

# Programmation orientée objet

Techniques avancées en programmation statistique R

---

Patrick Fournier

Automne 2019

Université du Québec à Montréal

# Programmation orientée objet

---

# Qu'est-ce que la OOP ?

~→ Paradigme dans lequel le concept d'objet joue un rôle fondamental.

# Qu'est-ce que la OOP ?

- ~> Paradigme dans lequel le concept d'objet joue un rôle fondamental.
- ~> Un objet peut représenter

# Qu'est-ce que la OOP ?

- ~> Paradigme dans lequel le concept d'objet joue un rôle fondamental.
- ~> Un objet peut représenter
  - ~> un objet physique (table, chargé de cours, ...),

# Qu'est-ce que la OOP ?

- ~> Paradigme dans lequel le concept d'objet joue un rôle fondamental.
- ~> Un objet peut représenter
  - ~> un objet physique (table, chargé de cours, ...),
  - ~> un concept (matrice, paix dans le monde, ...).

# Qu'est-ce que la OOP ?

- ~> Paradigme dans lequel le concept d'objet joue un rôle fondamental.
- ~> Un objet peut représenter
  - ~> un objet physique (table, chargé de cours, ...),
  - ~> un concept (matrice, paix dans le monde, ...).
  - ~> ...

# Qu'est-ce que la OOP ?

- ~> Paradigme dans lequel le concept d'objet joue un rôle fondamental.
- ~> Un objet peut représenter
  - ~> un objet physique (table, chargé de cours, ...),
  - ~> un concept (matrice, paix dans le monde, ...).
  - ~> ...
- ~> L'approche orientée objet consiste à



# Qu'est-ce que la OOP ?

- ~> Paradigme dans lequel le concept d'objet joue un rôle fondamental.
- ~> Un objet peut représenter
  - ~> un objet physique (table, chargé de cours, ...),
  - ~> un concept (matrice, paix dans le monde, ...).
  - ~> ...
- ~> L'approche orientée objet consiste à
  - ~> définir un ensemble d'objet et

# Qu'est-ce que la OOP ?

- ~> Paradigme dans lequel le concept d'objet joue un rôle fondamental.
- ~> Un objet peut représenter
  - ~> un objet physique (table, chargé de cours, ...),
  - ~> un concept (matrice, paix dans le monde, ...).
  - ~> ...
- ~> L'approche orientée objet consiste à
  - ~> définir un ensemble d'objet et
  - ~> définir les interactions possibles entre ces objets.

# Qu'est-ce que la OOP ?

- ~> Paradigme dans lequel le concept d'objet joue un rôle fondamental.
- ~> Un objet peut représenter
  - ~> un objet physique (table, chargé de cours, ...),
  - ~> un concept (matrice, paix dans le monde, ...).
  - ~> ...
- ~> L'approche orientée objet consiste à
  - ~> définir un ensemble d'objet et
  - ~> définir les interactions possibles entre ces objets.
- ~> OOP possible dans la plupart des langages.

# Qu'est-ce que la OOP ?

- ~> Paradigme dans lequel le concept d'objet joue un rôle fondamental.
- ~> Un objet peut représenter
  - ~> un objet physique (table, chargé de cours, ...),
  - ~> un concept (matrice, paix dans le monde, ...).
  - ~> ...
- ~> L'approche orientée objet consiste à
  - ~> définir un ensemble d'objet et
  - ~> définir les interactions possibles entre ces objets.
- ~> OOP possible dans la plupart des langages.
- ~> Toutefois, facilitée par l'utilisation d'un langage approprié.

# Qu'est-ce que la OOP ?

- ~> Paradigme dans lequel le concept d'objet joue un rôle fondamental.
- ~> Un objet peut représenter
  - ~> un objet physique (table, chargé de cours, ...),
  - ~> un concept (matrice, paix dans le monde, ...).
  - ~> ...
- ~> L'approche orientée objet consiste à
  - ~> définir un ensemble d'objet et
  - ~> définir les interactions possibles entre ces objets.
- ~> OOP possible dans la plupart des langages.
- ~> Toutefois, facilitée par l'utilisation d'un langage approprié.
- ~> Premier langage OO : *Simula (1962)*, destiné aux simulation Monte Carlo.

## Classe

Déclaration (parfois définition) de la structure interne d'un ensemble d'objets.

## Classe

Déclaration (parfois définition) de la structure interne d'un ensemble d'objets.

## Slot/field/attribut

Variable appartenant à un objet/une classe.

## **Classe**

Déclaration (parfois définition) de la structure interne d'un ensemble d'objets.

## **Slot/field/attribut**

Variable appartenant à un objet/une classe.

## **Méthode**

Fonction associée à un objet/une classe.



## **Classe**

Déclaration (parfois définition) de la structure interne d'un ensemble d'objets.

## **Slot/field/attribut**

Variable appartenant à un objet/une classe.

## **Méthode**

Fonction associée à un objet/une classe.

## **Héritage**

Processus par lequel une classe fille acquiert la structure d'une classe mère.

## **Classe**

Déclaration (parfois définition) de la structure interne d'un ensemble d'objets.

## **Slot/field/attribut**

Variable appartenant à un objet/une classe.

## **Méthode**

Fonction associée à un objet/une classe.

## **Héritage**

Processus par lequel une classe fille acquiert la structure d'une classe mère.

## **Polymorphisme**

Une interface pour plusieurs entités de type différents [1].

# Terminologie

## Classe

Déclaration (parfois définition) de la structure interne d'un ensemble d'objets.

## Slot/field/attribut

Variable appartenant à un objet/une classe.

## Méthode

Fonction associée à un objet/une classe.

## Héritage

Processus par lequel une classe fille acquiert la structure d'une classe mère.

## Polymorphisme

Une interface pour plusieurs entités de type différents [1].

## Encapsulation

Cacher implémentation à l'utilisateur.

## Message passing

Les objets communiquent entre eux à l'aide de messages.

## Message passing

Les objets communiquent entre eux à l'aide de messages.

↪ Message = *idée fondamentale* de l'OOP originale (Alan Kay et Smalltalk).

## Message passing

Les objets communiquent entre eux à l'aide de messages.

- ~> Message = *idée fondamentale* de l'OOP originale (Alan Kay et Smalltalk).
- ~> Les messages sont souvent limités à des appels de méthodes (C++, Java, ...)

## Message passing

Les objets communiquent entre eux à l'aide de messages.

- ~> Message = *idée fondamentale* de l'OOP originale (Alan Kay et Smalltalk).
- ~> Les messages sont souvent limités à des appels de méthodes (C++, Java, ...)
- ~> Toutefois, le concept est beaucoup plus large (Smalltalk, Groovy, ...)

## Message passing

Les objets communiquent entre eux à l'aide de messages.

- ~> Message = *idée fondamentale* de l'OOP originale (Alan Kay et Smalltalk).
- ~> Les messages sont souvent limités à des appels de méthodes (C++, Java, ...)
- ~> Toutefois, le concept est beaucoup plus large (Smalltalk, Groovy, ...)

## Generic function

Les appels de méthodes sont dispatchés par des fonction spécifiques, les *fonctions génériques*.



# Typologie de la OOP

## Message passing

Les objets communiquent entre eux à l'aide de messages.

- ~> Message = *idée fondamentale* de l'OOP originale (Alan Kay et Smalltalk).
- ~> Les messages sont souvent limités à des appels de méthodes (C++, Java, ...)
- ~> Toutefois, le concept est beaucoup plus large (Smalltalk, Groovy, ...)

## Generic function

Les appels de méthodes sont dispatchés par des fonction spécifiques, les *fonctions génériques*.

- ~> Trouve ses origines dans Flavors (Lisp Machine Lisp) et CommonLoops (Common Lisp).

# Typologie de la OOP

## Message passing

Les objets communiquent entre eux à l'aide de messages.

- ↪ Message = *idée fondamentale* de l'OOP originale (Alan Kay et Smalltalk).
- ↪ Les messages sont souvent limités à des appels de méthodes (C++, Java, ...)
- ↪ Toutefois, le concept est beaucoup plus large (Smalltalk, Groovy, ...)

## Generic function

Les appels de méthodes sont dispatchés par des fonction spécifiques, les *fonctions génériques*.

- ↪ Trouve ses origines dans Flavors (Lisp Machine Lisp) et CommonLoops (Common Lisp).
- ↪ Approche “de base” de R.

# OOP en R

---

R est distribué avec 3 systèmes d'objets distincts.

S3

# Systèmes d'objets

R est distribué avec 3 systèmes d'objets distincts.

**S3**

↪ Fonction génériques.

# Systèmes d'objets

R est distribué avec 3 systèmes d'objets distincts.

## S3

- ↪ Fonction génériques.
- ↪ Pas (vraiment) de classes.

# Systèmes d'objets

R est distribué avec 3 systèmes d'objets distincts.

## S3

- ↪ Fonction génériques.
- ↪ Pas (vraiment) de classes.

## S4

# Systèmes d'objets

R est distribué avec 3 systèmes d'objets distincts.

## S3

- ↪ Fonction génériques.
- ↪ Pas (vraiment) de classes.

## S4

- ↪ S3 + classes formelles.



# Systèmes d'objets

R est distribué avec 3 systèmes d'objets distincts.

## S3

- ↪ Fonction génériques.
- ↪ Pas (vraiment) de classes.

## S4

- ↪ S3 + classes formelles.

Reference classes (RC, R5)

# Systèmes d'objets

R est distribué avec 3 systèmes d'objets distincts.

## S3

- ~> Fonction génériques.
- ~> Pas (vraiment) de classes.

## S4

- ~> S3 + classes formelles.

## Reference classes (RC, R5)

- ~> Message passing.

# Systèmes d'objets

R est distribué avec 3 systèmes d'objets distincts.

## S3

- ↪ Fonction génériques.
- ↪ Pas (vraiment) de classes.

## S4

- ↪ S3 + classes formelles.

## Reference classes (RC, R5)

- ↪ Message passing.
- ↪ Instances mutables.

# Systèmes d'objets

R est distribué avec 3 systèmes d'objets distincts.

## S3

- ↪ Fonction génériques.
- ↪ Pas (vraiment) de classes.

## S4

- ↪ S3 + classes formelles.

## Reference classes (RC, R5)

- ↪ Message passing.
- ↪ Instances mutables.

D'autres possibilités, notamment

# Systèmes d'objets

R est distribué avec 3 systèmes d'objets distincts.

## S3

- ↪ Fonction génériques.
- ↪ Pas (vraiment) de classes.

## S4

- ↪ S3 + classes formelles.

## Reference classes (RC, R5)

- ↪ Message passing.
- ↪ Instances mutables.

D'autre possibilités, notamment

## R6

# Systèmes d'objets

R est distribué avec 3 systèmes d'objets distincts.

## S3

- ↪ Fonction génériques.
- ↪ Pas (vraiment) de classes.

## S4

- ↪ S3 + classes formelles.

## Reference classes (RC, R5)

- ↪ Message passing.
- ↪ Instances mutables.

D'autre possibilités, notamment

## R6

- ↪ Version améliorée de R5.

# Systèmes d'objets

R est distribué avec 3 systèmes d'objets distincts.

## S3

- ↪ Fonction génériques.
- ↪ Pas (vraiment) de classes.

## S4

- ↪ S3 + classes formelles.

## Reference classes (RC, R5)

- ↪ Message passing.
- ↪ Instances mutables.

D'autre possibilités, notamment

## R6

- ↪ Version améliorée de R5.

## Closure

## Quoi choisir?

S3



## Quoi choisir?

S3

↪ Très informel.

# Quoi choisir?

S3

↪ Très informel.

↪ Peu de fonctionnalités.

# Quoi choisir?

S3

- ~> Très informel.
- ~> Peu de fonctionnalités.
- ~> Relativement facile à apprendre.

# Quoi choisir?

## S3

- ↪ Très informel.
- ↪ Peu de fonctionnalités.
- ↪ Relativement facile à apprendre.
- ↪ Très facile à utiliser.

## S3

- ↪ Très informel.
- ↪ Peu de fonctionnalités.
- ↪ Relativement facile à apprendre.
- ↪ Très facile à utiliser.
- ↪ Suffisant pour un très grand nombre d'utilisations.

# Quoi choisir?

## S3

- ↪ Très informel.
- ↪ Peu de fonctionnalités.
- ↪ Relativement facile à apprendre.
- ↪ Très facile à utiliser.
- ↪ Suffisant pour un très grand nombre d'utilisations.

## S4

# Quoi choisir?

## S3

- ↪ Très informel.
- ↪ Peu de fonctionnalités.
- ↪ Relativement facile à apprendre.
- ↪ Très facile à utiliser.
- ↪ Suffisant pour un très grand nombre d'utilisations.

## S4

- ↪ Moins performant.

# Quoi choisir?

## S3

- ↪ Très informel.
- ↪ Peu de fonctionnalités.
- ↪ Relativement facile à apprendre.
- ↪ Très facile à utiliser.
- ↪ Suffisant pour un très grand nombre d'utilisations.

## S4

- ↪ Moins performant.
- ↪ Difficile à utiliser.



# Quoi choisir?

## S3

- ↪ Très informel.
- ↪ Peu de fonctionnalités.
- ↪ Relativement facile à apprendre.
- ↪ Très facile à utiliser.
- ↪ Suffisant pour un très grand nombre d'utilisations.

## S4

- ↪ Moins performant.
- ↪ Difficile à utiliser.

## R5

# Quoi choisir?

## S3

- ↪ Très informel.
- ↪ Peu de fonctionnalités.
- ↪ Relativement facile à apprendre.
- ↪ Très facile à utiliser.
- ↪ Suffisant pour un très grand nombre d'utilisations.

## S4

- ↪ Moins performant.
- ↪ Difficile à utiliser.

## R5

- ↪ La mutabilité brise le comportement attendu en R.

# Quoi choisir?

## S3

- ↪ Très informel.
- ↪ Peu de fonctionnalités.
- ↪ Relativement facile à apprendre.
- ↪ Très facile à utiliser.
- ↪ Suffisant pour un très grand nombre d'utilisations.

## S4

- ↪ Moins performant.
- ↪ Difficile à utiliser.

## R5

- ↪ La mutabilité brise le comportement attendu en R.

## R6

# Quoi choisir?

## S3

- ↪ Très informel.
- ↪ Peu de fonctionnalités.
- ↪ Relativement facile à apprendre.
- ↪ Très facile à utiliser.
- ↪ Suffisant pour un très grand nombre d'utilisations.

## S4

- ↪ Moins performant.
- ↪ Difficile à utiliser.

## R5

- ↪ La mutabilité brise le comportement attendu en R.

## R6

- ↪ Semblable à R5.

# Quoi choisir?

## S3

- ↪ Très informel.
- ↪ Peu de fonctionnalités.
- ↪ Relativement facile à apprendre.
- ↪ Très facile à utiliser.
- ↪ Suffisant pour un très grand nombre d'utilisations.

## S4

- ↪ Moins performant.
- ↪ Difficile à utiliser.

## R5

- ↪ La mutabilité brise le comportement attendu en R.

## R6

- ↪ Semblable à R5.
- ↪ Malgré cela, un des paquet les plus téléchargé de CRAN (2017).

## Quoi choisir?

Pas de choix universel, mais

~→ Par défaut, choisir S3.

# Quoi choisir?

Pas de choix universel, mais

- ~> Par défaut, choisir S3.
- ~> Pour implémenter une structure de donnée mutable, choisir R6.

# Quoi choisir?

Pas de choix universel, mais

- ~> Par défaut, choisir S3.
- ~> Pour implémenter une structure de donnée mutable, choisir R6.
- ~> Il n'y a pas vraiment de bonne raison d'utiliser R5.



# Quoi choisir?

Pas de choix universel, mais

- ~> Par défaut, choisir S3.
- ~> Pour implémenter une structure de donnée mutable, choisir R6.
- ~> Il n'y a pas vraiment de bonne raison d'utiliser R5.
- ~> Utilisez S4 si vous avez besoin

# Quoi choisir?

Pas de choix universel, mais

- ~> Par défaut, choisir S3.
- ~> Pour implémenter une structure de donnée mutable, choisir R6.
- ~> Il n'y a pas vraiment de bonne raison d'utiliser R5.
- ~> Utilisez S4 si vous avez besoin
  - ~> de l'héritage multiple (plus d'une classe mère) ou

# Quoi choisir?

Pas de choix universel, mais

- ~> Par défaut, choisir S3.
- ~> Pour implémenter une structure de donnée mutable, choisir R6.
- ~> Il n'y a pas vraiment de bonne raison d'utiliser R5.
- ~> Utilisez S4 si vous avez besoin
  - ~> de l'héritage multiple (plus d'une classe mère) ou
  - ~> de dispatch multiple (dispatch sur plusieurs arguments).

# Quoi choisir?

Pas de choix universel, mais

- ~> Par défaut, choisir S3.
- ~> Pour implémenter une structure de donnée mutable, choisir R6.
- ~> Il n'y a pas vraiment de bonne raison d'utiliser R5.
- ~> Utilisez S4 si vous avez besoin
  - ~> de l'héritage multiple (plus d'une classe mère) ou
  - ~> de dispatch multiple (dispatch sur plusieurs arguments).

Il ne faut pas sous-estimer S3. En général, il est amplement suffisant.

S3

---

↪ S3 est basé sur un attribut : `class`.

# Attribut `class`

- ~> S3 est basé sur un attribut : `class`.
- ~> Cet attribut prend la forme d'un vecteur de chaînes de caractères.

# Attribut `class`

- ~> S3 est basé sur un attribut : `class`.
- ~> Cet attribut prend la forme d'un vecteur de chaînes de caractères.
- ~> Les entrées sont ordonnées de la classe la plus spécifique à la moins spécifique.



# Attribut class

- ~> S3 est basé sur un attribut : `class`.
- ~> Cet attribut prend la forme d'un vecteur de chaînes de caractères.
- ~> Les entrées sont ordonnées de la classe la plus spécifique à la moins spécifique.

```
1 > class(lm(dist ~ speed, cars))
2 [1] "lm"
3 > class(glm(dist ~ speed, cars, family = "gaussian"))
4 [1] "glm" "lm"
```

## Structure d'un objet S3

$\rightsquigarrow$  Un objet S3 est composé

## Structure d'un objet S3

- ↪ Un objet S3 est composé
  - ↪ d'un objet “standard” et

## Structure d'un objet S3

- ~> Un objet S3 est composé
  - ~> d'un objet "standard" et
  - ~> d'un attribut `class`.

## Structure d'un objet S3

- ~> Un objet S3 est composé
  - ~> d'un objet "standard" et
  - ~> d'un attribut `class`.
- ~> L'objet standard est souvent une `list`, mais il n'y a aucune obligation.

# Structure d'un objet S3

- ~> Un objet S3 est composé
  - ~> d'un objet "standard" et
  - ~> d'un attribut `class`.
- ~> L'objet standard est souvent une `list`, mais il n'y a aucune obligation.

```
1  p1 <- c(1, 2)
2  class(p1) <- c("point", "numeric")
3
4  p2 <- structure(c(3, 4), class = c("point", "numeric"))
```

# Structure d'un objet S3

- ↪ Un objet S3 est composé
  - ↪ d'un objet "standard" et
  - ↪ d'un attribut `class`.
- ↪ L'objet standard est souvent une `list`, mais il n'y a aucune obligation.

```
1 p1 <- c(1, 2)
2 class(p1) <- c("point", "numeric")
3
4 p2 <- structure(c(3, 4), class = c("point", "numeric"))
```

- ↪ Il est possible de changer la classe d'un objet.

# Structure d'un objet S3

- ↪ Un objet S3 est composé
  - ↪ d'un objet "standard" et
  - ↪ d'un attribut `class`.
- ↪ L'objet standard est souvent une `list`, mais il n'y a aucune obligation.

```
1 p1 <- c(1, 2)
2 class(p1) <- c("point", "numeric")
3
4 p2 <- structure(c(3, 4), class = c("point", "numeric"))
```

- ↪ Il est possible de changer la classe d'un objet.
- ↪ Toutefois, cela est déconseillé.



# Structure d'un objet S3

- ↪ Un objet S3 est composé
  - ↪ d'un objet "standard" et
  - ↪ d'un attribut `class`.
- ↪ L'objet standard est souvent une `list`, mais il n'y a aucune obligation.

```
1 p1 <- c(1, 2)
2 class(p1) <- c("point", "numeric")
3
4 p2 <- structure(c(3, 4), class = c("point", "numeric"))
```

- ↪ Il est possible de changer la classe d'un objet.
- ↪ Toutefois, cela est déconseillé.
- ↪ Particulièrement vrai si l'objet a été créé par quelqu'un d'autre.

~→ L'appel à `class<-` ou `structure` n'est pas très élégant, pratique ni sécuritaire.

- ~> L'appel à `class<-` ou `structure` n'est pas très élégant, pratique ni sécuritaire.
- ~> Pour cette raison, il est d'usage de définir un ou plusieurs *constructeurs*.

- ~> L'appel à `class<-` ou `structure` n'est pas très élégant, pratique ni sécuritaire.
- ~> Pour cette raison, il est d'usage de définir un ou plusieurs *constructeurs*.
- ~> Il s'agit d'une simple fonction qui se charge de faire cet appel.

- ~> L'appel à `class<-` ou `structure` n'est pas très élégant, pratique ni sécuritaire.
- ~> Pour cette raison, il est d'usage de définir un ou plusieurs *constructeurs*.
- ~> Il s'agit d'une simple fonction qui se charge de faire cet appel.
- ~> Habituellement, les constructeurs ont le même nom que la classe.

- ~> L'appel à `class<-` ou `structure` n'est pas très élégant, pratique ni sécuritaire.
- ~> Pour cette raison, il est d'usage de définir un ou plusieurs *constructeurs*.
- ~> Il s'agit d'une simple fonction qui se charge de faire cet appel.
- ~> Habituellement, les constructeurs ont le même nom que la classe.
- ~> On peut en profiter pour valider les valeurs fournies par l'utilisateur.

# Constructeur

```
1 point <- function(v){
2   identical(length(v), as.integer(2)) ||
3     stop("v doit être de longueur 2")
4   is.numeric(v) ||
5     stop("v doit être numeric")
6
7   structure(as.numeric(v), class = c("point", "numeric"))
8 }
```

```
1 > point(1:3)
2 Error in point(1:3) : v doit être de longueur 2.
3
4 Enter a frame number, or 0 to exit
5
6 1: point(1:3)
7 > point(letters[1:2])
8 Error in point(letters[1:2]) : v doit être numeric.
9
10 Enter a frame number, or 0 to exit
11 > point(1:2)
12 [1] 1 2
13 attr(,"class")
14 [1] "point" "numeric"
```

~→ On peut reconnaître les méthodes par leur nom :

`méthode.classe(...)`



~> On peut reconnaître les méthodes par leur nom :

`méthode.classe(...)`

~> Il est donc facile de définir de nouvelles méthodes : il suffit de définir une fonction nommée de manière appropriée.

~> On peut reconnaître les méthodes par leur nom :

`méthode.classe(...)`

~> Il est donc facile de définir de nouvelles méthodes : il suffit de définir une fonction nommée de manière appropriée.

~> Il est aussi possible d'implémenter des méthodes pour une classe que l'on n'a pas définie.

~> On peut reconnaître les méthodes par leur nom :

`méthode.classe(...)`

~> Il est donc facile de définir de nouvelles méthodes : il suffit de définir une fonction nommée de manière appropriée.

~> Il est aussi possible d'implémenter des méthodes pour une classe que l'on n'a pas définie.

~> Cette pratique est connue sous le nom de *monkey patching* ou *type piracy*.

~> On peut reconnaître les méthodes par leur nom :

`méthode.classe(...)`

~> Il est donc facile de définir de nouvelles méthodes : il suffit de définir une fonction nommée de manière appropriée.

~> Il est aussi possible d'implémenter des méthodes pour une classe que l'on n'a pas définie.

~> Cette pratique est connue sous le nom de *monkey patching* ou *type piracy*.

~> À utiliser avec parcimonie!

~→ Lorsque le nom d'une variable est saisi, R appelle `print` sur cette variable.

# Méthodes

- ~> Lorsque le nom d'une variable est saisi, R appelle `print` sur cette variable.
- ~> Définissons une méthode `print` pour notre type `point`.

# Méthodes

- ~> Lorsque le nom d'une variable est saisi, R appelle **print** sur cette variable.
- ~> Définissons une méthode **print** pour notre type **point**.
- ~> Du même coup, définissons **summary** comme une version plus détaillée de **print**.

# Méthodes

- ↪ Lorsque le nom d'une variable est saisi, R appelle **print** sur cette variable.
- ↪ Définissons une méthode **print** pour notre type **point**.
- ↪ Du même coup, définissons **summary** comme une version plus détaillée de **print**.

```
1 print.point <- function(x) cat("x =", x[1], "& y =", x[2], "\n")
2
3 summary.point <- function(object){
4     cat("Point de coordonnées ")
5     print(object)
6 }
```

```
1 > p1
2 x = 1 & y = 2
3 > summary(p1)
4 Point de coordonnées x = 1 & y = 2
```



# Method dispatch

```
1 > summary(cars)
2      speed      dist
3  Min.   : 4.0   Min.   : 2
4  1st Qu.:12.0   1st Qu.: 26
5  Median :15.0   Median : 36
6  Mean   :15.4   Mean   : 43
7  3rd Qu.:19.0   3rd Qu.: 56
8  Max.   :25.0   Max.   :120
9 > summary(lm(dist ~ speed, cars))
10 Call:
11 lm(formula = dist ~ speed, data = cars)
12
13 Residuals:
14      Min       1Q   Median       3Q      Max
15 -29.07  -9.53  -2.27   9.21  43.20
16
17 Coefficients:
18             Estimate Std. Error t value Pr(>|t|)
19 (Intercept)  -17.579      6.758   -2.60   0.012 *
20 speed         3.932      0.416    9.46 1.5e-12 ***
21 ---
22 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

## Method dispatch

~→ Les deux expressions sont des appels à la même fonction : `summary`.

## Method dispatch

- ~> Les deux expressions sont des appels à la même fonction : `summary`.
- ~> Ce qui change est l'*argument* fourni à `summary`.

## Method dispatch

- ~> Les deux expressions sont des appels à la même fonction : **summary**.
- ~> Ce qui change est l'*argument* fourni à **summary**.
- ~> Plus précisément, le *type* de l'argument change.

# Method dispatch

- ~> Les deux expressions sont des appels à la même fonction : `summary`.
- ~> Ce qui change est l'*argument* fourni à `summary`.
- ~> Plus précisément, le *type* de l'argument change.

```
1 > class(cars)
2 [1] "data.frame"
3 > class(lm(dist ~ speed, cars))
4 [1] "lm"
```

# Method dispatch

- ~> Les deux expressions sont des appels à la même fonction : `summary`.
- ~> Ce qui change est l'*argument* fourni à `summary`.
- ~> Plus précisément, le *type* de l'argument change.

```
1 > class(cars)
2 [1] "data.frame"
3 > class(lm(dist ~ speed, cars))
4 [1] "lm"
```

Il s'agit d'un exemple de *polymorphisme*.

# Method dispatch

- ~> Les deux expressions sont des appels à la même fonction : `summary`.
- ~> Ce qui change est l'*argument* fourni à `summary`.
- ~> Plus précisément, le *type* de l'argument change.

```
1 > class(cars)
2 [1] "data.frame"
3 > class(lm(dist ~ speed, cars))
4 [1] "lm"
```

Il s'agit d'un exemple de *polymorphisme*.

- ~> Interface commune : `summary`

# Method dispatch

- ↪ Les deux expressions sont des appels à la même fonction : **summary**.
- ↪ Ce qui change est l'*argument* fourni à **summary**.
- ↪ Plus précisément, le *type* de l'argument change.

```
1 > class(cars)
2 [1] "data.frame"
3 > class(lm(dist ~ speed, cars))
4 [1] "lm"
```

Il s'agit d'un exemple de *polymorphisme*.

- ↪ Interface commune : **summary**
- ↪ Entités différentes : Apparent par les comportements différents.



~> Le fonctionnement de la fonction `summary` est très complexe.

- ~> Le fonctionnement de la fonction `summary` est très complexe.
- ~> En conséquence, son code doit être remarquablement compliqué.

# Method dispatch

- ~> Le fonctionnement de la fonction `summary` est très complexe.
- ~> En conséquence, son code doit être remarquablement compliqué.

```
1 > body(summary)
2 UseMethod("summary")
```

# Method dispatch

- ~> Le fonctionnement de la fonction `summary` est très complexe.
- ~> En conséquence, son code doit être remarquablement compliqué.

```
1 > body(summary)
2 UseMethod("summary")
```

???

## Method dispatch

~→ En fait, `summary` ne calcule aucune statistique sommaire et n'affiche rien directement.

## Method dispatch

- ~> En fait, **summary** ne calcule aucune statistique sommaire et n'affiche rien directement.
- ~> Son unique rôle est de déterminer la méthode appropriée à appeler.

## Method dispatch

- ~> En fait, **summary** ne calcule aucune statistique sommaire et n'affiche rien directement.
- ~> Son unique rôle est de déterminer la méthode appropriée à appeler.
- ~> Une telle fonction est appelée *fonction générique*.

## Method dispatch

- ~> En fait, **summary** ne calcule aucune statistique sommaire et n'affiche rien directement.
- ~> Son unique rôle est de déterminer la méthode appropriée à appeler.
- ~> Une telle fonction est appelée *fonction générique*.
- ~> En R, les fonction génériques se reconnaissent facilement par leur appel à **UseMethod**.



## Method dispatch

- ~> En fait, **summary** ne calcule aucune statistique sommaire et n'affiche rien directement.
- ~> Son unique rôle est de déterminer la méthode appropriée à appeler.
- ~> Une telle fonction est appelée *fonction générique*.
- ~> En R, les fonction génériques se reconnaissent facilement par leur appel à **UseMethod**.
- ~> **UseMethod** trouve la méthode la plus spécialisée pouvant s'appliquer à un objet d'une certaine classe.

## Method dispatch

- ~> En fait, **summary** ne calcule aucune statistique sommaire et n'affiche rien directement.
- ~> Son unique rôle est de déterminer la méthode appropriée à appeler.
- ~> Une telle fonction est appelée *fonction générique*.
- ~> En R, les fonction génériques se reconnaissent facilement par leur appel à **UseMethod**.
- ~> **UseMethod** trouve la méthode la plus spécialisée pouvant s'appliquer à un objet d'une certaine classe.
- ~> Dans notre exemple, initialement, **summary** cherchait une méthode pour la classe la plus spécifique de **p1**, **point**.

## Method dispatch

- ~> En fait, **summary** ne calcule aucune statistique sommaire et n'affiche rien directement.
- ~> Son unique rôle est de déterminer la méthode appropriée à appeler.
- ~> Une telle fonction est appelée *fonction générique*.
- ~> En R, les fonction génériques se reconnaissent facilement par leur appel à **UseMethod**.
- ~> **UseMethod** trouve la méthode la plus spécialisée pouvant s'appliquer à un objet d'une certaine classe.
- ~> Dans notre exemple, initialement, **summary** cherchait une méthode pour la classe la plus spécifique de **p1**, **point**.
- ~> Comme une telle méthode n'existait pas, il recommence la recherche pour la classe **numeric**.

# Method dispatch

```
1 > summary.point
2 Error: object 'summary.point' not found
3 No suitable frames for recover()
4 > summary.numeric
5 Error: object 'summary.numeric' not found
6 No suitable frames for recover()
```

# Method dispatch

```
1 > summary.point
2 Error: object 'summary.point' not found
3 No suitable frames for recover()
4 > summary.numeric
5 Error: object 'summary.numeric' not found
6 No suitable frames for recover()
```

~> Arrivé à ce point, R a épuisé toutes les classes auxquelles appartient **p1**.

# Method dispatch

```
1 > summary.point
2 Error: object 'summary.point' not found
3 No suitable frames for recover()
4 > summary.numeric
5 Error: object 'summary.numeric' not found
6 No suitable frames for recover()
```

~> Arrivé à ce point, R a épuisé toutes les classes auxquelles appartient **p1**.

~> Il lui reste un dernier atout : `summary.default`.

# Method dispatch

```
1 > summary.point
2 Error: object 'summary.point' not found
3 No suitable frames for recover()
4 > summary.numeric
5 Error: object 'summary.numeric' not found
6 No suitable frames for recover()
```

- ~> Arrivé à ce point, R a épuisé toutes les classes auxquelles appartient **p1**.
- ~> Il lui reste un dernier atout : **summary.default**.
- ~> En général, avant d'abandonner, R cherche une méthode **default**.

# Method dispatch

```
1 > summary.point
2 Error: object 'summary.point' not found
3 No suitable frames for recover()
4 > summary.numeric
5 Error: object 'summary.numeric' not found
6 No suitable frames for recover()
```

- ~> Arrivé à ce point, R a épuisé toutes les classes auxquelles appartient **p1**.
- ~> Il lui reste un dernier atout : **summary.default**.
- ~> En général, avant d'abandonner, R cherche une méthode **default**.
- ~> Peut être utile pour définir un comportement par défaut.



# Method dispatch

```
1 > summary.point
2 Error: object 'summary.point' not found
3 No suitable frames for recover()
4 > summary.numeric
5 Error: object 'summary.numeric' not found
6 No suitable frames for recover()
```

↪ Arrivé à ce point, R a épuisé toutes les classes auxquelles appartient **p1**.

↪ Il lui reste un dernier atout : **summary.default**.

↪ En général, avant d'abandonner, R cherche une méthode **default**.

↪ Peut être utile pour définir un comportement par défaut.

```
1 > summary.default(p1)
2   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
3   1.00   1.25   1.50   1.50   1.75   2.00
```

~→ Une méthode sans fonction générique correspondante n'est pas très utile.

# Fonction générique

- ~> Une méthode sans fonction générique correspondante n'est pas très utile.
- ~> Par conséquent, il faut régulièrement définir nos propres génériques.

# Fonction générique

- ~> Une méthode sans fonction générique correspondante n'est pas très utile.
- ~> Par conséquent, il faut régulièrement définir nos propres génériques.
- ~> La plupart du temps, ces fonctions se limitent à un appel à **UseMethod** qui prend 2 arguments :

# Fonction générique

- ~> Une méthode sans fonction générique correspondante n'est pas très utile.
- ~> Par conséquent, il faut régulièrement définir nos propres génériques.
- ~> La plupart du temps, ces fonctions se limitent à un appel à **UseMethod** qui prend 2 arguments :

**generic**

Nom de la méthode à appeler.

# Fonction générique

- ~> Une méthode sans fonction générique correspondante n'est pas très utile.
- ~> Par conséquent, il faut régulièrement définir nos propres génériques.
- ~> La plupart du temps, ces fonctions se limitent à un appel à **UseMethod** qui prend 2 arguments :

## **generic**

Nom de la méthode à appeler.

## **object**

Objet dont le *type* est utilisé pour le dispatch. Par défaut, premier argument de la fonction générique.

# Fonction générique

Définissons une méthode permettant de déterminer si des points sont alignés.

```
1  isAligned.point <- function(...){
2    points <- list(...)
3    isTRUE(length(points) <= 2) && return(TRUE)
4
5    fit <- crossprod(
6      solve(cbind(1, c(points[[1]][1], points[[2]][1])),
7        c(points[[1]][2], points[[2]][2]))
8
9    for (p in points[-(1:2)]){
10      pred <- crossprod(c(1, p[1]), fit)[1]
11
12      isTRUE(all.equal(pred, p[2])) || return(FALSE)
13    }
14
15    TRUE
16  }
17
```

# Fonction générique

```
1 > isAligned.point(point(1:2), point(3:4), point(5:6))
2 [1] TRUE
3 > isAligned.point(point(1:2), point(3:4), point(c(5, 7)))
4 [1] FALSE
```

~> À présent, définissons une générique.



# Fonction générique

```
1 > isAligned.point(point(1:2), point(3:4), point(5:6))
2 [1] TRUE
3 > isAligned.point(point(1:2), point(3:4), point(c(5, 7)))
4 [1] FALSE
```

~> À présent, définissons une générique.

```
1 isAligned <- function(...) UseMethod("isAligned")
```

# Fonction générique

```
1 > isAligned.point(point(1:2), point(3:4), point(5:6))
2 [1] TRUE
3 > isAligned.point(point(1:2), point(3:4), point(c(5, 7)))
4 [1] FALSE
```

↪ À présent, définissons une générique.

```
1 isAligned <- function(...) UseMethod("isAligned")

1 > isAligned(point(1:2), point(3:4), point(5:6))
2 [1] TRUE
3 > isAligned(point(1:2), point(3:4), point(c(5, 7)))
4 [1] FALSE
```

## Exemple

↪ Notre classe `point` n'est guère plus qu'une spécialisation de la classe `numeric`.

## Exemple

- ~> Notre classe point n'est guère plus qu'une spécialisation de la classe `numeric`.
- ~> Rien ne nous empêche de la complexifier!

## Exemple

- ~> Notre classe point n'est guère plus qu'une spécialisation de la classe `numeric`.
- ~> Rien ne nous empêche de la complexifier!
- ~> Nous allons donner la possibilité à l'utilisateur de marquer son point à l'aide d'une chaîne de caractères.

## Exemple

- ↪ Notre classe point n'est guère plus qu'une spécialisation de la classe `numeric`.
- ↪ Rien ne nous empêche de la complexifier!
- ↪ Nous allons donner la possibilité à l'utilisateur de marquer son point à l'aide d'une chaîne de caractères.

```
1 pointM <- function(v, mark = NULL){
2   is.null(mark) ||
3     (is.character(mark) &&
4       identical(length(mark), as.integer(1))) ||
5     stop("mark doit être une chaîne de caractères")
6
7   p <- point(v)
8   attr(p, "mark") <- mark
9
10  structure(p, class = c("pointM", class(p)))
11 }
```

## Exemple

Quelques remarques :

~→ On crée notre nouvelle classe en ajoutant un attribut à un point.

## Exemple

Quelques remarques :

- ~> On crée notre nouvelle classe en ajoutant un attribut à un point.
- ~> Pour ce faire, on fait appel à la fonction `attr<-`.



## Exemple

Quelques remarques :

- ~> On crée notre nouvelle classe en ajoutant un attribut à un point.
- ~> Pour ce faire, on fait appel à la fonction `attr<-`.
- ~> On peut ajouter un nombre arbitraire d'attributs à un objet.

## Exemple

Quelques remarques :

- ~> On crée notre nouvelle classe en ajoutant un attribut à un point.
- ~> Pour ce faire, on fait appel à la fonction `attr<-`.
- ~> On peut ajouter un nombre arbitraire d'attributs à un objet.
  - ~> Toutefois, certains tels que `class` jouent un rôle spécial (voir la documentation de `attr`).

## Exemple

Quelques remarques :

- ~> On crée notre nouvelle classe en ajoutant un attribut à un point.
- ~> Pour ce faire, on fait appel à la fonction `attr<-`.
- ~> On peut ajouter un nombre arbitraire d'attributs à un objet.
  - ~> Toutefois, certains tels que `class` jouent un rôle spécial (voir la documentation de `attr`).
- ~> Une autre possibilité aurait été d'utiliser une liste.

## Exemple

Quelques remarques :

- ~> On crée notre nouvelle classe en ajoutant un attribut à un point.
- ~> Pour ce faire, on fait appel à la fonction `attr<-`.
- ~> On peut ajouter un nombre arbitraire d'attributs à un objet.
  - ~> Toutefois, certains tels que `class` jouent un rôle spécial (voir la documentation de `attr`).
- ~> Une autre possibilité aurait été d'utiliser une liste.
  - ~> La première entrée est un `point`.

## Exemple

Quelques remarques :

- ~> On crée notre nouvelle classe en ajoutant un attribut à un point.
- ~> Pour ce faire, on fait appel à la fonction `attr<-`.
- ~> On peut ajouter un nombre arbitraire d'attributs à un objet.
  - ~> Toutefois, certains tels que `class` jouent un rôle spécial (voir la documentation de `attr`).
- ~> Une autre possibilité aurait été d'utiliser une liste.
  - ~> La première entrée est un `point`.
  - ~> La seconde entrée est la marque.

## Exemple

Quelques remarques :

- ~> On crée notre nouvelle classe en ajoutant un attribut à un point.
- ~> Pour ce faire, on fait appel à la fonction `attr<-`.
- ~> On peut ajouter un nombre arbitraire d'attributs à un objet.
  - ~> Toutefois, certains tels que `class` jouent un rôle spécial (voir la documentation de `attr`).
- ~> Une autre possibilité aurait été d'utiliser une liste.
  - ~> La première entrée est un `point`.
  - ~> La seconde entrée est la marque.
- ~> Notre approche comporte certains avantages.

## Exemple

```
1 > p1 <- pointM(1:2, "Premier point")
2 > p2 <- pointM(3:4, "Second point")
3 > p3 <- pointM(5:6, "Troisième point")
```

Comme nos **pointM**s héritent de **point**, on dispose déjà de quelques méthodes!

```
1 > p1
2 x = 1 & y = 2
3 > summary(p2)
4 Point de coordonnées x = 3 & y = 4
5 > isAligned(p1, p2, p3)
6 [1] TRUE
```

Conceptuellement, cela est juste : tout ce qui peut être fait avec un point devrait pouvoir se faire avec un point marqué.

## Exemple

Fournissons maintenant a l'utilisateur un moyen de voir la marque d'un point.



## Exemple

Fournissons maintenant a l'utilisateur un moyen de voir la marque d'un point.

```
1 mark <- function(point) UseMethod("mark")  
2  
3 mark.pointM <- function(point) attr(point, "mark")
```

## Exemple

Fournissons maintenant a l'utilisateur un moyen de voir la marque d'un point.

```
1 mark <- function(point) UseMethod("mark")
2
3 mark.pointM <- function(point) attr(point, "mark")
```

```
1 > mark(p1)
2 [1] "Premier point"
3 > mark(p2)
4 [1] "Second point"
5 > mark(p3)
6 [1] "Troisième point"
```

## Exemple

Malheureusement, il n'est pas possible de modifier la marque de la manière habituelle :

## Exemple

Malheureusement, il n'est pas possible de modifier la marque de la manière habituelle :

```
1 > mark(p1) <- "1er point"
2 Error in mark(p1) <- "1er point" : could not find function
  ↪ "mark<-"
3 No suitable frames for recover()
```

## Exemple

Malheureusement, il n'est pas possible de modifier la marque de la manière habituelle :

```
1 > mark(p1) <- "1er point"
2 Error in mark(p1) <- "1er point" : could not find function
  ↪ "mark<-"
3 No suitable frames for recover()
```

Remédions immédiatement à cette situation malheureuse :

## Exemple

Malheureusement, il n'est pas possible de modifier la marque de la manière habituelle :

```
1 > mark(p1) <- "1er point"
2 Error in mark(p1) <- "1er point" : could not find function
  ↳ "mark<-"
3 No suitable frames for recover()
```

Remédions immédiatement à cette situation malheureuse :

```
1 `mark<-` <- function(point, value) UseMethod("mark<-")
2
3 `mark<-.pointM` <- function(point, value){
4   attr(point, "mark") <- value
5
6   point
7 }
```

## Exemple

```
1 > mark(p1)
2 [1] "Premier point"
3 > mark(p1) <- "1er point"
4 > mark(p1)
5 [1] "1er point"
```

Finalement, spécialisons `print`.

## Exemple

```
1 > mark(p1)
2 [1] "Premier point"
3 > mark(p1) <- "1er point"
4 > mark(p1)
5 [1] "1er point"
```

Finalement, spécialisons `print`.

```
1 print.pointM <- function(x){
2   print.point(x)
3   cat(mark(x), "\n")
4 }

1 > p1
2 x = 1 & y = 2
3 1er point
4 > summary(p2)
5 Point de coordonnées x = 3 & y = 4
6 Second point
```



~→ Le système de fonction générique permet une *grande* flexibilité.

- ~> Le système de fonction générique permet une *grande* flexibilité.
- ~> Toutefois, cela a un coût : fonctions génériques  $\Rightarrow$  plus d'appels de fonctions (\$\$\$).

- ~> Le système de fonction générique permet une *grande* flexibilité.
- ~> Toutefois, cela a un coût : fonctions génériques  $\Rightarrow$  plus d'appels de fonctions (\$\$\$).
- ~> En général, pas un problème.

- ~> Le système de fonction générique permet une *grande* flexibilité.
- ~> Toutefois, cela a un coût : fonctions génériques  $\Rightarrow$  plus d'appels de fonctions (\$\$\$).
- ~> En général, pas un problème.
  - ~> Dans du code critique, il peut être utile d'appeler directement la méthode.

### Héritage multiple

↪ Une classe S3 peut avoir plusieurs classes filles.

### Héritage multiple

- ~→ Une classe S3 peut avoir plusieurs classes filles.
- ~→ Toutefois, elle ne peut avoir qu'une seule classe mère.

### Héritage multiple

- ~> Une classe S3 peu avoir plusieurs classes filles.
- ~> Toutefois, elle ne peut avoir qu'une seule classe mère.
- ~> On parle d'*héritage simple*.

### Héritage multiple

- ~> Une classe S3 peut avoir plusieurs classes filles.
- ~> Toutefois, elle ne peut avoir qu'une seule classe mère.
- ~> On parle d'*héritage simple*.
- ~> L'héritage multiple peut parfois être utile.



### Héritage multiple

- ~> Une classe S3 peu avoir plusieurs classes filles.
- ~> Toutefois, elle ne peut avoir qu'une seule classe mère.
- ~> On parle d'*héritage simple*.
- ~> L'héritage multiple peut parfois être utile.
- ~> Par exemple, `pointM` pourrait hériter de `point` et de `character`.

### Héritage multiple

- ~> Une classe S3 peu avoir plusieurs classes filles.
- ~> Toutefois, elle ne peut avoir qu'une seule classe mère.
- ~> On parle d'*héritage simple*.
- ~> L'héritage multiple peut parfois être utile.
- ~> Par exemple, **pointM** pourrait hériter de **point** et de **character**.
- ~> Pour cela, il faut utiliser S4.

### Dispatch multiple

~→ Nous avons vu que `UseMethod` ne permet le dispatch que sur un seul argument.

### Dispatch multiple

- ~> Nous avons vu que `UseMethod` ne permet le dispatch que sur un seul argument.
- ~> Par défaut, il s'agit du premier argument de la générique qui l'appelle.

### Dispatch multiple

- ~> Nous avons vu que `UseMethod` ne permet le dispatch que sur un seul argument.
- ~> Par défaut, il s'agit du premier argument de la générique qui l'appelle.
- ~> Il peut être utile de dispatcher sur plusieurs arguments.

### Dispatch multiple

- ~> Nous avons vu que `UseMethod` ne permet le dispatch que sur un seul argument.
- ~> Par défaut, il s'agit du premier argument de la générique qui l'appelle.
- ~> Il peut être utile de dispatcher sur plusieurs arguments.
- ~> Exemple : tambour et baguette, méthode `jouer`.

### Dispatch multiple

- ↪ Nous avons vu que `UseMethod` ne permet le dispatch que sur un seul argument.
- ↪ Par défaut, il s'agit du premier argument de la générique qui l'appelle.
- ↪ Il peut être utile de dispatcher sur plusieurs arguments.
- ↪ Exemple : tambour et baguette, méthode `jouer`.
- ↪ Pour cela, il faut utiliser S4.

## Références

---

- [1] Bjarne STROUSTRUP. *Bjarne Stroustrup's C++ Glossary*. English. Oct. 2012. URL : <http://www.stroustrup.com/glossary.html>.