit à chaque fois préciser le nom de l'attribut en question :

```
> new(Class="Trajectoires",temps=c(1,3,4))
```

```
An object of class "Trajectoires"
Slot "temps":
[1] 1 3 4

Slot "traj":
<0 x 0 matrix>
```

```
> new(Class="Trajectoires",temps=c(1,3),traj=matrix(1:4,ncol=2))
```

```
An object of class "Trajectoires"
Slot "temps":
[1] 1 3

Slot "traj":
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Naturellement, un objet se stocke dans une variable comme n'importe quelle autre valeur de R. Pour illustrer nos dires, nous allons constituer une petite base de travail. Trois hôpitaux participent à l'étude. La Pitié Salpêtrière (qui n'a pas encore rendu son fichier de données, honte à eux), Cochin et Sainte-Anne :

```
> trajPitie <- new(Class="Trajectoires")
> trajCochin <- new(
+     Class="Trajectoires",
```

```
+     temps=c(1,3,4,5),
+     traj=rbind(
+         c(15,15.1,15.2,15.2),
+         c(16,15.9,16,16.4),
+         c(15.2,NA,15.3,15.3),
+         c(15.7,15.6,15.8,16)
+     )
+ )
> trajStAnne <- new(
+     Class="Trajectoires",
+     temps=c(1:10,(6:12)*2),
+     traj=rbind(
+       matrix(seq(16,19,length=17),ncol=17,nrow=50,byrow=TRUE),
+       matrix(seq(15.8,18,length=17),ncol=17,nrow=30,byrow=TRUE)
+     )+rnorm(17*80,0,0.2)
+ )
```

## 4.3 Accéder aux attributs

Toute cette section est placée sous le double signe
du danger...
et du danger...

L'accès aux attributs se fait grâce à l'opérateur `@` :

```
> trajCochin@temps
```

```
[1] 1 3 4 5
```

```
> trajCochin@temps <- c(1,2,4,5)
> trajCochin
```

```
An object of class "Trajectoires"
Slot "temps":
[1] 1 2 4 5

Slot "traj":
     [,1] [,2] [,3] [,4]
[1,] 15.0 15.1 15.2 15.2
[2,] 16.0 15.9 16.0 16.4
[3,] 15.2   NA 15.3 15.3
[4,] 15.7 15.6 15.8 16.0
```

Comme nous le verrons par la suite, l'utilisation de `@` est à éviter. En effet, il ne fait pas appel aux méthodes de vérification. L'utilisation que nous présentons ici (affichage d'un attribut, et pire encore affectation d'une valeur à un attribut) est donc à proscrire dans la plupart des cas. En tout état de cause, l'utilisateur final ne devrait jamais avoir à l'utiliser.

Il est également possible d'utiliser les fonctions `attr` ou `attributes`, mais c'est largement pire : en effet, si on fait une simple erreur de typographie, on modifie la structure de l'objet. Et ça, c'est *très très très mal* !

## 4.4 Valeurs par défaut

On peut déclarer un objet en lui donnant des valeurs par défaut. A chaque création, si l'utilisateur ne spécifie pas les valeurs des attributs, ceux-ci en auront quand même une. Pour cela, on doit ajouter l'argument `prototype` à la définition de l'objet :

```
> setClass(
+     Class="TrajectoiresBis",
+     representation=representation(
+         temps = "numeric",
+         traj = "matrix"
+     ),
+     prototype=prototype(
+         temps = 1,
+         traj = matrix(0)
+     )
+ )
```

```
[1] "TrajectoiresBis"
```

L'initialisation par défaut était quelque chose de nécessaire à l'époque lointaine où, si on n'initialisait pas une variable, on risquait d'écrire dans la mémoire système (et donc de provoquer un blocage de l'ordinateur, la perte de notre programme et d'autres trucs encore plus terribles). Aujourd'hui, une telle chose n'est plus possible. Si on n'initialise pas un attribut, R lui donne pour valeur un objet vide du type adéquat.

Du point de vue philosophique, lorsqu'on crée un objet, soit on connaît sa valeur auquel cas on la lui affecte, soit on ne la connaît pas auquel cas il n'y a aucune raison de lui donner une valeur plutôt qu'une autre. L'utilisation d'une valeur par défaut semble donc être plus une réminiscence du passé qu'une réelle nécessité. Elle n'a plus lieu d'être.

De plus, dans le feu de la programmation, il peut arriver qu'on "oublie" de donner à un objet sa vraie valeur. S'il existe une valeur par défaut, elle sera utilisée. S'il n'y a pas de valeur par défaut, cela provoquera une erreur... ce qui, dans ce cas précis, est préférable, cela attire notre attention et nous permet de corriger. Donc, les valeurs par défaut autres que les valeurs vides sont à éviter.

## 4.5 Supprimer un objet

Dans le cas précis de l'objet `trajectoires`, il n'y a pas vraiment de valeur par défaut qui s'impose. Il est donc préférable de conserver la classe comme elle a été initialement définie. La classe `TrajectoiresBis` n'a plus raison d'être. On peut la supprimer grâce à `removeClass` :

```
> removeClass("TrajectoiresBis")
```

```
[1] TRUE
```

```
> new(Class="TrajectoiresBis")
```

```
Error in getClass(Class, where = topenv(parent.frame())) :

"TrajectoiresBis" is not a defined class
```

Supprimer la définition d'une classe ne supprime pas les méthodes qui lui sont associées. Pour supprimer définitivement une classe dans le sens classique du terme, il faut supprimer la classe *puis* supprimer toutes ses méthodes...

En particulier, vous créez une classe et ses méthodes. Ça ne marche pas comme prévu. Vous décidez de tout reprendre à zéro. Vous effacez donc la classe. Si vous la recréez, toutes les anciennes méthodes seront à nouveau actives.

## 4.6 L'objet vide

Certaines fonctionnalités objets appellent la fonction `new` sans lui transmettre d'argument. Par exemple, nous utiliserons dans la section sur l'héritage l'instruction `as(tdCochin,"Trajectoires")` (voir section 9.8 page 77). Cette instruction fait appel à `new("Trajectoires")`. Il est donc indispensable, lors de la construction d'un objet, de toujours garder à l'esprit que `new` doit être utilisable sans argument. Comme les valeurs par défaut sont déconseillées, il faut prévoir la construction de l'objet vide. Cela sera important en particulier lors de la construction de la méthode `show`.

Un objet vide est un objet possédant tous les attributs normaux d'un objet, mais ceux-ci sont vides, c'est à dire de longueur zéro. Par contre, un objet vide a une classe. Ainsi, un `numeric` vide est différent d'un `integer` vide.

```
> identical(numeric(),integer())
```

```
[1] FALSE
```

Quelques règles à connaître concernant les objets vides :

- `numeric()` et `numeric(0)` désignent un `numeric` vide.
- Même chose pour `character()` et `character(0)`.
- Même chose pour `integer()` et `integer(0)`.
- Par contre, `factor()` désigne un `factor` vide, `factor(0)` désigne un `factor` de longueur 1 et contenant l'élement zéro.
- Plus problématique, `matrix()` désigne une matrice à une ligne et une colonne contenant NA. En tout état de cause, ça n'est pas la matrice vide (son attribut `length` vaut 1). Pour définir une matrice vide, il faut utiliser `matrix(nrow=0,ncol=0)`.
- Même chose pour les `array()`.
- `NULL` représente l'objet nul. Ainsi, `NULL` est de classe `NULL` alors que `numeric()` est de classe `numeric`.

Pour tester qu'un objet est l'objet vide, il faut tester son attribut `length`. De même, si nous décidons de définir la méthode `length` pour nos objets, il faudra prendre garde

à ce que `length(monObjet)=0` soit vrai si et seulement si l'objet est vide (pour assurer une certaine cohérence à R).

## 4.7 Voir l'objet

Vous venez de créer votre première classe. Félicitations ! Pour pouvoir vérifier ce que vous venez de faire, plusieurs instructions permettent de "voir" la classe et sa structure. Le terme savant désignant ce qui permet au programme de voir le contenu ou la structure des objets s'appelle l'*introspection*.

`slotNames` donne le nom des attributs. `getSlots` donne le nom des attributs et leur type. `getClass` donne les noms des attributs et leur type, mais aussi les héritiers et les ancêtres. Comme l'héritage est encore "terra incognita", ça ne fait pour l'instant aucune différence :

```
> slotNames("Trajectoires")
```

```
[1] "temps" "traj"
```

```
> getSlots("Trajectoires")
```

```
    temps      traj
"numeric"  "matrix"
```

```
> getClass("Trajectoires")
```

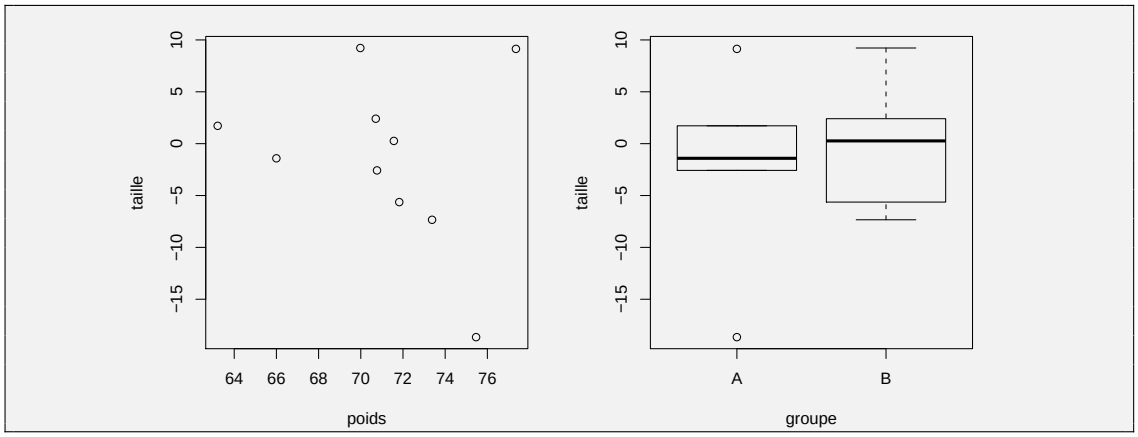
```
Class "Trajectoires" [in ".GlobalEnv"]

Slots:

Name:    temps    traj
Class: numeric  matrix
```

# Chapitre 5

# Les méthodes

Un des intérêts de l'objet est de pouvoir définir des fonctions qui vont adapter leur comportement à l'objet. Exemple que vous connaissez déjà, la fonction `plot` réagit différemment selon la classe de ses arguments :

```
> taille <- rnorm(10,1.70,10)
> poids <- rnorm(10,70,5)
> groupe <- as.factor(rep(c("A","B"),5))
> plot(taille~poids)
> plot(taille~groupe)
```




Le premier `plot` trace un nuage de points, le deuxième dessine des boites à moustaches. De même, il va être possible de définir un comportement spécifique pour nos trajectoires.

## 5.1 `setMethod`

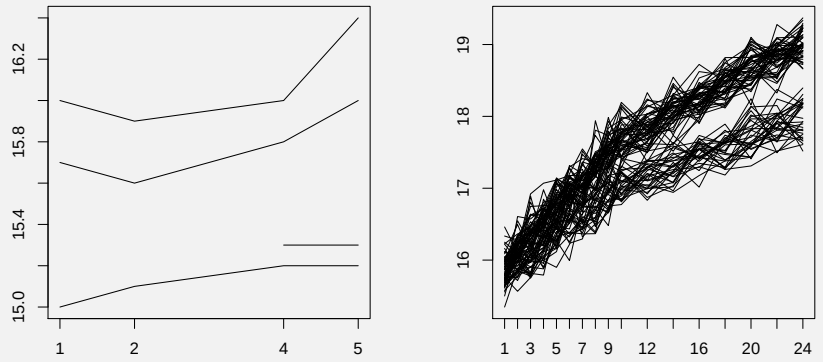
Pour cela, on utilise la fonction `setMethod`. Elle prend trois arguments :

1. `f` est le nom de la fonction que nous sommes en train de redéfinir. Dans notre cas, `plot`
2. `signature` est le nom de la classe à laquelle elle s'applique. Nous aurons l'occasion d'y revenir section 8.2 page 64
3. `definition` est la fonction à utiliser. Dans notre cas, nous allons simplement utiliser un `matplot` en prenant en compte les temps des mesures et en les affichant sur l'axe des abscisses

```
> setMethod(
+     f="plot",
+     signature="Trajectoires",
+     definition=function(x,y,...){
+         matplot(x@temps,t(x@traj),xaxt="n",type="l",
+                 ylab="",xlab="",pch=1,col=1,lty=1)
+         axis(1,at=x@temps)
+     }
+ )
```

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

```
> plot(trajCochin)
> plot(trajStAnne)
```




Petite remarque : R nous impose, lors de la redéfinition d'une fonction, d'utiliser les mêmes arguments que la fonction en question. Pour connaître les arguments de `plot`, on peut utiliser `args`

```
> args(plot)
```

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

Nous sommes donc obligés d'utiliser `function(x,y,...)` même si nous savons d'ors et déjà que l'argument `y` n'a pas de sens. De plus, les noms par défaut ne sont pas vraiment uniformisés, certaines fonctions appellent `object`, d'autres `.Object`, d'autres encore `x`. R-la-goélette vogue au gré de ses programmeurs...

## 5.2 show et print

`show` et `print` servent pour l'affichage. Nous allons les définir pour les trajectoires. `args(print)` nous indique que `print` prend pour argument `(x,...)`. Donc :

```
> setMethod("print","Trajectoires",
+     function(x,...){
+         cat("*** Class Trajectoires, method Print ***\n")
+         cat("* Temps = ");print(x@temps)
+         cat("* Traj  = \n");print(x@traj)
+         cat("******* Fin  Print(trajectoires) *******\n")
+     }
+ )
```

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

```
> print(trajCochin)
```

```
*** Class Trajectoires, method Print ***
* Temps = [1] 1 2 4 5
* Traj  =
     [,1] [,2] [,3] [,4]
[1,] 15.0 15.1 15.2 15.2
[2,] 16.0 15.9 16.0 16.4
[3,] 15.2   NA 15.3 15.3
[4,] 15.7 15.6 15.8 16.0
******* Fin  Print(trajectoires) *******
```

Pour Cochin, le résultat est satisfaisant. Pour Sainte-Anne (qui compte 80 lignes), on ne verrait pas grand-chose, d'où le besoin d'une deuxième méthode d'affichage.

`show` est la méthode utilisée quand on tape le nom d'un objet. Nous allons donc la redéfinir en prenant en compte la taille de l'objet : s'il y a trop de trajectoires, `show` n'en affichera qu'une partie.

```
> setMethod("show","Trajectoires",
+     function(object){
+         cat("*** Class Trajectoires, method Show ***\n")
+         cat("* Temps = ");print(object@temps)
+         nrowShow <- min(10,nrow(object@traj))
+         ncolShow <- min(10,ncol(object@traj))
+         cat("* Traj  (limité à une matrice 10x10) = \n")
+         print(formatC(object@traj[1:nrowShow,1:ncolShow]),
+               quote=FALSE)
+         cat("* ... ...\n")
+         cat("******* Fin  Show(trajectoires) *******\n")
+     }
+ )
```

```
[1] "show"
```

```
> trajStAnne
```

```
*** Class Trajectoires, method Show ***
* Temps =  [1]  1  2  3  4  5  6  7  8  9 10 12 14 16 18 20 22 24
* Traj  (limité à une matrice 10x10) =
      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9]  [,10]
 [1,] 16.25 16.1  16.16 16.53 16.67 17.2  17.55 17.3  17.47 17.55
 [2,] 16.23 16.36 16.45 16.4  17.14 17    16.84 17.25 17.98 17.43
 [3,] 15.89 16.02 16.15 16.69 16.77 16.62 17.52 17.26 17.46 17.6
 [4,] 15.63 16.25 16.4  16.6  16.65 16.96 17.02 17.39 17.5  17.67
 [5,] 16.16 15.85 15.97 16.32 16.73 16.94 16.73 16.98 17.64 17.72
 [6,] 15.96 16.2  16.53 16.4  16.47 16.95 16.73 17.36 17.33 17.55
 [7,] 16.03 16.33 16.23 16.67 16.79 17.14 17    17.35 17.58 17.99
 [8,] 15.69 16.06 16.63 16.72 16.81 17.16 16.98 17.41 17.51 17.43
 [9,] 15.82 16.17 16.75 16.76 16.78 16.51 17.19 17.21 17.84 17.95
[10,] 15.98 15.76 16.1  16.54 16.78 16.89 17.22 17.18 16.94 17.36
* ... ...
******* Fin  Show(trajectoires) *******
```

Reste un problème à régler. Nous avons vu section 4.6 page 33 que `new` devait être utilisable sans argument. Or, il ne l'est plus :

```
> new("Trajectoires")
```

```
*** Class Trajectoires, method Show ***
* Temps = numeric(0)
* Traj  (limité à une matrice 10x10) =
Error in print(formatC(object@traj[1:nrowShow, 1:ncolShow]), quote
= FALSE) :
  erreur lors de l'évaluation de l'argument 'x' lors de
la sélection d'une méthode pour la fonction 'print'
```

En effet, `new` crée un objet, puis l'affiche en utilisant `show`. Dans le cas de `new` sans argument, l'objet vide est passé à `show`. Or, `show` tel que nous l'avons conçu ne peut pas traiter l'objet vide.

D'une manière plus générale, toutes nos méthodes doivent prendre en compte le fait qu'elles auront peut-être à traiter l'objet vide :

```
> setMethod("show","Trajectoires",
+     function(object){
+         cat("*** Class Trajectoires, method Show ***\n")
+         cat("* Temps = ");print(object@temps)
+         nrowShow <- min(10,nrow(object@traj))
+         ncolShow <- min(10,ncol(object@traj))
+         cat("* Traj  (limité à une matrice 10x10) = \n")
+         if(length(object@traj)!=0){
+             print(formatC(object@traj[1:nrowShow,1:ncolShow]),
+             quote=FALSE)
+         }else{}
+         cat("* ... ...\n")
```

```
+          cat("******* Fin  Show(trajectoires) *******\n")
+      }
+ )
```

```
[1] "show"
```

```
> new("Trajectoires")
```

```
*** Class Trajectoires, method Show ***
* Temps = numeric(0)
* Traj  (limité à une matrice 10x10) =
* ... ...
******* Fin  Show(trajectoires) *******
```

Ça marche !

Nous disposons donc de deux méthodes d'affichage. Par défaut, `show` montre l'objet ou une partie de l'objet si celui-ci est trop grand ; `print` permet d'afficher l'intégralité de l'objet.

## 5.3 setGeneric

Jusqu'à présent, nous n'avons fait que définir pour l'objet `Trajectoires` des méthodes qui existaient déjà par ailleurs (`print` existait pour les `numeric`, pour les `character`...). Nous allons maintenant définir une méthode nouvelle. Pour cela, il nous faut la déclarer. Cela se fait à l'aide de la fonction `setGeneric`. Á ce stade, une petite digression sur le concept de générique s'impose.

### 5.3.1 Générique versus Spécifique

En S4, les fonctions doivent être définies de deux manières : générique et spécifique. La définition générique d'une fonction est la donnée d'un comportement global (ou conceptuel). La définition spécifique d'une fonction est l'application de ce concept à un cas particulier. Un exemple va éclaircir les choses.

`plot` est une fonction qui représente graphiquement des données. Telle est sa définition générique. Le type précis de représentation n'entre pas en ligne de compte dans la définition générique. Celle-ci ne s'occupe pas de détail ou des cas particuliers, elle reste floue, générale...

`plot` *appliqué à* une variable `numeric` trace un histogramme. C'est une définition spécifique. Nous ne sommes plus au niveau du concept mais dans la discussion pratique : il faut définir le type de graphe qui sera précisément utilisé. Dans notre cas, c'est un histogramme.

Une fonction n'a qu'une définition générique, mais peut avoir plusieurs définitions spécifiques. Par exemple, `plot` appliqué à un `(numeric,factor)` trace des boites à moustaches. C'est une deuxième définition spécifique.

Dans R, chaque fonction spécifique doit nécessairement être connue du programme comme fonction générique : avant d'entrer dans les détails, il faut définir le concept global.

Une méthode nouvelle devra donc en premier lieu être déclarée comme fonction générique, puis comme fonction spécifique. Une fonction dont la générique existe déjà (comme `print`) n'a pas besoin d'être déclarée en générique et peut directement être déclarée en spécifique, comme nous l'avons fait au paragraphe précédent. Mais toute fonction nouvelle doit être déclarée en générique.

### 5.3.2 Définition formelle

Définir une fonction générique se fait grâce à `setGeneric`. `setGeneric` prend deux arguments :

- `name` est le nom de la méthode que nous allons définir.
- `def` est un exemple de fonction qui sera utilisé pour la définir.

A ce stade, il n'est pas possible de la typer puisqu'elle est générique et doit donc être utilisable pour plusieurs types différents.

```
> setGeneric(
+     name="compterManquantes",
+     def=function(object){standardGeneric("compterManquantes")}
+ )
```

```
[1] "compterManquantes"
```

`compterManquantes` a donc été ajouté à la liste des méthodes que R connait. Nous pouvons maintenant la définir comme fonction spécifique pour l'objet `Trajectoires` :

```
> setMethod(
+     f="compterManquantes",
+     signature="Trajectoires",
+     definition=function(object){
+         return(sum(is.na(object@traj)))
+     }
+ )
```

```
[1] "compterManquantes"
```

```
> compterManquantes(trajCochin)
```

```
[1] 1
```

### 5.3.3 `lockBinding`

Il n'y a pas de contrôle sur l'utilisation d'un `setGeneric` : si une fonction générique existe déjà, la nouvelle définition détruit l'ancienne - un peu de la même manière qu'affecter une valeur à une variable détruit la précédente valeur -. Sauf que dans ce cas précis, une redéfinition est plus probablement liée au fait que le programmeur ignore que la fonction existe déjà... Pour se protéger de ce problème, il est possible de "verrouiller" la définition d'une méthode grâce à `lockBinding` :

```
> lockBinding("compterManquantes",.GlobalEnv)
```

```
> setGeneric(
+     name="compterManquantes",
+     def=function(object,value){
+         standardGeneric("compterManquantes")
+     }
+ )
```

```
Error in assign(name, fdef, where) :
  impossible de changer la
valeur d'un lien verrouillé pour 'compterManquantes'
```

Il n'est plus possible d'effacer "par erreur" le `setGeneric`.

Cette méthode présente toutefois un inconvénient majeur, celui de la non ré-exécutabilité du code. Pendant la phase de développement, on a tendance à exécuter notre code, le modifier et le ré-exécuter. `lockBinding` empêche une telle ré exécution puisqu'une fonction générique ne peut-être définie qu'une fois (et que la ré-exécution est une seconde définition).

### 5.3.4 Déclaration des génériques

Une autre manière de se protéger contre l'écrasement des génériques est de regrouper la déclaration des génériques dans un fichier unique. En tout état de cause, une fonction générique ne concerne pas un objet particulier puisque, par définition, elle doit s'adapter à tous les objets. Il est donc préférable de déclarer toutes les fonctions génériques ensemble en début de programme, éventuellement classées par ordre alphabétique. Si par erreur nous devions décider de déclarer une générique deux fois, il serait alors facile de s'en rendre compte. Plus de détail sur l'art et la manière de placer son code dans le fichier adéquat section **??** page **??**.

## 5.4 Voir les méthodes

Notre classe commence à s'étoffer. Il est temps de faire une petite pause et d'admirer notre travail. `showMethods` est la méthode de la situation. Il existe plusieurs manières de l'utiliser. L'une d'entre elles permet de voir les noms des méthodes que nous avons définies pour une classe donnée :

```
> showMethods(class="Trajectoires")
```

```
Function: initialize (package methods)
.Object="Trajectoires"
    (inherited from: .Object="ANY")

Function: plot (package graphics)
x="Trajectoires"

Function: print (package base)
```

```
x="Trajectoires"

Function: show (package methods)
object="Trajectoires"
```

Maintenant que nous avons listé ce qui existe, nous pouvons nous intéresser d'un peu plus près à une méthode particulière : `getMethod` permet d'afficher la définition (le contenu du corps de la fonction) d'une méthode pour un objet donné. Si la méthode en question n'existe pas, `getMethod` renvoie une erreur :

```
> getMethod(f="plot",signature="Trajectoires")
```

```
Method Definition:

function (x, y, ...)
{
    matplot(x@temps, t(x@traj), xaxt = "n", type = "l", ylab = "",
        xlab = "", pch = 1, col = 1, lty = 1)
    axis(1, at = x@temps)
}

Signatures:
        x
target  "Trajectoires"
defined "Trajectoires"
```

```
> getMethod(f="plot",signature="Partition")
```

```
Error in getMethod(f = "plot", signature = "Trjectoires") :
  No
method found for function "plot" and signature Trjectoires
```

Plus simplement, `existsMethod` indique si une méthode est ou n'est pas définie pour une classe :

```
> existsMethod(f="plot",signature="Trajectoires")
```

```
[1] TRUE
```

```
> existsMethod(f="plot",signature="Partition")
```

```
[1] FALSE
```

Ça n'est pas vraiment utile pour l'utilisateur, ça l'est plus pour le programmeur qui peut écrire des choses du genre : "Si telle méthode existe pour tel objet, adopte le comportement 1, sinon le comportement 2".

# Chapitre 6

# Construction

La construction regroupe tous les outils permettant de fabriquer une instance correcte d'un objet. Entrent dans cette catégorie les méthodes de création proprement dites (méthodes qui stockent les valeurs dans les attributs) et les méthodes de validation (méthodes qui vérifient que les valeurs des attribuées sont conformes à ce que le programmeur souhaite).

## 6.1 Vérificateur

Le vérificateur est là pour contrôler qu'il n'y a pas d'incohérence interne dans l'objet. Par exemple, une taille doit être positive. On lui donne des règles et à chaque création d'objet, il vérifie que l'objet suit les règles.

Pour cela, on inclut les paramètres de vérification dans la définition de l'objet lui-même via l'argument `validity`. Pour l'objet `Trajectoires`, on peut vouloir vérifier que le nombre de groupes effectivement présents dans `cluster` est inférieur ou égal au nombre de groupes déclaré dans `nbCluster`.

```
> setClass(
+     Class="Trajectoires",
+     representation(temps = "numeric",traj = "matrix"),
+     validity=function(object){
+         cat("  ~~~ Trajectoires : vérificateur ~~~\n")
+         if(length(object@temps)!=ncol(object@traj)){
+             stop("[Trajectoire:validation] Le nombre de mesures
+               temporelles ne correspond pas au nombre de
+               colonnes de la matrice")
+         }else{}
+         return(TRUE)
+     }
+ )
```

```
[1] "Trajectoires"
```

```
> new(Class="Trajectoires",temps=1:2,traj=matrix(1:2,ncol=2))
```

```
  ~~~ Trajectoires : vérificateur ~~~
*** Class Trajectoires, method Show ***
* Temps = [1] 1 2
* Traj  (limité à une matrice 10x10) =
[1] 1 2
* ... ...
******* Fin  Show(trajectoires) *******
```

```
> new(Class="Trajectoires",temps=1:3,traj=matrix(1:2,ncol=2))
```

```
  ~~~ Trajectoires : vérificateur ~~~
Error in validityMethod(object) :
  [Trajectoire:validation] Le
nombre de mesures
              temporelles ne correspond pas au
nombre de
              colonnes de la matrice
```

Comme vous pouvez le constater, la fonction `validity` telle que nous venons de la définir ne prend aucune précaution concernant l'objet vide. Mais cela n'a pas d'importance. En effet, `new` n'appelle pas le vérificateur quand on ne lui donne pas d'argument.

Il est également possible de définir une classe, puis plus tard de définir sa validité grâce à une fonction appelée `setValidity`. De la même manière, il est possible de définir la `representation` et le `prototype` en externe. Mais cette manière de faire est conceptuellement moins propre. En effet, la conception d'un objet doit être réfléchie, et non pas faite d'ajouts à droite et à gauche...

Le vérificateur n'est appelé QUE lors de la création initiale de l'objet. Si ensuite il est modifié, rien ne va plus, il n'y a plus de contrôle. Nous pourrons bientôt corriger cela grâce aux "setteurs". Pour l'instant, on note juste l'intérêt de proscrire l'utilisation de `@` : la modification directe d'un attribut n'est pas soumise à vérification...

```
> trajStLouis <- new(
+     Class="Trajectoires",temps=c(1),traj=matrix(1)
+ )
```

```
  ~~~ Trajectoires : vérificateur ~~~
```

```
> ### Pas de vérification, le nombre de mesures temporelles ne
> ### correspond plus aux trajectoires
> (trajStLouis@temps <- c(1,2,3))
```

```
[1] 1 2 3
```

## 6.2 L'initiateur

Le vérificateur est une version simplifiée d'un outil plus général appelé l'*initiateur*. L'initiateur est une méthode permettant de fabriquer un objet. Il est appelé à chaque construction d'un objet, c'est à dire à chaque utilisation de la fonction `new`.

Reprenons nos trajectoires. Il serait assez plaisant que les colonnes de la matrice des trajectoires aient des noms, les noms des temps où les mesures ont été prises. De même, les lignes pourraient être indicées par un numéro d'individu :

```
     T0    T1    T4    T5
I1   15  15.1  15.2  15.2
I2   16  15.9    16  16.4
I3 15.2    NA  15.3  15.3
I4 15.5  15.6  15.8    16
```

L'initiateur va nous permettre de faire tout ça. L'initiateur est une méthode qui, lors de l'appel de `new`, fabrique l'objet tel que nous le voulons. Le nom de la méthode est `initialize`. `initialize` utilise une fonction (définie par l'utilisateur) qui prend pour argument l'objet en train d'être construit et les différentes valeurs à affecter aux attributs de l'objet. Cette fonction travaille sur une version locale de l'objet. Elle doit se terminer par l'affectation des valeurs aux attributs de l'objet puis par un `return(object)`.

```
> setMethod(
+     f="initialize",
+     signature="Trajectoires",
+     definition=function(.Object,temps,traj){
+         cat(" ~~~ Trajectoires : initiateur ~~~\n")
+         rownames(traj) <- paste("I",1:nrow(traj),sep="")
+ # Affectation des attributs
```

```
+         .Object@traj <- traj
+
+         .Object@temps <- temps
+ # return de l'objet
+         return(.Object)
+     }
+ )
```

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

```
> new(
+     Class="Trajectoires",
+     temps=c(1,2,4,8),
+     traj=matrix(1:8,nrow=2)
+ )
```

```
 ~~~ Trajectoires : initiateur ~~~
*** Class Trajectoires, method Show ***
* Temps = [1] 1 2 4 8
* Traj  (limité à une matrice 10x10) =
   [,1] [,2] [,3] [,4]
I1 1    3    5    7
I2 2    4    6    8
* ... ...
******* Fin  Show(trajectoires) *******
```

La définition d'un initiateur désactive le vérificateur. Dans notre cas, `temps` peut à nouveau comporter plus ou moins de valeurs que de colonnes dans `traj`.

```
> new(
+     Class="Trajectoires",
+     temps=c(1,2,48),
+     traj=matrix(1:8,nrow=2)
+ )
```

```
 ~~~ Trajectoires : initiateur ~~~
*** Class Trajectoires, method Show ***
* Temps = [1]  1  2 48
* Traj  (limité à une matrice 10x10) =
   [,1] [,2] [,3] [,4]
I1 1    3    5    7
I2 2    4    6    8
* ... ...
******* Fin  Show(trajectoires) *******
```

Pour utiliser un initiateur et un vérificateur dans le même objet, il faut donc appeler "manuellement" le vérificateur grâce à l'instruction `validObject`. Notre initiateur *incluant* le vérificateur devient :

```
> setMethod(
+     f="initialize",
+     signature="Trajectoires",
+     definition=function(.Object,temps,traj){
+         cat(" ~~~~~ Trajectoires : initiateur ~~~~~\n")
```

```
+         if(!missing(traj)){
+             colnames(traj) <- paste("T",temps,sep="")
+             rownames(traj) <- paste("I",1:nrow(traj),sep="")
+             .Object@temps <- temps
+             .Object@traj <- traj
+             validObject(.Object)
+         }
+         return(.Object)
+     }
+ )
```

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

```
> new(
+     Class="Trajectoires",
+     temps=c(1,2,4,8),
+     traj=matrix(1:8,nrow=2)
+ )
```

```
 ~~~~~ Trajectoires : initiateur ~~~~~
  ~~~ Trajectoires : vérificateur ~~~
*** Class Trajectoires, method Show ***
* Temps = [1] 1 2 4 8
* Traj  (limité à une matrice 10x10) =
   T1 T2 T4 T8
I1 1  3  5  7
I2 2  4  6  8
* ... ...
******* Fin  Show(trajectoires) *******
```

```
> new(
+     Class="Trajectoires",
+     temps=c(1,2,48),
+     traj=matrix(1:8,nrow=2)
+ )
```

```
 ~~~~~ Trajectoires : initiateur ~~~~~
Error in dimnames(x) <- dn :
  la longueur de 'dimnames' [2] n'est
pas égale à l'étendue du tableau
```

Cette nouvelle définition a supprimé l'ancienne. Vous aurez noté la condition portant sur `missing(traj)` pour prendre en compte l'objet vide...

Un constructeur ne prend pas nécessairement pour argument les attributs de l'objet. Par exemple, si on sait (ça n'est pas le cas dans la réalité, mais imaginons) que l'IMC augmente de 0.1 toutes les semaines, on pourrait construire des trajectoires en fournissant le nombre de semaines et les IMC initiaux :

```
> setClass(
+     Class="TrajectoiresBis",
+     representation(
+         temps = "numeric",
```

```
+          traj = "matrix"
+      )
+ )
```

```
[1] "TrajectoiresBis"
```

```
> setMethod("initialize",
+     "TrajectoiresBis",
+     function(.Object,nbSemaine,IMCinit){
+         calculTraj <- function(init,nbSem){
+             return(init+0.1*nbSem)
+         }
+         traj <- outer(IMCinit,1:nbSemaine,calculTraj)
+         colnames(traj) <- paste("T",1:nbSemaine,sep="")
+         rownames(traj) <- paste("I",1:nrow(traj),sep="")
+         .Object@temps <- 1:nbSemaine
+         .Object@traj <- traj
+         return(.Object)
+     }
+ )
```

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

```
> new(Class="TrajectoiresBis",nbSemaine=4,IMCinit=c(16,17,15.6))
```

```
An object of class "TrajectoiresBis"
Slot "temps":
[1] 1 2 3 4

Slot "traj":
     T1   T2   T3   T4
I1 16.1 16.2 16.3 16.4
I2 17.1 17.2 17.3 17.4
I3 15.7 15.8 15.9 16.0
```

Il ne peut y avoir qu'un seul initiateur par classe. Il faut donc qu'il soit le plus général possible. La définition ci-dessus interdirait la construction d'une trajectoire à partir d'une matrice. Elle est donc fortement déconseillée car trop spécifique. Au final, il vaut mieux laisser ce genre de transformation aux constructeurs grand public.

## 6.3 Constructeur grand public

Comme nous l'avions dit en introduction, le (gentil) programmeur, ayant conscience du fait que `new` n'est pas une fonction *sympathique*, ajoute des constructeurs "grand public". Cela se fait grâce à une fonction (fonction classique, pas nécessairement une méthode S4) portant généralement le nom de la classe. Dans notre cas, ça sera la fonction `trajectoires`.

```
> trajectoires <- function(temps,traj){
+     cat("~~~~~ Trajectoires : constructeur ~~~~~\n")
+     new(Class="Trajectoires",temps=temps,traj=traj)
+ }
> trajectoires(temps=c(1,2,4),traj=matrix(1:6,ncol=3))
```

```
~~~~~ Trajectoires : constructeur ~~~~~
 ~~~~~ Trajectoires : initiateur ~~~~~
  ~~~ Trajectoires : vérificateur ~~~
*** Class Trajectoires, method Show ***
* Temps = [1] 1 2 4
* Traj  (limité à une matrice 10x10) =
   T1 T2 T4
I1 1  3  5
I2 2  4  6
* ... ...
******* Fin  Show(trajectoires) *******
```

L'intérêt est de pouvoir faire un traitement plus sophistiqué. Par exemple, dans un grand nombre de cas, Tam mesure les trajectoires toutes les semaines et elle les stocke dans une matrice. Elle souhaite donc avoir le choix : soit définir l'objet `trajectoires` simplement en donnant une matrice, soit en donnant une matrice et les temps :

```
> trajectoires <- function(temps,traj){
+     if(missing(temps)){temps <- 1:ncol(traj)}
+     new(Class="Trajectoires",temps=temps,traj=traj)
+ }
> trajectoires(traj=matrix(1:8,ncol=4))
```

```
 ~~~~~ Trajectoires : initiateur ~~~~~
  ~~~ Trajectoires : vérificateur ~~~
*** Class Trajectoires, method Show ***
* Temps = [1] 1 2 3 4
* Traj  (limité à une matrice 10x10) =
   T1 T2 T3 T4
I1 1  3  5  7
I2 2  4  6  8
* ... ...
******* Fin  Show(trajectoires) *******
```

R accepte parfois que deux entités différentes portent le même nom. Dans le cas présent, il est possible de définir une fonction portant le même nom qu'une classe. Un inconvénient à cela est qu'on ne sait plus ensuite de quoi on parle. Nous vous conseillons plutôt de donner à la classe un nom avec une majuscule et à la fonction constructeur le même nom mais avec une minuscule.

Contrairement à l'initiateur, on peut définir plusieurs constructeurs. Toujours sous l'hypothèse que l'IMC augmente de 0.1 toutes les semaines, on peut définir `trajectoiresRegulieres` :

```
> trajectoiresRegulieres <- function(nbSemaine,IMCinit){
+     funcInit <- function(init,nbSem){return(init+0.1*nbSem)}
+     traj <- outer(IMCinit,1:nbSemaine,funcInit)
+     temps <- 1:nbSemaine
+     return(new(Class="Trajectoires",temps=temps,traj=traj))
+ }
> trajectoiresRegulieres(nbSemaine=3,IMCinit=c(14,15,16))
```

```
 ~~~~~ Trajectoires : initiateur ~~~~~
  ~~~ Trajectoires : vérificateur ~~~
*** Class Trajectoires, method Show ***
* Temps = [1] 1 2 3
* Traj  (limité à une matrice 10x10) =
   T1   T2   T3
I1 14.1 14.2 14.3
I2 15.1 15.2 15.3
I3 16.1 16.2 16.3
* ... ...
******* Fin  Show(trajectoires) *******
```

Ainsi, les deux constructeurs font tous les deux appel à l'initiateur. D'où l'importance d'un initiateur *généraliste*.

## 6.4 Petit bilan

Lors de la construction d'un objet, il y a donc trois endroits où il est possible d'effectuer des opérations : dans la fonction de construction, dans l'initiateur et dans le vérificateur. Le vérificateur ne peut faire que vérifier, il ne permet pas de modifier l'objet. Par contre, on peut l'appeler sur un objet déjà construit. Pour ne pas trop se mélanger, il est donc préférable de spécialiser chacun de ces opérateurs et de lui réserver certaines tâches précises :

- **La fonction de construction** est celle qui sera appelée par l'utilisateur. Elle est la plus générale et peut prendre des arguments variables, éventuellement des arguments qui ne sont pas des attributs de l'objet. Elle transforme ensuite ses arguments en des attributs. Nous conseillons donc de lui confier la transformation de ses arguments en futurs attributs (comme `trajectoiresRegulieres` a préparé des arguments pour `new("Trajectoires")`).
  La fonction de construction se termine toujours par `new`.
- **L'initiateur** est appelé par `new`. Il est chargé de donner à chaque attribut sa valeur, après modification éventuelle. On peut le charger des tâches qui doivent être effectuées pour tous les objets, quels que soit les constructeurs qui les appellent (comme le renommage des lignes et des colonnes de `Trajectoires`).
  Si l'initiateur n'a pas été défini, R appelle un initiateur par défaut qui se contente d'affecter les valeurs aux attributs puis d'appeler le validateur.
  Au final, l'initiateur doit appeler le vérificateur (l'initiateur par défaut appelle le vérificateur, l'initiateur défini par l'utilisateur doit faire un appel explicite).

– **Le vérificateur** contrôle la cohérence interne de l'objet. Il peut par exemple interdire certaines valeurs à certains attributs, vérifier que la taille de attributs est conforme à ce qui est attendu,... Il ne peut pas modifier les valeurs des attributs, il doit se contenter de vérifier qu'ils suivent des règles.

Ce découpage des tâches a pour avantage de bien séparer les choses. Par contre, il n'est pas le meilleur en terme d'efficacité:

*"En effet,* précise un relecteur, *l'initiateur par défaut de R est bien plus efficace que les initiateurs écrits par les programmeurs. Il est donc intéressant de l'utiliser. Pour cela, il suffit de* **ne pas** *le définir. Dans cette optique :*
– **Le constructeur** *fait tout le travail préparatoire.*
– **L'initiateur** *n'est pas défini pas l'utilisateur. C'est donc l'initiateur par défaut qui est appelé.*
– **Le vérificateur** *contrôle la cohérence interne de l'objet."*

Cette manière de faire, un tout petit peu moins claire pour le débutant, est plus efficace par la suite.

Pour le néophyte, savoir quelle méthode est appelée, et quand, est un vrai casse-tête. `new` utilise l'initiateur s'il existe, le vérificateur sinon (mais pas les deux sauf appel explicite, ce que nous avons fait). D'autres instructions, comme `as` (section 9.8 page 77) font appel aux initiateurs et vérificateurs. Dans le doute, pour bien comprendre qui est appelé et quand, nous avons ajouté une petite ligne en tête de fonction qui affiche le nom de la méthode utilisée. "Top moche" ont commenté certains relecteurs. Hélas, ils ont raison. Mais la pédagogie prime ici sur l'esthétique... D'ailleurs, quand nous en serons à l'héritage, les choses deviendront un peu plus compliquées et cet affichage "top moche" sera un peu plus nécessaire...

# Chapitre 7

# Accesseur

Nous en avons déjà parlé, utiliser `@` en dehors d'une méthode est fortement déconseillé... Pourtant, il est nécessaire de pouvoir récupérer les valeurs des attributs. C'est le rôle des *accesseurs*. Dans la langue de Molière, on les appelle les *selecteurs* et les *affectants*. En français option informatique, on parle plutôt de *getteurs* et *setteurs*, francisation de *get* et *set*.

## 7.1 Les getteurs

Un getteur est une méthode qui renvoie la valeur d'un attribut. En programmation classique, un grand nombre de fonctions prennent une variable et retournent une partie de ces arguments. Par exemple, `names` appliqué à un `data.frame` retourne les noms des colonnes ; `nrow` appliqué au même `data.frame` donne le nombre de lignes. Et ainsi de suite.

Pour nos trajectoires, nous pouvons définir plusieurs getteurs : bien sûr, il nous en faut un qui renvoie `temps` et un qui renvoie `traj`. Nos getteurs étant des méthodes nouvelles pour R, il faut les déclarer grâce à `setGeneric` puis les définir simplement grâce à un `setMethod` :

```
> ### Getteur pour `temps'
> setGeneric("getTemps",
+     function(object){standardGeneric("getTemps")}
+ )
```

```
[1] "getTemps"
```

```
> setMethod("getTemps","Trajectoires",
+     function(object){
+         return(object@temps)
+     }
+ )
```

```
[1] "getTemps"
```

```
> getTemps(trajCochin)
```

```
[1] 1 2 4 5
```

```
> ### Getteur pour `traj'
> setGeneric("getTraj",
+     function(object){standardGeneric("getTraj")}
+ )
```

```
[1] "getTraj"
```

```
> setMethod("getTraj","Trajectoires",
+     function(object){
+         return(object@traj)
+     }
+ )
```

```
[1] "getTraj"
```

```
> getTraj(trajCochin)
```

```
     [,1] [,2] [,3] [,4]
[1,] 15.0 15.1 15.2 15.2
[2,] 16.0 15.9 16.0 16.4
[3,] 15.2   NA 15.3 15.3
[4,] 15.7 15.6 15.8 16.0
```

Mais on peut aussi faire des getteurs plus élaborés. Par exemple, on peut avoir régulièrement besoin de l'IMC au temps d'inclusion :

```
> ### Getteur pour les IMC à l'inclusion
> ###    (première colonne de `traj')
> setGeneric("getTrajInclusion",
+     function(object){standardGeneric("getTrajInclusion")}
+ )
```

```
[1] "getTrajInclusion"
```

```
> setMethod("getTrajInclusion","Trajectoires",
+     function(object){
+         return(object@traj[,1])
+     }
+ )
```

```
[1] "getTrajInclusion"
```

```
> getTrajInclusion(trajCochin)
```

```
[1] 15.0 16.0 15.2 15.7
```

## 7.2 Les setteurs

Un setteur est une méthode qui affecte une valeur à un attribut. Sous R, l'affectation est faite par `<-`. Sans entrer dans les méandres du programme, l'opérateur `<-` fait en réalité appel à une méthode spécifique. Par exemple, quand on utilise `names(data) <- "A"`, R fait appel à la fonction `names<-`. Cette fonction duplique l'objet `data`, modifie l'attribut `names` de ce nouvel objet puis remplace `data` par ce nouvel objet. Nous allons faire pareil pour les attributs de nos `trajectoires`. `setTemps<-` permettra de modifier l'attribut `temps`. Pour cela, on utilise la fonction `setReplaceMethod`

```
> setGeneric("setTemps<-",
+     function(object,value){standardGeneric("setTemps<-")}
+ )
```

```
[1] "setTemps<-"
```

```
> setReplaceMethod(
+     f="setTemps",
+     signature="Trajectoires",
+     definition=function(object,value){
+         object@temps <- value
+         return(object)
+     }
+ )
```

```
[1] "setTemps<-"
```

```
> (setTemps(trajCochin) <- 1:3)
```

```
[1] 1 2 3
```

Tout l'intérêt du setteur est de pouvoir faire des contrôles. Comme dans `initialize`, nous pouvons appeler explicitement le vérificateur :

```
> setReplaceMethod(
+     f="setTemps",
+     signature="Trajectoires",
+     definition=function(object,value){
+         object@temps <- value
+         validObject(object)
+         return(object)
+     }
+ )
```

```
[1] "setTemps<-"
```

```
> setTemps(trajCochin) <- c(1,2,4,6)
```

```
  ~~~ Trajectoires : vérificateur ~~~
```

```
> setTemps(trajCochin) <- 1:4
```

```
  ~~~ Trajectoires : vérificateur ~~~
Error in validityMethod(object) :
  [Trajectoire:validation] Le
nombre de mesures
              temporelles ne correspond pas au
nombre de
              colonnes de la matrice
```

## 7.3 Les opérateurs [ et [<-

Il est également possible de définir les getteurs grâce à l'opérateur `[` et les setteurs grâce à `[<-`. Cela se fait comme pour une méthode quelconque en précisant la classe et la fonction à appliquer. Cette fonction prend quatre arguments :

- `x` est l'objet.
- `i` désigne l'attribut auquel nous voulons accéder.
- Si l'attribut désigné par `i` est complexe (une matrice, une liste,...), `j` permet d'accéder à un élément particulier.
- Enfin, `drop` est un boléen permettant de préciser si ce qui est retourné doit garder sa structure initiale ou non (par exemple si une matrice d'une seule ligne doit être considéré comme un vecteur ou comme une matrice).

Dans notre exemple, `[` peut simplement retourner l'attribut correspondant à `i`.

```
> setMethod(
+     f="[",
+     signature="Trajectoires",
+     definition=function(x,i,j,drop){
+         switch(EXPR=i,
+             "temps"={return(x@temps)},
+             "traj"={return(x@traj)}
+         )
+     }
+ )
```

```
[1] "["
```

Ensuite, l'appel de la fonction `[` se fait sous la forme `x[i="Attribut1", j=3, drop=FALSE]`, ou sous la forme simplifiée `x["Attribut1",3,FALSE]`. Dans notre cas, `j` et `drop` n'étant pas utilisés, on peut simplement les omettre :

```
> trajCochin[i="temps"]
```

```
[1] 1 2 4 6
```

```
> trajCochin["traj"]
```

```
     [,1] [,2] [,3] [,4]
[1,] 15.0 15.1 15.2 15.2
[2,] 16.0 15.9 16.0 16.4
```

```
[3,] 15.2   NA 15.3 15.3
[4,] 15.7 15.6 15.8 16.0
```

La définition que nous venons d'écrire n'offre aucune protection contre les erreurs typographiques :

```
> trajCochin["trak"]
```

```
NULL
```

Il est donc important de finir l'énumération par le comportement à adopter quand l'attribut n'existe pas :

```
> setMethod(
+     f="[",
+     signature="Trajectoires",
+     definition=function(x,i,j,drop){
+         switch(EXPR=i,
+             "temps"={return(x@temps)},
+             "traj"={return(x@traj)},
+             stop("Cet attribut n'existe pas !")
+         )
+
+     }
+ )
```

```
[1] "["
```

Nous sommes maintenant à l'abri d'une erreur de typographie.

`drop` est un argument à utiliser avec précaution. En effet, il a pour vocation de modifier le type de l'objet. Par exemple, si `M` est une matrice, quel est le type de `M[,b]` ? Cela dépend de `b`. Si `b` est un vecteur, alors `M[,b]` est une matrice. Si `b` est un entier, alors `M[,b]` est un vecteur... Comme toujours, un comportement *qui dépend* est à proscrire.

Les setteurs se définissent selon le même principe grâce à l'opérateur `[<-`. On utilise `setReplaceMethod`. Là encore, une fonction décrit le comportement à adopter. Là encore, il est important de veiller à contrôler une éventuelle faute de frappe. :

```
> setReplaceMethod(
+     f="[",
+     signature="Trajectoires",
+     definition=function(x,i,j,value){
+         switch(EXPR=i,
+             "temps"={x@temps<-value},
+             "traj"={x@traj<-value},
+             stop("Cet attribut n'existe pas !")
+         )
+         validObject(x)
+         return(x)
+     }
+ )
```

```
[1] "[<-"
```

```
> trajCochin["temps"]<-2:5
```

```
  ~~~ Trajectoires : vérificateur ~~~
```

Dans nos définitions de `[` et `[<-`, nous avons listé les différents attributs possibles pour `i` (`i=="temps"` et `i=="traj"`). Il serait également possible de les numéroter (`i==1` et `i==2`). L'accès au premier attribut se ferait alors via `trajCochin[1]` à la place de `trajCochin["temps"]`. C'est bien évidemment totalement impropre : on se mettrait à la merci de l'erreur typographique `trajCochin[2]` à la place de `trajCochin[1]` (alors que `trajCochin["trmps"]` ne présente pas de danger puisqu'il propovoque une erreur).

## 7.4 `[`, `@` ou get ?

Quand doit-on utiliser `get`, quand doit-on utiliser `@`, quand doit-on utiliser `[` ?

`@` est à réserver exclusivement aux méthodes internes à la classe : si une méthode de partition `Partition` a besoin de `traj` (c'est à dire d'un attribut d'une autre classe), il lui est formellement interdit d'utiliser `@` pour accéder aux attributs directement, elle doit passer par `[` ou par `get`.

A l'intérieur d'une classe (si une méthode de `Partition` a besoin de `nbCluster`), il y a deux écoles : ceux qui utilisent `@` et ceux qui utilisent `[` (ou `get`).

Entre `get` et `[`, il n'y a pas vraiment de différence, c'est simplement un jeu d'écriture : `getTemps(trajCochin)` ou `trajCochin["temps"]` ? La première notation rappelle les autres langages objets, la deuxième est plus spécifique à R. Dans le cas d'un package et donc de méthodes qui ont pour vocation d'être utilisées par d'autres, `[` sera plus conforme à la syntaxe “classique” de R, donc plus intuitif.

# Troisième partie

# Pour aller plus loin

La suite comprend les signatures, l'héritage et quelques autres concepts avancés. Si vous êtes vraiment novice en programmation objet, il est peut-être temps de faire une petite pause pour intégrer ce qui vient d'être présenté. Ce que nous venons de voir suffit largement à faire quelques petites classes. Vos difficultés vous permettront entre autres de mieux comprendre le pourquoi de ce qui va suivre...

# Chapitre 8

# Méthodes utilisant plusieurs arguments

Nous avons fait notre premier objet. La suite est plus liée aux interactions entre les objets. Il est donc temps de définir nos deux autres objets. En premier lieu, `Partition`.

## 8.1 Le problème

Ceci est un manuel. Nous n'allons donc pas définir partition et le cortège de méthodes qui accompagne chaque nouvelle classe mais simplement ce dont nous avons besoin :

```
> setClass(
+     Class="Partition",
+     representation=representation(
+         nbGroupes="numeric",
+         part="factor"
+     )
+ )
```

```
[1] "Partition"
```

```
> setMethod(f="[",signature="Partition",
+     definition=function(x,i,j,drop){
+         switch(EXPR=i,
+             "part"={return(x@part)},
+             "nbGroupes"={return(x@nbGroupes)},
+             stop("Cet attribut n'existe pas !")
+         )
+
+     }
+ )
```

```
[1] "["
```

```
> partCochin <- new(Class="Partition",nbGroupes=2,
+     part=factor(c("A","B","A","B"))
```

```
+ )
> partStAnne <- new(Class="Partition",nbGroupes=2,
+     part=factor(rep(c("A","B"),c(50,30)))
+ )
```

Nous supposerons de plus que `part` est toujours composé de lettres majuscules allant de `A` à `LETTERS[nbGroupes]` (il faudra bien préciser dans la documentation de cette classe que le nombre de groupes doit être inférieur à 26). Nous pouvons nous permettre une telle supposition en programmant `initialize` et `part<-` de manière à toujours vérifier que c'est le cas.

Nous avons donc un objet `trajCochin` de classe `Trajectoires` et un découpage de cet objet en un objet `partCochin` de classe `Partition`. Lorsque nous représentons `trajCochin` graphiquement, nous obtenons un faisceau de courbes multicolores. Maintenant que les trajectoires sont associées à des groupes, il serait intéressant de dessiner les courbes en donnant une couleur à chaque groupe. Pour cela, il va falloir définir une méthode `plot` qui prendra comme argument un objet `Trajectoires` *plus* un objet `Partition`. C'est possible grâce à l'utilisation de `signature`.

## 8.2 signature

La signature, nous l'avons déjà vu lors de la définition de nos premières méthodes section 5.1 page 35, est le deuxième argument fourni à `setMethod`. Jusqu'à présent, nous utilisions des signatures simples constituées d'une seule classe. Dans `setMethod(f="plot",signature="Trajectoires",definition=function)`, la signature est simplement `"Trajectoires"`

Pour avoir des fonctions dont le comportement dépend de plusieurs objets, il est possible de définir un *vecteur signature* comportant plusieurs classes. Ensuite, quand nous appelons une méthode, R cherche la signature qui lui correspond le mieux et applique la fonction correspondante. Un petit exemple s'impose. Nous allons définir une fonction `essai`

```
> setGeneric("essai",function(x,y,...){standardGeneric("essai")})
```

```
[1] "essai"
```

Cette fonction a pour mission d'adopter un certain comportement si son argument est `numeric`, un autre comportement si s'est un `character`.

```
> setMethod(
+     f="essai",
+     signature="numeric",
+     function(x,y,...){cat("x est un numeric = ",x,"\n")}
+ )
```

```
[1] "essai"
```

Á ce stade, `essai` sait afficher les `numeric`, mais ne sait pas afficher les `character` :

```
> ### 3.17 étant un numeric, R va applique la methode
> ### essai pour les numeric
> essai(3.17)
```

```
x est un numeric =  3.17
```

```
> essai("E")
```

```
Error in function (classes, fdef, mtable)  :
  unable to find an
inherited method for function "essai", for signature "character"
```

Pour que `essai` soit compatible avec les `character`, il faut définir la méthode avec la signature `character` :

```
> setMethod(
+     f="essai",
+     signature="character",
+     function(x,y,...){cat("x est character = ",x,"\n")}
+ )
```

```
[1] "essai"
```

```
> ### 'E' étant un character, R applique maintenant la méthode
> ### essai pour les characters
> essai("E")
```

```
x est character =  E
```

Plus compliqué, nous souhaitons que `essai` ait un comportement différent si on combine un `numeric` et un `character`.

```
> ### Pour une méthode qui combine numeric et character :
> setMethod(
+     f="essai",
+     signature=c(x="numeric",y="character"),
+     definition=function(x,y,...){
+         cat("Plus compliqué :")
+         cat("x est un num =",x," ET y un est un char =",y,"\n")
+     }
+ )
```

```
[1] "essai"
```

Maintenant, R connait trois méthodes à appliquer à `essai` : la première est appliquée si l'argument de `essai` est un `numeric` ; la deuxième est appliquée si l'argument de `essai` est un `character` ; la troisième est appliquée si `essai` a deux arguments, un `numeric` puis un `character`

```
> essai(3.2,"E")
```

```
Plus compliqué :x est un num = 3.2  ET y un est un char = E
```

```
> essai(3.2)
```

```
x est un numeric =  3.2
```

```
> essai("E")
```

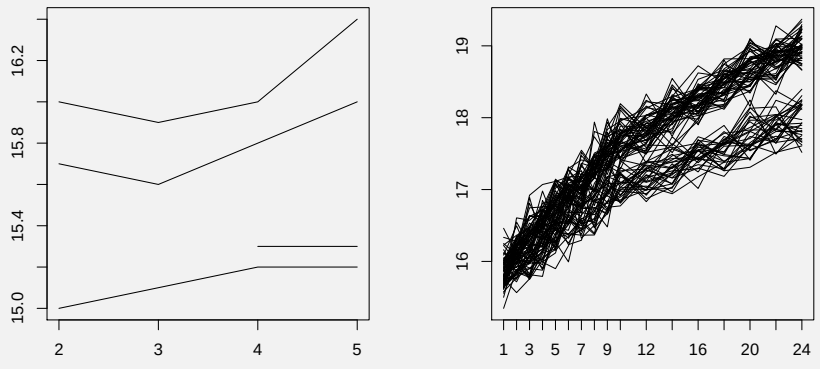
```
x est character =  E
```

Retour à `miniKml` (le package que nous sommes en train de construire). De la même manière, nous avions défini `plot` pour la signature `Trajectoires`, nous allons maintenant définir `plot` pour la signature `c("Trajectoires","Partition")`. Nous pourrons ensuite représenter graphiquement les trajectoires selon des partitions spécifiques.

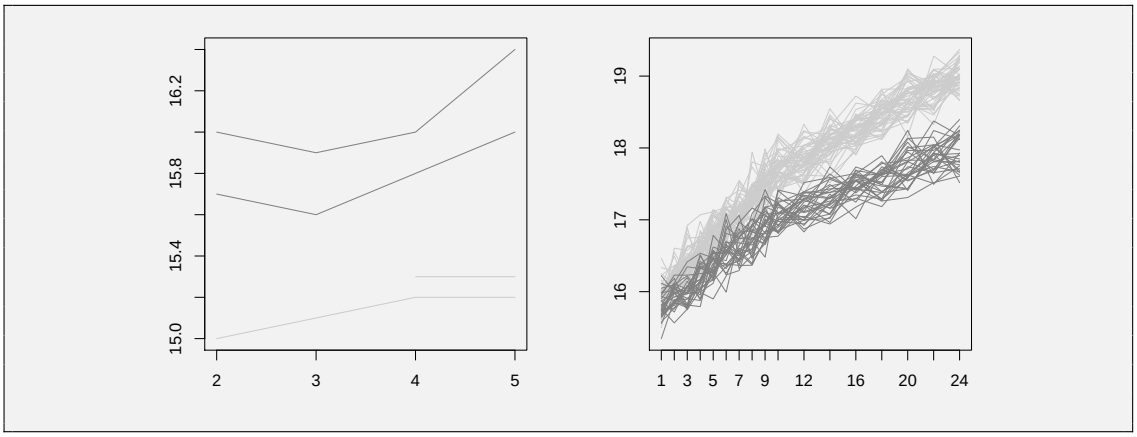
```
> setMethod(
+     f="plot",
+     signature=c(x="Trajectoires",y="Partition"),
+     definition=function(x,y,...){
+         matplot(x@temps,t(x@traj[y@part=="A",]),
+                 ylim=range(x@traj,na.rm=TRUE),
+                 xaxt="n",type="l",ylab="",xlab="",col=2
+         )
+         for (i in 2:y@nbGroupes){
+             matlines(x@temps,t(x@traj[y@part==LETTERS[i],]),
+                 xaxt="n",type="l",col=i+1
+             )
+         }
+         axis(1,at=x@temps)
+     }
+ )
```

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

```
> ### Plot pour `Trajectoire'
> plot(trajCochin)
> plot(trajStAnne)
```



```
> ### Plot pour `Trajectoire' et `Partition'
> plot(trajCochin,partCochin)
> plot(trajStAnne,partStAnne)
```




Plutôt élégant, n'est-il pas ?

## 8.3 missing

Il est également possible de définir une méthode ayant un certain comportement si elle a un unique argument, un autre comportement si elle en a plusieurs. Cela est possible grâce à `missing`. `missing` est vrai si l'argument *est* manquant :

```
> setMethod(
+     f="essai",
+     signature=c(x="numeric",y="missing"),
+     definition=function(x,y,...){
+         cat("  x est numeric = ",x," et y est `missing'\n")
+     }
+ )
```

```
[1] "essai"
```

```
> ### Méthode sans y donc utilisant le missing
> essai(3.17)
```

```
  x est numeric =  3.17  et y est `missing'
```

```
> ### Methode avec y='character'
> essai(3.17,"E")
```

```
Plus compliqué :x est un num = 3.17  ET y un est un char = E
```

## 8.4 Nombre d'arguments d'une signature

Sans entrer dans les méandres internes de R, quelques petites règles concernant les signatures : une signature doit comporter autant d'arguments que sa méthode générique, ni plus, ni moins. Cela signifie qu'il n'est pas possible de définir une méthode pour `print` qui prendra en compte deux arguments puisque `print` a pour signature `x`. De même, `plot` ne peut être défini que pour deux arguments, impossible d'en préciser un troisième dans la signature.

## 8.5 ANY

Réciproquement, la signature doit comporter tous les arguments. Jusqu'à présent, nous ne nous en sommes pas soucié et nous avons défini des `plot` avec un seul argument. C'est simplement une facilité d'écriture : R est passé dernière nous et a ajouté le deuxième argument. Comme nous n'avions pas précisé le type de ce deuxième argument, il en a conclu que la méthode devait s'appliquer quel que soit le type du deuxième argument. Pour le déclarer explicitement, il existe un argument spécial, la classe originelle, la cause première : `ANY` (plus de détail sur `ANY` section 9.2 page 70). Donc, quand nous omettons un argument, R lui donne pour nom `ANY`.

La fonction `showMethods`, la même qui nous permettait de voir les toutes les méthodes existant pour un objet section 5.4 page 41, permet d'afficher la liste des signatures que R connait pour une certaine méthode :

```
> showMethods(essai)
```

```
Function: essai (package .GlobalEnv)
x="character", y="ANY"
x="numeric", y="ANY"
x="numeric", y="character"
x="numeric", y="missing"
```

Comme vous pouvez le constater, la liste ne contient pas les signatures que nous avons définies, mais des signatures complétées : les arguments que nous n'avions pas précisés (à savoir `y` dans les cas `x="character"` et `x="numeric"`) ont été remplacés par `"ANY"`.

Plus précisément, `ANY` n'est utilisé que si aucun argument autre ne convient. Dans le cas de `essai`, si `x` est un `numeric`, R hésite entre trois méthodes. En premier lieu, il essaie de voir si `y` a un type défini par ailleurs. Si `y` est un `character`, la méthode utilisée sera celle correspondant à `(x="numeric",y="character")`. Si `y` n'est pas un `character`, R ne trouve pas de correspondance exacte entre `y` et un type, il utilise donc la méthode fourre-tout : `x="numeric", y="ANY"`.

# Chapitre 9

# Héritage

L'héritage est l'un des concepts clefs de la programmation objet. Il permet de réutiliser des pans entiers de programme à peu de frais. Dans notre cas, nous devons maintenant définir `TrajDecoupees`. Il va nous falloir coder l'objet, les constructeurs, les setteurs et le getteurs, l'affichage... Tout. Les plus astucieux d'entre nous sont déjà en train de se dire qu'il suffira de faire un copier-coller de méthodes créées pour `Trajectoires` et de les adapter. La programmation objet permet de faire mieux : nous allons définir `TrajDecoupees` comme objet *héritier* de `Trajectoires`.

## 9.1 Principe

L'héritage est un principe de programmation permettant de créer une nouvelle classe à partir d'une classe déjà existante, la nouvelle classe étant une *spécialisation* de la précédente. On dit de la nouvelle classe qu'elle *hérite* de la classe à partir de laquelle elle est définie. La nouvelle classe s'appelle donc *classe fils* alors que la classe ayant servi à la création est la *classe père*.

Plus précisément, la classe *fils* est une spécialisation de la classe *père* dans le sens où elle peut faire tout ce que fait la classe père *plus* de nouvelles choses. Prenons un exemple, classique en programmation objet. La classe `Vehicule` est définie comme comprenant un attribut `vitesse` et une méthode `tempsDeParcours`, une méthode qui permet de calculer le temps mis pour parcourir une certaine distance. C'est donc une classe assez générale comprenant tout type de véhicule, de la trottinette au pétrolier.

On souhaite maintenant définir la classe `Voiture`. Cette classe peut être considérée comme un véhicule particulier. On peut donc définir `Voiture` comme classe *héritière* de `Vehicule`. Un objet `Voiture` aura les mêmes arributs et méthodes que `Vehicule`, plus des attributs et méthodes propres comme `NombreDePortes`. Cet attribut fait sens pour une voiture, mais ferait bien moins de sens pour le concept de véhicule au sens large du terme (parce qu'il serait ridicule de parler de portes pour une trottinette).

Formellement, une classe `Fils` peut hériter d'une classe `Pere` quand `Fils` contient au moins tous les attributs de `Pere` (plus éventuellement d'autres). Le fait d'hériter rend toutes les méthodes de `Pere` disponibles pour `Fils` : chaque fois qu'on utilisera

une méthode sur un objet de classe `Fils`, R cherchera si cette méthode existe. S'il ne la trouve pas dans la liste des méthodes spécifiques à `Fils`, il cherchera dans les méthodes de `Pere`. S'il la trouve, il l'appliquera. S'il ne la trouve pas, il cherchera dans les méthodes dont `Pere` hérite. Et ainsi de suite.

On représente le lien qui unit le père et le fils par une flèche allant du fils vers le père. Cela symbolise que lorsqu'une méthode n'est pas trouvée pour le fils, R "suit" la flèche et cherche dans les méthodes du père : `classPere` ⟵ `classFils` ou encore :

`Vehicule` ⟵ `Voiture`

On note classiquement le père à gauche du fils (ou au dessus) puisqu'il est défini avant.

## 9.2 Père, grand-père et `ANY`

Nous venons de le voir, quand R ne trouve pas une méthode pour un objet, le principe de l'héritage lui demande de chercher parmi les méthodes du père. Le principe est récursif. S'il ne trouve pas chez le père, il cherche chez le grand-père, et ainsi de suite. Sur notre exemple, nous pourrions considérerer une classe de voiture particulière, les voitures sportives.

`Vehicule` ⟵ `Voiture` ⟵ `Sportive`

Si R ne trouve pas une méthode pour `Sportive`, il cherche parmi celle de `Voiture`. S'il ne trouve pas dans `Voiture`, il cherche dans `Vehicule` et ainsi de suite. Se pose alors la question de l'origine, de l'ancêtre ultime, la racine des racines. Chez les hommes, c'est -selon certaines sources non vérifiées- Adam. Chez les objets, c'est `ANY`. `ANY` est la classe première, celle dont toutes les autres héritent. Une classe créée de toute pièce sans que le programmeur ne la définisse comme héritière à partir d'une autre (comme toutes celles que nous avons créées dans les chapitres précédents) est considérée par R comme héritière de `ANY`. Donc, si une méthode n'est pas trouvée pour une classe `Fils`, elle sera cherchée dans la classe `Pere`, puis `GrandPere` et ainsi de suite. Si elle n'est pas trouvée parmi les ancêtres (ou si `Fils` n'a pas de père), elle sera cherchée dans la classe `ANY`. Si elle n'existe pas pour `ANY`, une erreur est affichée.

## 9.3 `contains`

Nous allons donc définir `TrajDecoupees` comme héritière de `Trajectoires`. Pour cela, on déclare l'objet en ajoutant l'argument `contains` suivi du nom de la classe père.

```
> setClass(
+     Class="TrajDecoupees",
+     representation=representation(listePartitions="list"),
+     contains="Trajectoires"
+ )
```

```
 ~~~~~ Trajectoires : initiateur ~~~~~
 ~~~~~ Trajectoires : initiateur ~~~~~
[1] "TrajDecoupees"
```

```
> tdPitie <- new("TrajDecoupees")
```

```
 ~~~~~ Trajectoires : initiateur ~~~~~
```

## 9.4 unclass

`TrajDecoupees` contient donc tous les attributs de `Trajectoires` plus son attribut personnel `listePartitions` (attribut qui contiendra une liste de partition).

Pour l'instant, il n'est pas possible de le vérifier directement. En effet, si nous essayons de 'voir' `tdCochin`, nous obtenons l'affichaqe d'un objet `Trajectoires`, et non pas une `TrajDecoupees`.

```
> tdPitie
```

```
*** Class Trajectoires, method Show ***
* Temps = numeric(0)
* Traj  (limité à une matrice 10x10) =
* ... ...
******* Fin  Show(trajectoires) *******
```

Voilà qui appelle quelques commentaires : `TrajDecoupees` est un héritier de `Trajectoires`. A chaque fois qu'une fonction est appelée, R cherche cette fonction pour la signature `TrajDecoupees`. S'il ne trouve pas, il cherche la fonction pour la classe parent à savoir `Trajectoires`. S'il ne trouve pas, il cherche dans les fonctions de `ANY`, c'est à dire les fonctions par défaut.

Voir un objet se fait grâce à `show`. Comme à ce stade `show` n'existe pas pour `TrajDecoupees`, c'est `show` pour `Trajectoires` qui est appelé à la place. Taper `tdPitie` ne nous montre donc pas l'objet tel qu'il est réellement mais via le prisme `show` pour `Trajectoires`. Il est donc urgent de définir une méthode `show` pour les `TrajDecoupees`.

Néanmoins, il serait intéressant de pouvoir jeter un oeil à l'objet que nous venons de créer. On peut pour cela utiliser `unclass`. `unclass` fait comme si un objet avait pour classe `ANY`. `unclass(tdPitie)` va donc appeler la méthode `show` comme si `tdPitie` avait pour classe `ANY`, et va donc utiliser la méthode `show` par défaut. Résultat, l'objet est affiché sans fioritures, certes, mais dans son intégralité.

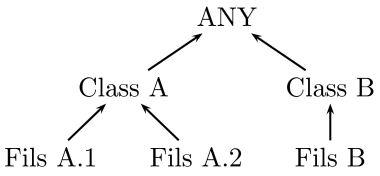
```
> unclass(tdPitie)
```

```
<S4 Type Object>
attr(,"listePartitions")
list()
attr(,"temps")
numeric(0)
attr(,"traj")
<0 x 0 matrix>
```

L'objet comporte donc effectivement les attributs de `Trajectoires` plus une liste.

## 9.5 Arbre d'héritage

Plusieurs classes peuvent hériter d'un même père. Au final, la représentation graphique des liens unissant les classes donne un arbre (un arbre informatique, désolé pour les poètes qui espéraient un peu de verdure dans cet ouvrage) :




Il est théoriquement possible d'hériter de plusieurs pères. La représentation graphique n'est alors plus un arbre mais un graphe. Mal utilisée, cette pratique peut être dangereuse. En effet, supposons qu'un objet `B` hérite à la fois de `A1` et `A2`. Si une méthode n'existant pas pour `B` est appelée, elle sera recherchée dans les méthodes *des* pères. Si la méthode existe pour `A1` ET pour `A2`, laquelle sera utilisée, celle de `A1` ou celle de `A2` ? Là réside une source de confusion. C'est encore plus problématique pour l'utilisation de `callNextMethod` (voir 9.7 page 75).

Pourtant, dans certains cas, l'héritage multiple semble plein de bon sens. Prenons un exemple : la classe `machineDeCalcul` dispose de méthodes donnant la précision d'un calcul. La classe `plastic` précise les propriétés physiques du plastique, par exemple ce qui se passe quand il brûle, sa résistance... un ordinateur est à la fois une `machineDeCalcul` et outil en `plastic`. Il est donc intéressant qu'il accède à la fois à la méthode `precision` (pour savoir ce qu'on peut lui demander) et à la méthode `brulle` pour savoir comment il réagira dans un incendie. La classe ordinateur pourrait donc légitimement hériter de deux pères.

Donc, l'héritage multiple est à manier avec précaution : il peut être utile, mais il faut prendre garde à ce qu'une classe n'hérite jamais de pères ayant des méthodes communes.

## 9.6 Voir la méthode en autorisant l'héritage

L'héritage est une force mais il peut être déroutant. Créons un deuxième objet `TrajDecoupees` :

```
> partCochin2 <- new("Partition",nbGroupes=3,
+                    part=factor(c("A","C","C","B"))
+                   )
```

```
> tdCochin <- new(
+     Class="TrajDecoupees",
```

```
+     temps=c(1,3,4,5),
+     traj=trajCochin@traj,
+     listePartitions=list(partCochin,partCochin2)
+ )
```

```
Error in .local(.Object, ...) :
  argument(s) inutilisé(s)
(listePartitions = list(<S4 object of class "Partition">, <S4
object of class "Partition">))
```

Ça ne marche pas... Pourquoi ? Pour le savoir, il est possible d'afficher la méthode `initialize` appelée par `new` :

```
> getMethod("initialize","TrajDecoupees")
```

```
Error in getMethod("initialize", "TrajDecoupees") :
  No method
found for function "initialize" and signature TrajDecoupees
```

Là encore, ça ne marche pas, mais cette fois la cause est plus simple à identifier : nous n'avons pas encore défini `initialize` pour `Partition`.

```
> existsMethod("initialize","TrajDecoupees")
```

```
[1] FALSE
```

Voilà confirmation de nos doutes. Pourtant, R semble tout de même exécuter un code puisqu'il détecte une erreur. Mais que se passe-t-il donc [1] ?

C'est un des effets indésirables de l'héritage, une sorte d'héritage involontaire. En effet, à l'appel de `new("TrajDecoupees")`, `new` cherche la fonction `initialize` pour `TrajDecoupees`. Comme il ne la trouve pas ET que `TrajDecoupees` hérite de `Trajectoires`, il remplace la méthode manquante par `initialize` pour `Trajectoires`.

Pour vérifier cela, deux méthodes : `hasMethods` permet de savoir si une méthode existe pour une classe donnée en prenant en compte l'héritage. Pour mémoire, quand `existsMethod` ne trouvait pas une méthode, elle renvoyait faux. Quand `hasMethod` ne trouve pas, elle cherche chez le père, puis chez le grand-père et ainsi de suite :

```
> hasMethod("initialize","TrajDecoupees")
```

```
[1] TRUE
```

Confirmation, `new` ne repart pas bredouille, il est effectivement réorienté vers une méthode héritée. Pour afficher la méthode en question, on peut utiliser `selectMethod`. `selectMethod` a globalement le même comportement que `getMethod`. Seule différence, lorsqu'il ne trouve pas une méthode, il cherche chez les ancêtres...

```
> selectMethod("initialize","TrajDecoupees")
```

1. Vous noterez au passage tous les trésors d'imagination développés par l'auteur pour ménager du suspense dans un livre de statistique-informatique...

```
Method Definition:

function (.Object, ...)
{
    .local <- function (.Object, temps, traj)
    {
        cat(" ~~~~~ Trajectoires : initiateur ~~~~~\n")
        if (!missing(traj)) {
            colnames(traj) <- paste("T", temps, sep = "")
            rownames(traj) <- paste("I", 1:nrow(traj), sep = "")
            .Object@temps <- temps
            .Object@traj <- traj
            validObject(.Object)
        }
        return(.Object)
    }
    .local(.Object, ...)
}

Signatures:
        .Object
target  "TrajDecoupees"
defined "Trajectoires"
```

Notre hypothèse était la bonne, `TrajDecoupees` utilise l'initiateur de `Partition`. Comme ce dernier ne connait pas l'attribut `listePartitions`, il retourne une erreur. Le mystère est maintenant éclairci.

Pour pouvoir définir des objets `TrajDecoupees` un peu plus complexes, il faut donc au préalable définir `initialize` pour `TrajDecoupees`

```
> setMethod("initialize","TrajDecoupees",
+     function(.Object,temps,traj,listePartitions){
+         cat(" ~~~~ TrajDecoupees : initiateur ~~~~\n")
+         if(!missing(traj)){
+             .Object@temps <- temps
+ # Affectation des attributs
+             .Object@traj <- traj
+             .Object@listePartitions <- listePartitions
+         }
+ # return de l'objet
+         return(.Object)
+     }
+ )
```

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

```
> tdCochin <- new(
+     Class="TrajDecoupees",
+     traj=trajCochin@traj,
+     temps=c(1,3,4,5),
+     listePartitions=list(partCochin,partCochin2)
```

```
+ )
```

```
 ~~~~ TrajDecoupees : initiateur ~~~~
```

Et voilà !

## 9.7 callNextMethod

Nous venons de le voir : lorsqu'il ne trouve pas une méthode, R dispose d'un mécanisme lui permettant de la remplacer par une méthode héritée. Il est possible de contrôler ce mécanisme et de forcer une méthode à appeler la méthode héritée. Ça permet entre autre chose de ré-utiliser du code sans avoir à le retaper. Cela se fait grâce à `callNextMethod`. `callNextMethod` n'est utilisable que dans une méthode. Elle a pour effet d'appeler *la méthode qui serait utilisée si la méthode actuelle n'existait pas.* Par exemple, considérons la méthode `print` pour `TrajDecoupees`. Actuellement, cette méthode n'existe pas. Pourtant, un appel a `print(tdCochin)` serait quand même exécuté grâce à l'héritage :

```
> print(tdCochin)
```

```
*** Class Trajectoires, method Print ***
* Temps = [1] 1 3 4 5
* Traj  =
     [,1] [,2] [,3] [,4]
[1,] 15.0 15.1 15.2 15.2
[2,] 16.0 15.9 16.0 16.4
[3,] 15.2   NA 15.3 15.3
[4,] 15.7 15.6 15.8 16.0
******* Fin  Print(trajectoires) *******
```

Comme `print` n'existe pas pour `TrajDecoupees`, c'est `print` pour `Trajectoires` qui est appelée. Autrement dit, la `nextMethod` de `print` pour `TrajDecoupees` est `print` pour `Trajectoires`. Un petit exemple et tout sera plus clair. Nous allons définir `print` pour `TrajDecoupees` [2].

```
> setMethod(
+     f="print",
+     signature="TrajDecoupees",
+     definition=function(x,...){
+         callNextMethod()
+         cat("L'objet contient également")
+         cat(length(x@listePartitions),"partition")
+         cat("\n***** Fin de  print(TrajDecoupees) *****\n")
+         return(invisible())
+     }
+ )
```

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

2. Naturellement, dans un cas réel, nous ferions beaucoup plus qu'afficher les trajectoires et le nombre de partitions.

```
> print(tdCochin)
```

```
*** Class Trajectoires, method Print ***
* Temps = [1] 1 3 4 5
* Traj  =
     [,1] [,2] [,3] [,4]
[1,] 15.0 15.1 15.2 15.2
[2,] 16.0 15.9 16.0 16.4
[3,] 15.2   NA 15.3 15.3
[4,] 15.7 15.6 15.8 16.0
******* Fin  Print(trajectoires) *******
L'objet contient également2 partition
***** Fin de  print(TrajDecoupees) *****
```

`callNextMethod` peut soit prendre des arguments explicites, soit aucun argument. Dans ce cas, les arguments qui ont été passés à la méthode actuelle sont intégralement passés à la méthode suivante.

Quelle est la méthode suivante, voilà toute la difficulté et l'ambigüité de `callNextMethod`. Dans la plupart des cas, les gens très forts savent, les moins forts ne savent pas. Mais là où cela devient totalement impropre, c'est que la méthode suivante peut *dépendre* de la structure d'une autre classe. Au final, personne ne peut savoir. Exemple : définissons une classe `A` qui hérite de la classe `B`, classe que quelqu'un d'autre a programmée. Quelle est la méthode suivante de `initialize` pour `A` ? Cela dépend. Comme `A` hérite, R va chercher dans l'ordre :

- `initialize` pour `B`
- `initialize` par défaut. Cette méthode se termine par un `validObject`.
- `validObject` pour `A`
- `validObject` pour `B`
- `validObject` par défaut.

Á partir de là, il n'est pas possible de savoir *parce que ça ne dépend plus de nous !* Si le programmeur de `B` a défini un initialize, il est appelé et donc peut-être que le `validObject` pour `B` sera appelé, peut-être pas. Sinon, c'est le `initialize` par défaut qui est appelé, et donc le `validObject` pour `A`, celui de `B` ou celui par défaut en fonction de ce qui existe. Il est donc très difficile de savoir ce que le programme fait vraiment.

Plus grave, si le programmeur de `B` supprime ou ajoute un `initialize`, cela peut changer le comportement de *notre* méthode. Par exemple, si `initialize` pour `B` n'existe pas et que `validObject` pour `A` existe, `initialize` par défaut est appelé puis `validObject` pour `A`. Si la méthode `initialize` pour B est créée, `initialize` par défaut ne sera plus utilisé mais surtout il est probable que `validObject` ne sera plus utilisé non plus. Probable... Mais on ne sait pas vraiment.

Incertitude fortement désagréable, n'est-ce pas ? Voilà pourquoi l'utilisation de `callNextMethod` est à proscrire, surtout que `as` et `is` permettent de faire à peu près la même chose.

## 9.8 is, as et as<-

Quand un objet hérite d'un autre, on peut avoir besoin qu'il adopte momentanément le comportement qu'aurait son père. Pour cela, il est possible de le transformer en objet de la classe de son père grâce à `as`. Par exemple, si on veut imprimer `tdPitie` en ne considérant que son aspect `Trajectoires` :

```
> print(as(tdPitie,"Trajectoires"))
```

```
 ~~~~~ Trajectoires : initiateur ~~~~~
*** Class Trajectoires, method Print ***
* Temps = numeric(0)
* Traj  =
<0 x 0 matrix>
******* Fin  Print(trajectoires) *******
```

Cela va nous servir dans la définition de `show` pour les `TrajDecoupees` :

```
> setMethod(
+     f="show",
+     signature="TrajDecoupees",
+     definition=function(object){
+         show(as(object,"Trajectoires"))
+         lapply(object@listePartitions,show)
+     }
+ )
```

```
[1] "show"
```

On peut vérifier qu'un objet est héritier d'un autre grâce à `is`. `is` vérifie que les attributs de l'objet sont présents dans la classe fils :

```
> is(trajCochin,"TrajDecoupees")
```

```
[1] FALSE
```

```
> is(tdCochin,"Trajectoires")
```

```
[1] TRUE
```

Enfin, `as<-` permet de modifier uniquement les attributs qu'un objet hérite de son père. `as(objetFils,"ClassPere")<-objetPere` affecte le contenu des attributs `objetPere` aux attributs dont `objetFils` a hérité.

```
> ### Création d'un TrajDecoupees vide
> tdStAnne <- new("TrajDecoupees")
```

```
 ~~~~ TrajDecoupees : initiateur ~~~~
```

```
> ### Affectation d'un objet Trajectoires
> ###    aux attributs d'un TrajDecoupees
> as(tdStAnne,"Trajectoires") <- trajStAnne
> tdStAnne
```

```
 ~~~~~ Trajectoires : initiateur ~~~~~
*** Class Trajectoires, method Show ***
* Temps =  [1]  1  2  3  4  5  6  7  8  9 10 12 14 16 18 20 22 24
* Traj  (limité à une matrice 10x10) =
      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9]  [,10]
 [1,] 16.25 16.1  16.16 16.53 16.67 17.2  17.55 17.3  17.47 17.55
 [2,] 16.23 16.36 16.45 16.4  17.14 17    16.84 17.25 17.98 17.43
 [3,] 15.89 16.02 16.15 16.69 16.77 16.62 17.52 17.26 17.46 17.6
 [4,] 15.63 16.25 16.4  16.6  16.65 16.96 17.02 17.39 17.5  17.67
 [5,] 16.16 15.85 15.97 16.32 16.73 16.94 16.73 16.98 17.64 17.72
 [6,] 15.96 16.2  16.53 16.4  16.47 16.95 16.73 17.36 17.33 17.55
 [7,] 16.03 16.33 16.23 16.67 16.79 17.14 17    17.35 17.58 17.99
 [8,] 15.69 16.06 16.63 16.72 16.81 17.16 16.98 17.41 17.51 17.43
 [9,] 15.82 16.17 16.75 16.76 16.78 16.51 17.19 17.21 17.84 17.95
[10,] 15.98 15.76 16.1  16.54 16.78 16.89 17.22 17.18 16.94 17.36
* ... ...
******* Fin  Show(trajectoires) *******
```

## 9.9 setIs

En cas d'héritage, `as` et `is` sont définis "naturellement", comme nous venons de le voir. Il est également possible de les préciser "manuellement". Par exemple, la classe `TrajDecoupees` contient une liste de `Partitions`. Elle n'hérite pas directement de `Partition` (ça serait un cas d'héritage multiple impropre), donc `is` et `as` ne sont pas définis par défaut.

Il est néanmoins possible de "forcer" les choses. Par exemple, on veut pouvoir considérer un objet de classe `TrajDecoupees` comme une `Partition`, celle ayant le plus grand nombre de groupes. Cela se fait avec l'instruction `setIs`. `setIs` est une méthode qui prend quatre arguments

- `from` est la classe de l'objet initial, celui qui doit être transformé.
- `to` est la classe en laquelle l'objet doit être transformé
- `coerce` est la fonction utilisée pour transformer `from` en `to`.

```
> setIs(
+     class1="TrajDecoupees",
+     class2="Partition",
+     coerce=function(from,to){
+         getNbGroupes <- function(partition){
+             return(partition["nbGroupes"])
+         }
+         nombreGroupes <-
+             sapply(from@listePartitions,getNbGroupes)
+         plusGrand <- which.max(nombreGroupes)
+         to<-new("Partition")
+         to@nbGroupes <-
+             from@listePartitions[[plusGrand]]["nbGroupes"]
+         to@part <- from@listePartitions[[plusGrand]]["part"]
+         return(to)
```

```
+     }
+ )
> as(tdCochin,"Partition")
```

```
An object of class "Partition"
Slot "nbGroupes":
[1] 3

Slot "part":
[1] A C C B
Levels: A B C
```

Ça marche pour ce dont nous avions besoin. Mais un `Warning` apparait. R nous annonce que `as<-` n'est pas défini. `as<-` est l'opérateur utilisé pour affecter une valeur à un objet *alors* qu'il est considéré comme un autre objet. Dans notre cas, `as<-` est l'opérateur utilisé pour modifier `TrajDecoupees` alors qu'elle est considérée comme une `Trajectoires`. Cela se fait grâce au quatrième argument de `setIs` :

– `replace` est la fonction utilisée pour les affectations.

Dans le cas présent, nous voudrions remplacer la partition ayant le plus grand nombre de groupes par une nouvelle :

```
> setIs(
+     class1="TrajDecoupees",
+     class2="Partition",
+     coerce=function(from,to){
+         getNbGroupes <- function(partition){
+             return(partition["nbGroupes"])
+         }
+         nombreGroupes <-
+             sapply(from@listePartitions,getNbGroupes)
+         plusGrand <- which.max(nombreGroupes)
+         to<-new("Partition")
+         to@nbGroupes <-
+             from@listePartitions[[plusGrand]]["nbGroupes"]
+         to@part <- from@listePartitions[[plusGrand]]["part"]
+         return(to)
+     },
+     replace=function(from,values){
+         getNbGroupes <- function(partition){
+             return(partition["nbGroupes"])
+         }
+         nombreGroupes <-
+             sapply(tdCochin@listePartitions,getNbGroupes)
+         plusGrand <- which.max(nombreGroupes)
+         from@listePartitions[[plusGrand]] <- values
+         return(from)
+     }
+ )
> as(tdCochin,"Partition")<-partCochin2
```

Plus de `Warning`, la vie est belle.

Éventuellement, il est possible d'ajouter un cinquième argument, la fonction `test` : elle subordonne la transformation de `class1` en `class2` à une condition.

Nous venons d'expliquer à R comment considérer un objet `TrajDecoupe` comme une `Partition`. En réalité, un tel usage serait impropre, une `TraDecoupees` n'a pas à devenir une `Partition`. Si nous voulons accéder à une partition particulière, il nous faut définir un getteur (par exemple `getPartitionMax`).

De manière plus générale, `setIs` est une fonction à éviter. R converti "naturellement" ce qui doit être converti. `setIs` ajoute des conversions artificielles au gré du programmeur, conversions qui seront peut-être "surprenantes" pour l'utilisateur. Et comme toujours, les surprises sont à proscrire...

## 9.10 Les classes virtuelles

Il peut arriver que des classes soient proches sans que l'une soit une extension de l'autre. Par exemple, nous allons concevoir deux types de partitionnement : des découpages qui "étiquettent" les individus sans les juger et ceux qui évaluent les individus. Le premier type de découpage sera non ordonné (la même chose que `Partition`) et le deuxième type de découpe sera ordonné (par exemple, il classera les individus en `Insuffisant, Moyen, Bien`). Les attributs de cette deuxième classe seront `nbGroupes`, un entier qui indique le nombre de modalités de l'évaluation et `part`, une variable ordonnée qui indiquera le groupe d'appartenance. Clairement, les attributs ne sont pas les mêmes dans les deux classes puisque `part` est ordonné dans l'une et pas dans l'autre. Donc l'une ne peut pas hériter de l'autre. Pourtant, les méthodes seront semblables dans bon nombre de cas.

Pour ne pas avoir à les programmer en double, on peut faire appel à une *classe virtuelle*. Une classe virtuelle est une classe pour laquelle il n'est pas possible de créer des objets mais qui peut avoir des héritiers. Les héritiers bénéficient ensuite des méthodes créées pour la classe. Dans notre cas, nous allons créer une classe virtuelle `PartitionPere` puis deux classes fils `PartitionSimple` et `PartitionEvaluante`. Toutes les méthodes communes seront créées pour `PartitionPere`, les deux fils en hériteront.

```
> setClass(
+     Class="PartitionPere",
+     representation=representation(nbGroupes="numeric","VIRTUAL")
+ )
```

```
[1] "PartitionPere"
```

```
> setClass(
+     Class="PartitionSimple",
+     representation=representation(part="factor"),
+     contains="PartitionPere"
+ )
```

```
[1] "PartitionSimple"
```

```
> setClass(
+     Class="PartitionEvaluante",
+     representation=representation(part="ordered"),
+     contains="PartitionPere"
+ )
```

```
[1] "PartitionEvaluante"
```

```
> setGeneric("nbMultDeux",
+     function(object){standardGeneric("nbMultDeux")}
+ )
```

```
[1] "nbMultDeux"
```

```
> setMethod("nbMultDeux","PartitionPere",
+     function(object){
+         object@nbGroupes <- object@nbGroupes*2
+         return(object)
+     }
+ )
```

```
[1] "nbMultDeux"
```

```
> a <- new("PartitionSimple",nbGroupes=3,
+     part=factor(LETTERS[c(1,2,3,2,2,1)])
+ )
> nbMultDeux(a)
```

```
An object of class "PartitionSimple"
Slot "part":
[1] A B C B B A
Levels: A B C

Slot "nbGroupes":
[1] 6
```

```
> b <- new("PartitionEvaluante",nbGroupes=5,
+     part=ordered(LETTERS[c(1,5,3,4,2,4)])
+ )
> nbMultDeux(b)
```

```
An object of class "PartitionEvaluante"
Slot "part":
[1] A E C D B D
Levels: A < B < C < D < E

Slot "nbGroupes":
[1] 10
```

Et voilà...

## 9.11 Pour les dyslexiques...

Pour conclure avec l'héritage, une petite astuce mnémotechnique destinée aux dyslexiques (dont moi) qui se mélangent assez vite les neurones et qui ne savent jamais si l'héritier peut utiliser les techniques du père ou si c'est l'inverse : l'histoire est issue de "Contes et légendes de la naissance de Rome" [3]. L'auteur explique que, *contrairement aux humains* les Dieux n'aiment pas avoir des enfants plus puissants qu'eux. C'est pour ça que Saturne dévore ses propres enfants à leur naissance (pour les émotifs, rassurez-vous, ça n'est tout de même pas trop grave pour les enfants puisque lorsque Saturne se fait finalement couper en deux par son fils Jupiter, tous sortent de son ventre et sont bien vivants !). Chez les humains, c'est l'inverse, les pères sont plutôt fiers d'avoir des enfants plus grands, qui les battent au tennis (sur le moment ils râlent, mais après ils racontent à leurs copains), avec un meilleur niveau d'étude... C'est l'ascenseur social qui fonctionne. Chez les objets, c'est pareil que chez les humains : *Les fils sont plus forts que les pères*. Un fils peut faire tout ce que fait le père, plus ce qui lui est propre [3]

3. Pour les méta-dyslexiques qui se souviendront de cette histoire mais qui ne sauront plus si les objets se comportent comme des Dieux ou des humains, on ne peut plus rien faire... [4]

4. *"Les objets sont truffés de bugs,* ajoute un relecteur, **comme les humains,** *(Alors que chacun sait que les Dieux sont parfaits !)"* Voilà pour les méta-dyslexiques.

# Chapitre 10

# Modification interne d'un objet

La suite nécessite de descendre un peu plus profondément dans les méandres du fonctionnement de R...

## 10.1 Fonctionnement interne de R : les `environnements`

Un environnement est un espace de travail permettant de stocker des variables. Pour simplifier, R travaille en utilisant deux environnements : le *global* et le *local*. Le global est celui auquel nous avons accès quand nous tapons des instructions dans la console. Le local est un environnement qui se crée à chaque appel de fonction. Puis, quand la fonction se termine, le local est détruit. Exemple :

```
> func <- function(){
+     x <- 5
+     cat(x)
+     return(invisible())
+ }
> ### Creation de x 'global'
> x <- 2
> cat(x)
```

```
2
```

```
> ### Appel de la fonction et création de x 'local'
> func()
```

```
5
```

```
> ### Retour au global, suppression de x local
> cat(x)
```

```
2
```

La fonction `func` est définie dans l'environnement global. Toujours dans le global, `x` reçoit la valeur 2. Puis la fonction `func` est appelée. R crée donc l'environnement local.

Dans le local, il donne à `x` la valeur 5. Mais uniquement dans le local. A ce stade, il existe donc *deux* `x` : un global qui vaut 2 et un local qui vaut 5.

La fonction affiche le `x` local puis se termine. L'environnement local est alors détruit. Ainsi, le `x` qui valait 5 disparait. Reste le `x` qui vaut 2, le global.

## 10.2 Méthode pour modifier un attribut

Retour à nos trajectoires. Il nous reste une troisième méthode à définir, celle qui impute les variables. Pour simplifier, nous imputerons en remplaçant par la moyenne [1].

```
> meanSansNa <- function(x){mean(x,na.rm=TRUE)}
> setGeneric("impute",function(object){standardGeneric("impute")})
```

```
[1] "impute"
```

```
> setMethod(
+     f="impute",
+     signature="Trajectoires",
+     def=function(object){
+         moyenne <- apply(object@traj,2,meanSansNa)
+         for (iCol in 1:ncol(object@traj)){
+             object@traj[is.na(object@traj[,iCol]),iCol] <-
+                 moyenne[iCol]
+         }
+         return(object)
+     }
+ )
```

```
[1] "impute"
```

```
> impute(trajCochin)
```

```
*** Class Trajectoires, method Show ***
* Temps = [1] 2 3 4 5
* Traj  (limité à une matrice 10x10) =
     [,1] [,2]  [,3] [,4]
[1,] 15   15.1  15.2 15.2
[2,] 16   15.9  16   16.4
[3,] 15.2 15.53 15.3 15.3
[4,] 15.7 15.6  15.8 16
* ... ...
******* Fin  Show(trajectoires) *******
```

La méthode `impute` fonctionne correctement. Par contre, elle ne modifie pas `trajCochin`.

```
> trajCochin
```

1. Pour des trajectoires, cette méthode n'est pas bonne, il vaudrait mieux imputer par la moyenne des valeurs encadrant la manquante. Mais le but de ce manuel est d'apprendre S4, pas de devenir des experts es-trajectoire. Nous allons donc au plus simple.

```
*** Class Trajectoires, method Show ***
* Temps = [1] 2 3 4 5
* Traj  (limité à une matrice 10x10) =
     [,1] [,2] [,3] [,4]
[1,] 15   15.1 15.2 15.2
[2,] 16   15.9 16   16.4
[3,] 15.2 NA   15.3 15.3
[4,] 15.7 15.6 15.8 16
* ... ...
******* Fin  Show(trajectoires) *******
```

A la lumière de ce que nous venons de voir sur les environnements, que fait cette méthode ? Elle crée *localement* un objet `object`, elle modifie ses trajectoires (imputation par la moyenne) puis elle retourne un objet. Le fait d'avoir tapé `impute(trajCochin)` n'a donc eu aucun effet sur `trajCochin`.

Conceptuellement, c'est un problème. Bien sûr, il est facile de le contourner, simplement en utilisant

```
> trajCochin <- impute(trajCochin)
```

Mais `impute` est une fonction qui a pour vocation de modifier l'intérieur de l'objet, pas de créer un nouvel objet et de le réaffecter. Pour lui rendre son sens premier, nous pouvons utiliser `assign`.

`assign` est un des opérateurs les plus impropres qui soit... Il permet, alors qu'on se trouve dans l'environnement local, de modifier des variables au niveau global. C'est très mal. Mais dans le cas présent, c'est justement ce que R ne nous offre pas, et qui est pourtant classique en programmation objet. Donc, nous nous permettons ici une petite entorse aux règles (ne le dites à personne, hein ?) en espérant qu'une prochaine version de R intègre la modification interne...

Nous allons donc ré-écrire `impute` en ajoutant deux petites instructions :

`deparse(substitute())` permet de connaitre le nom de la variable qui, au niveau global, a été passée comme argument à la fonction en cours d'exécution.

`assign` permet de modifier une variable de niveau supérieur. Plus précisément, il n'existe pas 'un' niveau local, mais 'des' niveaux locaux. Par exemple, une fonction *dans* une autre fonction crée un *local dans le local*, une sorte de *sous-local* ou de *local niveau 2*. L'utilisation de `assign` que nous proposons ici n'affecte donc pas le niveau global, mais le niveau local supérieur. Modifier directement le global serait encore plus impropre...

```
> essaiCarre <- function(x){
+     nomObject <- deparse(substitute(x))
+     print(nomObject)
+     assign(nomObject,x^2,envir=parent.frame())
+     return(invisible())
+ }
> a<-2
> essaiCarre(a)
```

```
[1] "a"
```

```
> a
```

```
[1] 4
```

Voilà donc `impute` nouvelle version. Pour l'utiliser, plus besoin de faire une affection. Pour les `Trajectoires`, cela donne :

```
> setMethod(
+     f="impute",
+     signature="Trajectoires",
+     def=function(object){
+         nameObject<-deparse(substitute(object))
+         moyenne <- apply(object@traj,2,meanSansNa)
+         for (iCol in 1:ncol(object@traj)){
+             object@traj[is.na(object@traj[,iCol]),iCol] <-
+                 moyenne[iCol]
+         }
+         assign(nameObject,object,envir=parent.frame())
+         return(invisible())
+     }
+ )
```

```
[1] "impute"
```

```
> impute(trajCochin)
> trajCochin
```

```
*** Class Trajectoires, method Show ***
* Temps = [1] 2 3 4 5
* Traj  (limité à une matrice 10x10) =
     [,1] [,2]  [,3] [,4]
[1,] 15   15.1  15.2 15.2
[2,] 16   15.9  16   16.4
[3,] 15.2 15.53 15.3 15.3
[4,] 15.7 15.6  15.8 16
* ... ...
******* Fin  Show(trajectoires) *******
```

Ça marche. Encore une fois, `assign` est un opérateur impropre à manipuler avec beaucoup de précautions, il peut provoquer des catastrophes parfaitement contre-intuitives. Mais pour la modification interne, il est à ma connaissance la seule solution.

# Quatrième partie

# Annexes

# Annexe A

# Remerciements

## A.1 Nous vivons une époque formidable

Quand on vit un moment historique, une révolution, une date clef, on ne s'en rend pas toujours compte. Je pense que la création d'Internet sera considérée par les historiens futurs comme une avancée majeure, quelque chose d'aussi énorme que l'invention de l'écriture ou l'imprimerie. L'écriture, c'est la conservation de l'information. L'imprimerie, c'est la diffusion de l'information à une élite, puis à tout le monde, mais avec un coût. Internet, c'est l'instantanéité, la gratuité, le partage global, les connaissances des experts mises à la disposition de tous... Les forums, c'est la fin des questions sans réponses...

Il y a quelque temps, je ne connaissais de la S4 que le nom. En quelques mois, j'ai pu acquérir des connaissances sérieuses grâce à des gens que je ne connais pas, mais qui m'ont aidé. Bien sûr, j'ai beaucoup lu. Mais dès que j'avais une question, je la posais le soir, je dormais du sommeil du juste et le lendemain matin, j'avais ma réponse. C'est gratuit, c'est de l'altruisme, c'est tout simplement merveilleux. Nous vivons une époque formidable...

## A.2 Ceux par qui ce tutorial existe...

Un grand merci donc à de nombreuses personnes dont certaines que je ne connais pas. Pierre Bady, inestimable relecteur, m'a fait des remarques très pertinentes, en particulier sur la structure générale du document qui partait un peu dans tous les sens... Martin Morgan, non seulement connait TOUT sur la S4, mais de surcroit dégaine plus vite que son ombre quand il faut répondre à une question sur r-help... Merci aussi à l'équipe du CIRAD qui anime le forum du Groupe des Utilisateurs de R. Jamais on ne se fait jeter, jamais le moindre RTFM ou GIYF [1]. C'est cool. Bruno Falissard m'a donné (par ordre décroissant d'importance) une équipe [2], son amour de R, un thème de recherche, des idées, des contacts, un bureau, un ordi méga puissant... Sans lui, je serais sans

1. *Read The Fucking Manuel* ou *Google Is Your Friend*, réponses classiques faites à ceux qui posent des questions sans avoir pris le temps de chercher par eux-mêmes.
2. La MEILLEEEEUUUURE des équipes !

doute encore en train de végéter dans une impasse. Merci à Scarabette pour ses avis, sa fraicheur et son impertinence. Merci à Cathy et Michèle, mes deux chasseuses de fautes préférées. Enfin, merci à la R Core Team pour “the free gift of R”...

# Annexe B

# Mémo

Ça y est, vous avez tout lu, tout compris, vous êtes en train de programmer vos propres objets ? Juste un petit trou de mémoire : c'est quoi déjà le nom du troisième attribut de `validObject` ?

## B.1 Création

```
### Création de la classe
.NewClass.valid <- function(object){return(TRUE)}
setClass(
    Class="NewClass",
    representation=representation(x="numeric",y="character"),
    prototype=prototype(x=1,y="A"),
    contains=c("FatherClass"),
    validity=.NewClass.valid
)
rm(.NewClass.validity)

### Création d'un objet
new(Class="NewClass")
A <- new(Class="NewClass",x=2,y="B")
A@x <- 4  #(beurk !)

### Destruction d'une classe
removeClass("NewClass")

### Constructeur
newClass <- function(){return(new(Class="NewClass"))}
```

## B.2 Validation

```
.NewClass.initialize <- function(object,value){
    if(....){STOP("initialize(NewClass) : Erreur")}else{}
    object@x <- value; object@y <- value^2;
```

```
    validObject(object)
    return(object)
}
setMethod(f="initialize",signature="NewClass",
    definition=.NewClass.initialize
)
rm(.NewClass.initialize)
```

## B.3 Accesseur

```
> ### Getteur
> setMethod(f="[",signature="NewClass",
+     definition=function(object){return(object@x)}
+ )
> ### Setteur
> setReplaceMethod(f="[",signature="NewClass",
+     def=function(object,value){object@x<-value;return(object)}
+ )
```

## B.4 Méthodes

```
> ### Rendre une méthode générique
> setGeneric(f="newFunction",
+     def=function(z,r){standardGeneric("newFunction")}
+ )
> ### Déclarer une méthode
> setMethod(f="newFunction",signature="NewClass",
+     def=function(z,r){.... ; return(....)}
+ )
> ### Pour connaitre les arguments d'une fonction
> args(NewFunction)
```

## B.5 Quelques fonctions incontournables

- `args(print)` : (x,...)
- `args(show)` : (object)
- `args(plot)` : (x,y,...)
- `args(summary)` : (object,...)
- `args(length)` : (x)

## B.6 Voir la structure des objets

- `slotNames("Trajectoires")` : donne le nom des attributs de l'objet (mais pas leur type).
- `getSlots("Trajectoires")` : donne les attributs de l'objet et leur type.

- `getClass("Trajectoires")` : donne les attributs de l'objet, leur type, les héritiers et les ancêtres.
- `getClass("Trajectoires",complete=FALSE)` : donne les attributs de l'objet, les fils et le père uniquement.
- `getMethod(f="plot",signature="Trajectoires")` : donne la définition de `plot` pour l'objet `Trajectoires` (sans héritage).
- `getMethods("plot")` : donne la définition de `plot` pour toutes les signatures possibles.
- `existsMethod(f="length",signature="Trajectoires")` : retourne vrai si la méthode existe (sans d'héritage) pour la classe en question.
- `selectMethod(f="length",signature="Trajectoires")` : donne la définition de `length` pour l'objet `Trajectoires` (avec héritage).
- `hasMethod("length","Trajectoires")` : retourne vrai si la méthode existe pour `Trajectoires` (avec héritage).
- `showMethods("plot")` : affiche toutes les signatures utilisables pour la méthode `plot`.
- `showMethods(classes="Trajectoires")` : affiche toutes les méthodes définies pour la classe `Trajectoires`
- `findMethod("plot","Trajectoires")` : donne l'environnement dans lequel `plot` pour `Trajectoires` est défini.

# Annexe C

# Pour aller plus loin

La littérature sur la S4 est assez sommaire, et pratiquement exclusivement en anglais. Voilà un petit aperçu de ce qu'on peut trouver :

- *S4 Classes in 15 pages more or less* [1] : Un bon tutorial pour commencer.
- *S4 Classes and Methods* [2] : comme précédemment, un tutorial simple pour commencer.
- *S Programming*[5] : le chapitre sur la S4 est court, mais il est assez clair.
- *A (Not so) Short Introduction to S4* : première version du livre que vous avez entre les mains.
- Le *Green Book* présente également un chapitre sur la S4, mais il est un peu obscur...

Enfin, signalons l'existence d'un wiki dédié à la programmation S4 [6]

# Liste des tables et figures

Diagramme des classes, exemple ..... 20
Trajectoires ..... 21
Diagramme des classes ..... 24
Arbre d'héritage ..... 72
Arborescence de `packClassic`, "à la main" ..... ??
Arborescence de `packClassic`, selon `pacakge.squeleton` ..... ??
Correspondance visible/invisible/publique/privée ..... ??
Arborescence de `packClassic`, version finale ..... ??
Arborescence de `packS4`, "à la main" ..... ??
Arborescence de `packS4`, selon `pacakge.squeleton` ..... ??
Liste des fichiers d'aides et alias selon `pacakge.squeleton` ..... ??
Classe et méthodes dans un unique ficher ..... ??
Classe et méthodes dans des fichers séparés ..... ??
Liste des fichiers d'aides et alias, après modifications ..... ??
Arborescence de `packS4`, version finale ..... ??

# Bibliographie


[1] R. Gentleman. S4 Classes in 15 Pages More or Less, 2002
http://www.bioconductor.org/develPage/guidelines/programming/S4Objects.pdf.

[2] F. Leisch. S4 Classes and Methods, 2004
http://www.ci.tuwien.ac.at/Conferences/useR-2004/Keynotes/Leisch.pdf.

[3] L. Orvieto. *Contes et Légendes de la naissance de Rome*. Nathan, 1968.

[4] E. Paradis. R pour les debutants, 2002
http://cran. r-project. org/doc/contrib/rdebuts_fr. pdf.

[5] W.N. Venables and B.D. Ripley. *S Programming*. Springer, 2000.

[6] Wiki. tips:classes-S4,
http://wiki.r-project.org/rwiki/doku.php?id=tips:classes-s4.

# Index


**- Symbols -**

`<-` ... 55
@ ... 31, 45, 53, 58
`[` ... 56, 58, 92
`[<-` ... 56, 92

**- A -**

accesseur ... 53, 92
affectant ... 53
affichage ... 37
`ANY` ... 68, 70
arbre d'héritage ... 72
`args` ... 36
`array` ... 33
`as` ... 77
`as<-` ... 77, 79
`assign` ... 85
`attr` ... 31
attribut ... 19, 29, 31
`attributes` ... 31
avancée majeure ... 89

**- B -**

boite à moustache ... 35

**- C -**

côté obscur 16, 17, 23, 25, 29–33, 37, 40, 44, 45, 48, 49, 57, 58, 72, 76, 80, 85, 86
`callNextMethod` ... 75, 76
`character` ... 64
`character` ... 33
CIRAD ... 89
`Class` ... 29
classe ... 19, 29
  virtuelle ... 80
`cluster` ... 43
`coerce` ... 78
constructeur ... 19, 30, 48, 50
construction ... 43, 45
`contains` ... 70
contrôle de type ... 29

**- D -**

`def` ... 40
`definition` ... 36
`deparse` ... 85
données manquantes ... 22
`drop` ... 56
dyslexique ... 82

**- E -**

encapsulation ... 15
environnement ... 85
  global ... 83
  local ... 83
`environnement` ... 83
époque formidable ... 89
erreur ... 12, 32
  initialisation ... 32
  typage ... 13
  typographique ... 57, 58
  valeur ... 14
exemple ... 21–24, 64
`existsMethod` ... 42, 73, 92

**- F -**

`f` ... 36
`factor` ... 23, 33
fichier ... 41
fils ... 14, 69, 80

`findMethod` ..... 92
fonction ..... 11, 19
`from` ..... 78

**- G -**

générique ..... 39, 41
get ..... 53, 58, 92
`getClass` ..... 34, 92
`getMethod` ..... 42, 73, 92
`getMethods` ..... 92
`getSlots` ..... 34, 92
getteur ..... 53, 92
grand-père ..... 70
graphe ..... 72

**- H -**

héritage ..... 14, 23, 69, 72, 77, 78, 80
  multiple ..... 72, 78
`hasMethod` ..... 92
`hasMethods` ..... 73

**- I -**

`i` ..... 56
IMC ..... 12, 21, 47
imputation ..... 23, 84
`impute` ..... 84–86
Indice de Masse Corporelle ..... 12
initialisation ..... 32
`initialize` ..... 45, 73, 76
initiateur ..... 45, 50
instruction surprenante ..... 17
`integer` ..... 33
`is` ..... 77

**- L -**

`length` ..... 33
`listePartitions` ..... 24, 71
`lockBinding` ..... 40

**- M -**

méthode ..... 19, 35, 64, 92
  autre ..... 20, 25
  création ..... 19, 25, 43
  générique ..... 39, 41
  manipulation d'attribut ..... 20, 25
  spécifique ..... 39
  validation ..... 20, 25, 43
`matplot` ..... 36
`matrix` ..... 33
`miniKml` ..... 3, 66
`missing` ..... 67

**- N -**

`name` ..... 40
`nbCluster` ..... 43
`nbGroupes` ..... 23, 80
`new` ..... 30, 38, 73, 91
`nextMethod` ..... 75
nuage de points ..... 35
`NULL` ..... 33
`numeric` ..... 23, 33, 64

**- O -**

objet ..... 11, 19
  définition ..... 19
  vide ..... 33, 38, 47

**- P -**

père ..... 14, 69, 70, 80
`part` ..... 23, 64, 80
`Partition` ..... 23, 63, 80
partition ..... 23
`PartitionEvaluante` ..... 80
`PartitionPere` ..... 80
`PartitionSimple` ..... 80
pause ..... 61
`plot` ..... 25, 35, 39, 64, 66
`print` ..... 25, 37
programmation objet ..... 11
  inconvénients ..... 15
  présentation ..... 13
`prototype` ..... 32, 44

**- R -**

R ..... 11
R Core Team ..... 90
règle ..... 25
racine ..... 70
remerciement ..... 89
`removeClass` ..... 32, 91

replace ..... 79
representation ..... 29, 44

**- S -**

S-plus ..... 11
S3 ..... 11
S4 ..... 11
selecteur ..... 53
selectMethod ..... 73, 92
set ..... 53, 92
setClass ..... 29, 91
setGeneric ..... 39, 40, 92
setIs ..... 78, 80
setMethod ..... 35, 64, 91, 92
setReplaceMethod ..... 55
setteur ..... 53, 55, 57, 92
setValidity ..... 44
show ..... 25, 37, 71, 77
showMethods ..... 41, 68, 92
signature ..... 68
signature ..... 36, 64
slotNames ..... 34, 92
spécifique ..... 39
Sportive ..... 70
substitute ..... 85

**- T -**

temps ..... 22, 24
test ..... 80
to ..... 78
traj ..... 22, 24
TrajDecoupees ..... 23, 69, 70, 72, 79
trajectoire ..... 21, 35
Trajectoires ... 22, 43, 64, 70, 77, 79, 86
trajectoires ..... 37
trajectoires ..... 48
type ..... 29, 40
  fonction ..... 20

**- U -**

unclass ..... 71

**- V -**

vérificateur ..... 43, 51
valeur manquante ..... 22
valeur par défaut ..... 32
validity ..... 43
validObject ..... 46, 76
variable ..... 11, 19
  nature ..... 23
  nominale ..... 23
  numérique ..... 23
  typée ..... 19
Vehicule ..... 69, 70
Voiture ..... 69, 70

**- W -**

Warning ..... 79

**- X -**

x ..... 56