LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Programmieren mit statistischer Software

Eva Endres, M.Sc.

Institut für Statistik
Ludwig-Maximilians-Universität München

Das S3-Objektsystem

Objektorientierte Programmierung I

- Wichtige Begriffe:
  - Objekt
  - Klasse: Definition der Struktur des Objektes
  - Instanz: Konkretes Objekt einer Klasse
  - Methode: Funktion, die auf ein Objekt dieser Klasse angewendet wird
- In R gibt es drei objekt-orientierte Systeme (nur zum Überblick)

```
> # S3: informal "generic-function" style
> # (Auswahl der Methode basierend auf der Klasse des ersten Arguments)
> ?class
> # S4: formal "generic-function" style
> # (Implementiert im Paket methods)
> ?Classes
> ?Methods
> # ReferenceClasses: "message-passing" style
> ?ReferenceClasses
```

Objektorientierte Programmierung II

- hier: mehr zum S3-Objektsystem
- Beispiel zur Auswirkung: dieselbe Funktion, angewendet auf zwei Vektoren, führt zu unterschiedlichen Ergebnissen

```
> par(mfrow = c(1, 2), mar = c(4, 4, 1, 0))
> plot(runif(10))
> plot(gl(5, 2))
```

Objektklassen I

- Im S3-System gibt es keine formale Klassendefinition
- Stattdessen: `class`-Attribut eines Objektes (Vektor mit Klassennamen)
- Verschiedene Möglichkeiten zur Definition einer Klasse

```
> a <- 11
> class(a) <- "PrimeNumber"
> a
[1] 11
attr(,"class")
[1] "PrimeNumber"

> b <- 42
> attr(b, "class") <- "PrimeNumber"
> b
[1] 42
attr(,"class")
[1] "PrimeNumber"
```

Objektklassen II

```
> # in einer Zeile:
> c <- structure(56, class = "PrimeNumber")
> c
[1] 56
attr(,"class")
[1] "PrimeNumber"
> # jetzt: zusaetzliches Attribut fuer jedes Objekt
> a
[1] 11
attr(,"class")
[1] "PrimeNumber"
> class(a)
[1] "PrimeNumber"
> attributes(a)
$class
[1] "PrimeNumber"
```

Objektklassen III

```
> b
[1] 42
attr(,"class")
[1] "PrimeNumber"
> class(b)
[1] "PrimeNumber"
> attributes(b)
$class
[1] "PrimeNumber"
```

Objektklassen IV

```
> c
[1] 56
attr(,"class")
[1] "PrimeNumber"
> class(c)
[1] "PrimeNumber"
> attributes(c)
$class
[1] "PrimeNumber"
```

Objektklassen V

- Die Definition einer Klasse ist für alle Objekte möglich, es wird nicht geprüft, ob die Klasse sinnvoll gewählt ist

```
> structure("10", class = "PrimeNumber")
[1] "10"
attr(,"class")
[1] "PrimeNumber"
> structure("Hello World!", class = "PrimeNumber")
[1] "Hello World!"
attr(,"class")
[1] "PrimeNumber"
```

- jedes Objekt hat eine Klasse

```
> class(pi)
[1] "numeric"
```

- aber nicht unbedingt ein Klassen-Attribut

```
> attributes(pi)
NULL
```

Objektklassen VI

- wenn kein Klassen-Attribut gesetzt ist, hat das Objekt eine implizite Klasse:

```
> # matrix
> d <- matrix(c(1,2,3,4), ncol=2)
> d
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> class(d)
[1] "matrix"
> attributes(d)
$dim
[1] 2 2
```

Objektklassen VII

```
> # array
> e <- array(data=d)
> e
[1] 1 2 3 4
> class(e)
[1] "array"
> attributes(e)
$dim
[1] 4
```

Objektklassen VIII

```
> # data.frame
> f <- data.frame(x=d[,1], y=d[,2])
> f
  x y
1 1 3
2 2 4
> class(f)
[1] "data.frame"
> attributes(f)
$names
[1] "x" "y"

$row.names
[1] 1 2

$class
[1] "data.frame"
```

Objektklassen IX

- Ergebnis von `mode(x)`, wenn nichts weiter definiert ist

```
> g <- runif(10)
> g
 [1] 0.91303049 0.62945084 0.55022797 0.57649307 0.49246849 0.47190111
 [7] 0.03498721 0.10845382 0.96382188 0.63575712
> mode(g)
[1] "numeric"
> class(g)
[1] "numeric"
> attributes(g)
NULL
```

Objektklassen X

```
> h <- gl(5, 2)
> h
 [1] 1 1 2 2 3 3 4 4 5 5
Levels: 1 2 3 4 5
> mode(h)
[1] "numeric"
> class(h)
[1] "factor"
> attributes(h)
$levels
[1] "1" "2" "3" "4" "5"

$class
[1] "factor"
```

Beispiel: Monte-Carlo Schätzung für pi I

```
> pi
[1] 3.141593
```

- Kreis mit Radius R, innerhalb eines Quadrats mit Seitenlänge $2 \cdot R$
- Fläche des Kreises: $pi \cdot R^2$, Fläches des Quadrats: $(2 \cdot R)^2$
- Verhältnis der Fläche des Kreises zur Fläches des Quadrats: $\frac{pi}{4}$
- Idee: wenn man n Punkte zufällig in dem Quadrat verteilt, dann sollten $n \cdot \frac{pi}{4}$ innerhalb des Kreises liegen

Beispiel: Monte-Carlo Schätzung für pi II

- Implementierung dieses Algorithmus

```
> mcpi <- function(n) {
+   stopifnot(is.numeric(n))
+   stopifnot(n >= 0)

+   x <- runif(n)
+   y <- runif(n)

+   inside <- ((x^2 + y^2) <= 1)

+   pi <- 4 * sum(inside) / n

+   # Rueckgabewert: Liste mit drei Argumenten
+   ret <- list(pi = pi, # das geschaetzte pi
+               n = n, # die Anzahl gezogener Punkte
+               sim = data.frame(x = x, y = y, inside = inside)
+               # die Simulationsdaten
+   )

+   # Definition der Klasse des Rueckgabewertes (vor Rueckgabe)
+   class(ret) <- "mcpi"

+   ret
+ }
```

Beispiel: Monte-Carlo Schätzung für pi III

- Funktionsaufruf

```
> set.seed(1234)
> mcpi(5)              # Objekt mit Klassen-Attribut
$pi
[1] 3.2

$n
[1] 5

$sim
          x           y inside
1 0.1137034 0.640310605   TRUE
2 0.6222994 0.009495756   TRUE
3 0.6092747 0.232550506   TRUE
4 0.6233794 0.666083758   TRUE
5 0.8609154 0.514251141  FALSE

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

Beispiel: Monte-Carlo Schätzung für pi IV

```
> mcpi_100 <- mcpi(100)
> class(mcpi_100)
[1] "mcpi"
```

Klassenmethoden I

- generische Funktionen, wie `print`, `summary`, `plot`

```
> print

function (x, ...)
UseMethod("print")
<bytecode: 0x00000000086fef20>
<environment: namespace:base>

> summary

function (object, ...)
UseMethod("summary")
<bytecode: 0x000000000a737560>
<environment: namespace:base>

> plot

function (x, y, ...)
UseMethod("plot")
<bytecode: 0x00000000087760b8>
<environment: namespace:graphics>

> # erkennbar an "UseMethod"
```

Klassenmethoden II

- zugehörige Methoden für spezifische Klassen

```
> methods("print")
> methods("summary")
> methods("plot")
> # Namenskonvention: generic.class
```

- die Klasse des ersten Arguments wird verwendet, um die jeweilige Methode zu wählen
- wenn keine solche Methode gefunden wird, wird die default-Methode verwendet (generic.default)
- Bezeichnung für diese Vorgehensweise: method dispatching

Klassenmethoden III

- Beispiel

```
> a <- 11
> class(a)
[1] "numeric"
> print(a)
[1] 11
> # probiert zuerst: print.numeric(a) - existiert nicht
> # probiert dann:   print.default(a)

> # neue Klasse fuer a:
> class(a) <- "PrimeNumber"
> print(a)
[1] 11
attr(,"class")
[1] "PrimeNumber"
> # fuehrt print.default(a) aus
```

Klassenmethoden IV

- Definition einer print-Methode für die Klasse PrimeNumber

```
> print.PrimeNumber <- function(x, ...) {
+   cat("Prime number:", x, "\n")
+ }
>
> print(a)
Prime number: 11
> # fuehrt print.PrimeNumber(a) aus
```

Ergänzung des Monte-Carlo pi-Beispiels I

- print-Methode für Monte-Carlo pi-Beispiel

```
> set.seed(1234)
> p <- mcpi(1000)
>
> print.mcpi <- function(x, ...) {
+   cat(x$pi, "(estimated by the Monte-Carlo method)\n")
+ }
>
> p
3.164 (estimated by the Monte-Carlo method)
```

Ergänzung des Monte-Carlo pi-Beispiels II

- summary-Methode für Monte-Carlo pi-Beispiel

```
> summary.mcpi <- function(object, ...) {
+   hits <- sum(object$sim$inside)

+   print(object)
+   cat(sprintf("Estimated by %s hits from %s trials.\n",
+               hits, object$n))
+ }
>
> summary(p)
3.164 (estimated by the Monte-Carlo method)
Estimated by 791 hits from 1000 trials.
```

Ergänzung des Monte-Carlo pi-Beispiels III

- oder mit paste

```
> summary.mcpi <- function(object, ...) {
+   hits <- sum(object$sim$inside)

+   print(object)
+   cat(paste("Estimated by", hits, "hits from", object$n, "trials.\n"))
+ }
>
> summary(p)
3.164 (estimated by the Monte-Carlo method)
Estimated by 791 hits from 1000 trials.
```

- Definition einer neuen generischen Methode, um auf den berechneten Wert von pi zuzugreifen

```
> estpi <- function(x, ...) {
+   UseMethod("estpi", x)
+ }
```

Ergänzung des Monte-Carlo pi-Beispiels IV

- Implementierung der generischen Methode für die Klasse mcpi

```
> estpi.mcpi <- function(x, ...) {
+   x$pi
+ }
>
> estpi(p)
[1] 3.164
```

Vererbung I

- Objekte können mehr als eine Klasse haben (Vektor von Klassennamen)

```
> class(a) <- c("PrimeNumber", "Number")
```

- Definiert Klassen-/Objekt-Hierarchien basierend auf der Reihenfolge der Klassennamen
  - hier: PrimeNumber erbt von Number
  - d.h.: wenn eine Methode für die Klasse Number definiert ist, aber nicht für PrimeNumber, dann wird diese verwendet

Vererbung II

- plot-Methode für Klasse Number

```
> plot.Number <- function(x, ...) {
+   # Werte zur Definition der x-Achse
+   x0 <- ifelse(x > 0, -1, +1)
+   x1 <- ceiling(x) + 1

+   plot(1, type = "n", xlim = c(x0, x1), ylim = c(-1, 1),
+        axes = FALSE, xlab = "Number line", ...)
+   arrows(0, 0, x, 0, lwd = 2)
+   axis(1, at = seq(x0, x1))
+   abline(v = 0, col = "gray", lty = 2)
+ }

> par(mar = c(4, 0, 0, 0))
> plot(a)
> # sucht zuerst plot.PrimeNumber - existiert nicht
> # sucht dann   plot.Number      - existiert und wird ausgefuehrt
```

Vererbung III

- allgemeines Schema:
  - Objekt "b", generische Funktion "generic"
  - class(b) <- c("first", ßecond")
  - Aufruf: generic(b)
  - erster Versuch: generic.first(b)
  - zweiter Versuch: generic.second(b)
  - letzter Versuch: generic.default(b)
- good practice: die Original-Klasse eines Objekts sollte immer zu neu definierten Klassen hinzugefügt werden!

```
> a <- 11
> class(a) <- c("PrimeNumber", "Number", class(a))
> class(a)
[1] "PrimeNumber" "Number"      "numeric"
```

Ergänzung des Monte-Carlo pi-Beispiels I

- Leibniz-Formel zur Schätzung von pi (unendliche Reihe)

```
> leibnizpi <- function(n) {
+   stopifnot(is.numeric(n))
+   stopifnot(n >= 0)

+   # Summand fuer i==0
+   approx <- 1
+   # Summanden fuer i==1 bis n
+   for ( k in seq(length = n) ) {
+     approx <- approx + (-1)^k / (2*k + 1)
+   }

+   pi <- 4 * approx

+   ret <- list(pi = pi, n = n)
+   class(ret) <- c("leibnizpi", "piest", class(ret))

+   ret
+ }
```

Ergänzung des Monte-Carlo pi-Beispiels II

- Funktionsaufruf

```
> pi_leibniz <- leibnizpi(100)
```

```
> mcpi <- function(n) {
+   stopifnot(is.numeric(n))
+   stopifnot(n >= 0)

+   x <- runif(n)
+   y <- runif(n)

+   inside <- ((x^2 + y^2) <= 1)

+   pi <- 4 * sum(inside) / n

+   # Rueckgabewert: Liste mit drei Argumenten
+   ret <- list(pi = pi, # das geschaetzte pi
+               n = n, # die Anzahl gezogener Punkte
+               sim = data.frame(x = x, y = y, inside = inside)
+               # die Simulationsdaten
+   )

+   # Definition der Klasse des Rueckgabewertes (vor Rueckgabe)
+   # Ergaenzung der Klasse des Rueckgabewertes wie fuer leibnizpi
+   class(ret) <- c("mcpi", "piest", class(ret))

+   ret
+ }
```

Ergänzung des Monte-Carlo pi-Beispiels III

- Funktionsaufruf

```
> pi_mc <- mcpi(100)
```

- generische plot-Methode für die Klasse piest

```
> plot.piest <- function(object, ...){
+   plot.Number(object$pi, ...)
+ }
```

- Funktionsaufruf

```
> par(mfrow=c(2,1), mar=c(4,0,4,0))
> plot(pi_leibniz, main="Leibniz")
> plot(pi_mc, main="Monte-Carlo")
```