V11 - Objektorientierte Programmierung

24. Mai 2021

Contents

1 Intro

1.1 Beispiel print()

`print()` gibt ein R-Objekt auf der Konsole aus. Die Art der Ausgabe hängt von der Klasse des Objektes ab.

```
x <- 0
print(x)
## [1] 0
print(structure(x, class="Date"))
## [1] "1970-01-01"
print(structure(x, class=c("POSIXct", "POSIXt")))
## [1] "1970-01-01 01:00:00 CET"
```

Wie würden wir eine solche Methode implementieren?

```
print2 <- function(x) {
  cls <- class(x)[[1]]
  if (cls == "Date") {
    print2.Date(x)
```

```
  } else if (cls == "POSIXct") {
    print2.POSIXct(x)
  } else {
    print2.default(x)
  }
}
# implementiere print2.Date, print2.POSIXct, print2.default
```

Angenommen wir möchten für eine neue Klasse an Objekten eine selbst-definierte `print()`-Funktion implementieren.

```
weight <- structure(x, class = "mass", unit = "kg")
print(weight)
## [1] 0
## attr(,"class")
## [1] "mass"
## attr(,"unit")
## [1] "kg"
print2.mass <- function(m) {
  print(paste(m, attr(m , "unit")))
}
print2.mass(weight)
## [1] "0 kg"
```

Wir müssten die `print()`-Funktion nochmal neu schreiben, um auch die neue Klasse behandeln zu können.

```
print2 <- function(x) {
  cls <- class(x)[[1]]
  if (cls == "Date") {
    print2.Date(x)
  } else if (cls == "POSIXct") {
    print2.POSIXct(x)
  } else if (cls == "mass") {
    print2.mass(x)
  } else {
    print2.default(x)
  }
}
print2(weight)
## [1] "0 kg"
```

Es geht auch einfacher und ohne das Verändern bestehender Funktionen.

```
print.mass <- function(m) {
  print(paste(m, attr(m , "unit")))
}
print(weight)
## [1] "0 kg"
```

Wie ist das nur möglich??? Antwort: Objektorientierte Programmierung und Polymorphismus!

1.2 Objektorientierte Programmierung

Objektorientierte Programmierung (kurz OOP) zeichnet sich (unter anderem) durch zwei Merkmale aus: **Polymorphismus** und **Datenkapselung** (*Encapsulation*).

Polymorphismus bedeutet, dass sich der selbe Funktionsname für verschiedene Funktionen nutzen lässt und abhängig von der Klasse der Argumente die richtige Funktion aufgerufen wird.

Bei Datenkapselung werden Daten und Funktionen, die auf diesen Daten operieren, zu einer Einheit zusammengefasst.

In der OOP wird die formale Beschreibung dieser Einheit als **Klasse** bezeichnet. Variablen, die dieser Beschreibung entsprechen, sind **Objekte** der Klasse. Die zur Klasse gehörenden Funktionen heißen **Methoden**, die Daten heißen **Felder** (*fields*).

Der Begriff **Objekt** wird mehrfach verwendet:

- In R: Alles, was existiert, ist ein Objekt (ein R-Objekt).
- In der OOP: Instanzen von Klassen heißen Objekte.

Zwei Unter-Paradigmen der OOP sind **encapsulated OOP** und **functional OOP**.

In der **encapsulated OOP** gehören Methoden direkt zu den Objekten oder Klassen und werden typischerweise als `object.method(arg1, arg2)` aufgerufen.

In der **functional OOP** werden **generische Funktionen** (*generics*) aufgerufen `generic(object, arg1, arg2)`, die in Abhängigkeit von der Klasse des Objekts die richtige Methode aufrufen.

1.3 OOP in R

In R gibt es gleich mehrere OOP-Systeme. In Base-R sind enthalten: S3, S4, *reference classes* (RC). Mit zusätzlichen Paketen können weitere OOP-Systeme geladen werden, zB: R6, R.oo, proto.

Wir besprechen die wichtigsten 3 OOP-Systeme: S3, S4, R6.

S3 ist die erste (1992) und simpelste Umsetzung von OOP in R. Es folgt dem Paradigma der functional OOP und erzwingt kaum Datenkapselung. Es gibt nur wenige Regeln, die S3 steuern. Dadurch ist es sehr flexibel, aber auch fehleranfällig.

S4 (1998) setzt ebenfalls auf functional OOP mit mehr Fokus auf Datenkapselung und deutlich strikteren Regeln als S3.

R6 (2017) folgt der encapsulated OOP. R6-Objekte folgen nicht der Copy-on-modify-Semantik.

1.4 Sloop

Um OOP in R besser zu verstehen, ist das Paket `sloop` nützlich. Wir werden es im Laufe dieses Kapitels des Öfteren einsetzen.

```
# install.packages("sloop")
library(sloop)
```

Mit `sloop::otype()` lässt sich feststellen, aus welchem OOP-System ein Objekt entstammt.

```
otype(1:10)
## [1] "base"
otype(tibble::tibble(x=1:3, y=3:1)) # Tibbles sind S3-Objekte
## [1] "S3"
otype(lubridate::hm("13:37")) # das Paket lubridate nutzt S4
## [1] "S4"
```

2 S3

S3 ist das am häufigsten verwendete OOP-System in R.

Es folgt dem Paradigma der funktionalen OOP.

S3 ist sehr simpel und dadurch leicht zu erlernen.

Wir kennen bereits einige S3-Klassen: `factor`, `data.frame`, `tibble`, `POSIXct`, `Date`, `proc_time`.

Ein S3-Objekt ist nichts anderes als ein Objekt mit Attribut `class`.

```
tyrion <- structure(list(name="Tyrion", age=38), class="Person")
milk <- structure(list(amount=0.75, animal="cow", time_good=0.05), class="Milk")
```

Methoden werden von generischen Funktionen aufgerufen. Dieser Vorgang wird mit **Method-Dispatch** bezeichnet.

Eine generische Funktion besteht aus dem Aufruf der Funktion `UseMethod()`.

```
elapse_time <- function(obj, time) {
  UseMethod("elapse_time")
}
```

Methoden sind Funktionen mit dem Namen `GENERIC.CLASS()`. Beim Aufruf der generischen Funktion wird die Klasse des ersten Argumentes abgefragt und versucht eine entsprechende Methode aufzurufen. Ist keine Methode vorhanden, wird `GENERIC.default()` aufgerufen, falls vorhanden.

```
elapse_time.Person <- function(obj, time) {
  obj$age <- obj$age + time
  return(obj)
}
elapse_time.Milk <- function(obj, time) {
  obj$time_good <- obj$time_good - time
  if (obj$time_good < 0) return("Milk is now spoiled!")
  return(obj)
}
elapse_time.default <- function(obj, time) {
  cat("cannot elapse time for object of class", class(obj), "\n")
}
str(elapse_time(tyrion, 1))
## List of 2
##  $ name: chr "Tyrion"
##  $ age : num 39
##  - attr(*, "class")= chr "Person"
str(elapse_time(milk, 1))
##  chr "Milk is now spoiled!"
str(elapse_time(structure(0, class="blub"), 1))
## cannot elapse time for object of class blub
##  NULL
```

Stil: Da für S3 Funktionsnamen mit `.` eine besondere Bedeutung haben, sollten “normale” (Funktions-)Namen kein `.` enthalten. Leider brechen auch viele Base-R Funktion mit dieser Konvention (zB `data.frame()`).

Einige bekannte Generics sind zB `print()`, `summary()`, `str()`, `mean()`. Mittels `sloop::ftype()` können wir herausfinden, ob eine Funktion generisch ist.

```
ftype(mean)
## [1] "S3"      "generic"
ftype(elapse_time)
## [1] "S3"      "generic"
ftype(apply)
## [1] "function"
```

S3 ermöglicht es, neue Klassen und Methoden zu definieren, ohne die bisherigen Funktionen verändern zu

müssen.

```
print.Person <- function(x) {
  print(sprintf("%s (%d)", x$name, x$age))
}
print(tyrion)
## [1] "Tyrion (38)"
tyrion # gleich print(tyrion)
## [1] "Tyrion (38)"
```

2.1 Method-Dispatch

Generische Funktionen rufen `UseMethod()` auf. Sonst sollten sie keine weiteren Befehle enthalten.

`UseMethod()` hat zwei Argumente:

1. (nötig) das Präfix der aufzurufenden Methoden, typischerweise identisch mit dem Namen der generischen Funktion
2. (optional) das Argument, das zum Method-Dispatch genutzt wird

```
# Dispatches on x
generic <- function(x, y, ...) {
  UseMethod("generic")
}

# Dispatches on y
generic2 <- function(x, y, ...) {
  UseMethod("generic2", y)
}
```

Beachte, dass keine (weiteren) Argumente der generischen Funktion an `UseMethod()` übergeben werden. Diese jedoch alle als Argumente der von `UseMethod()` aufgerufenen Methode zur Verfügung stehen. Um dies zu bewerkstelligen ist viel Dunkle Magie von Nöten. Einen kleinen Einblick in die Dunklen Künste wird uns das nächste Kapitel (Meta-Programmierung) geben.

Alle Methoden einer generischen Funktion werden von `sloop::s3_methods_generic()` angezeigt.

```
s3_methods_generic("elapse_time")
## # A tibble: 3 x 4
##   generic      class   visible source
##   <chr>        <chr>   <lgl>   <chr>
## 1 elapse_time default TRUE    .GlobalEnv
## 2 elapse_time Milk    TRUE    .GlobalEnv
## 3 elapse_time Person  TRUE    .GlobalEnv
```

2.2 Vererbung

Das Attribut `class` ist ein `character`-Vektor. Einträge nach dem ersten werden als Vorfahren der Klasse an Position 1 bezeichnet. Ein Objekt einer Klasse soll auch als Objekt seiner Vorfahren-Klassen fungieren können. Dadurch werden Eigenschaften der Vorfahren vererbt.

```
tib <- tibble::tibble(x=1:3, y=3:1)
class(tib)
## [1] "tbl_df"     "tbl"        "data.frame"
class(Sys.time())
## [1] "POSIXct" "POSIXt"
```

`inherits()` gibt aus, ob ein Objekt von einer bestimmten Klasse erbt.

```r
inherits(tib, "data.frame")
## [1] TRUE
inherits(tib, "Person")
## [1] FALSE
```

Beim Method-Dispatch wird Vererbung beachtet. Beim Aufruf einer generischen Funktion `function(arg1, arg2, ...) UseMethod(prefix)` wird bei S3 Klassen ein Vektor `paste0(prefix, ".", c(class(arg1), "default"))` erstellt und der Reihe nach Methoden dieser Namen gesucht. Die zuerst gefundene Methode wird ausgeführt.

```r
tyrion <- structure(
  list(name="Tyrion", age=38),
  house="Lannister",
  class=c("Noble", "Person")
)
varys <- structure(
  list(name="Varys", age=51),
  class=c("Person")
)
varys
## [1] "Varys (51)"
tyrion
## [1] "Tyrion (38)"
print.Noble <- function(x) {
  print(sprintf("%s of house %s (%d)", x$name, attr(x, "house"), x$age))
}
varys
## [1] "Varys (51)"
tyrion
## [1] "Tyrion of house Lannister (38)"
```

`sloop::s3_dispatch()` zeigt an, welche Methoden beim Aufruf einer generischen Funktion gesucht und gefunden werden.

```r
s3_dispatch(print(tib))
##    print.tbl_df
## => print.tbl
##  * print.data.frame
##  * print.default
s3_dispatch(print(varys))
## => print.Person
##  * print.default
s3_dispatch(print(tyrion))
## => print.Noble
##  * print.Person
##  * print.default
```

Bei impliziten Klassen (kein explizites `class`-Attribut, zB Matrizen) gibt `class()` leider keine vollständige Auskunft darüber, welche Methodennamen gesucht werden.

```r
x <- matrix(1:10, nrow = 2)
s3_dispatch(print(x))
##    print.matrix
##    print.integer
##    print.numeric
## => print.default
```

Die Funktion `sloop::s3_class()` zeigt die korrekte Liste an.

```
s3_class(x)
## [1] "matrix"  "integer" "numeric"
```

Der Aufruf von `NextMethod()` in einer Methode führt zum Aufruf der nächsten Methode in der Vererbungshierarchie.

```
showoff <- function(x) UseMethod("showoff")
showoff.default <- function(x) print("showoff: default")
showoff.a <- function(x) {
  print("showoff: a")
  NextMethod()
}
showoff.b <- function(x) {
  print("showoff: b")
  NextMethod()
}
x <- structure(list(), class = c("z", "a", "b", "c"))
s3_dispatch(showoff(x))
##    showoff.z
## => showoff.a
## -> showoff.b
##    showoff.c
## -> showoff.default
showoff(x)
## [1] "showoff: a"
## [1] "showoff: b"
## [1] "showoff: default"
```

2.3 Common Methods

2.3.1 Basis-Konstruktoren

Der **Konstruktor** einer Klasse ist eine Funktion, die ein Objekt der Klasse erstellt.

Die Namenskonvention für Konstruktoren ist `new_CLASS()`

```
new_Date <- function(x = double()) {
  stopifnot(is.double(x))
  structure(x, class = "Date")
}

new_Date(c(-1, 0, 1))
## [1] "1969-12-31" "1970-01-01" "1970-01-02"

new_Person <- function(x) {
  stopifnot(is.list(x))
  structure(x, class="Person")
}

new_Person(list(name="Daenerys", age=22))
## [1] "Daenerys (22)"
```

`stopifnot(expr)` gibt eine Fehlermeldung aus, falls `expr` nicht zu ausschließlich `TRUE` evaluiert.

```r
stopifnot(TRUE)
stopifnot(c(T, T))
stopifnot(1 == 0)
## Error: 1 == 0 is not TRUE
stopifnot("Haus")
## Error: "Haus" is not TRUE
```

Beim Erstellen einer Unterklasse sollten folgende Prinzipien beachtet werden:

1. Eine Unterklasse sollte auf dem selben Datentyp aufbauen wie die Elternklasse.
2. Das Klassenattribut der Unterklasse hat die Form `c(subclass, parent_class)`
3. Die Unterklasse sollte alle Felder und Attribute der Elternklasse haben.

Diese Prinzipien sollten durch den Konstruktor gesichert sein.

Um Unterklassen zu erlauben, sollte der Konstruktor der Elternklasse `...` und `subclass = NULL` als zusätzliche Argumente haben.

```r
new_my_class <- function(x, attr1, ..., subclass = NULL) {
  stopifnot(is.numeric(x))
  stopifnot(is.logical(attr1))
  structure(
    x,
    attr1name = attr1,
    ...,
    class = c(subclass, "my_class")
  )
}
new_subclass <- function(x, attr1, attr2, ..., subclass = NULL) {
  stopifnot(is.character(attr2))
  new_my_class(x, attr1, attr2name = attr2, ..., subclass = c(subclass, "subclass"))
}
```

```r
new_Person <- function(x, ..., subclass = NULL) {
  stopifnot(is.list(x))
  structure(
    x,
    ...,
    class = c(subclass, "Person")
  )
}
new_Noble <- function(x, house, ..., subclass = NULL) {
  stopifnot(is.character(house))
  new_Person(x, house=house, ..., subclass = c(subclass, "Noble"))
}
tyrion <- new_Noble(list(name="Tyrion", age=38), "Lannister")
str(tyrion)
## List of 2
##  $ name: chr "Tyrion"
##  $ age : num 38
##  - attr(*, "house")= chr "Lannister"
##  - attr(*, "class")= chr [1:2] "Noble" "Person"
```

2.3.2 Validators

Da S3-Klassen keine Struktur aufgezwungen wird, können leicht unsinnige Objekte entstehen.

Um zu prüfen, ob ein Objekt einer Klasse der erwarteten Struktur entspricht, biete es sich an eine Validator-Funktion `validate_CLASS()` zu schreiben.

```
validate_Person <- function(p) {
  if (!inherits(p, "Person")) stop('Object must inherit "Person".')
  x <- unclass(p)
  if (!is.list(x)) stop('Object must be a list.')
  if (is.null(x$name)) stop('List must contain the entry name.')
  if (is.null(x$age)) stop('List must contain the entry age.')
  if (!is.character(x$name)) stop('Name must be of type character.')
  if (!is.numeric(x$age)) stop('Age must be a numeric type.')
  if (length(x$name) != 1) stop('Name must be of length 1.')
  if (length(x$age) != 1) stop('Age must be of length 1.')
  invisible(p)
}
mr_cheese <- structure(list(name="Cheddar", weight=200), class="Person")
validate_Person(tyrion)
validate_Person(mr_cheese)
## Error in validate_Person(mr_cheese): List must contain the entry age.
```

2.3.3 Hilfs-Konstruktoren

Um das Erstellen von Objekten einer Klasse zu vereinfachen, definiert man entsprechende Hilfsfunktionen, mit dem Namen `CLASS()`.

```
Person <- function(name, age) {
  p <- new_Person(list(name=name, age=age))
  validate_Person(p)
  p
}
Person("Daenerys", 22)
## [1] "Daenerys (22)"
Noble <- function(name, age, house) {
  p <- new_Noble(list(name=name, age=age), house)
  validate_Person(p)
  p
}
dany <- Noble("Daenerys", 22, "Targaryen")
dany
## [1] "Daenerys of house Targaryen (22)"
```

2.3.4 Coercion

Um Coercion (Typenumwandlung) für eine selbst definierte S3-Klasse durchzuführen, empfiehlt es sich generische Funktionen vom Namen `as_CLASS()` zu definieren.

Achtung: In Base-R wird dieser Konvention nicht nachgekommen. Funktionen zur expliziten Typenumwandlung heißen dort meist `as.CLASS()`, zB `as.factor()`. Sie sind jedoch nicht Methoden der generischen Funktion `as()`. Sie sind auch selbst nicht generisch, wodurch etwa `as.factor()` nicht auf zusätzliche Klassen (als Eingabe) erweitert werden kann.

Die Methode `as_CLASS.CLASS()` sollte ihre Eingabe unverändert zurückgeben.

Die Methode `as_CLASS.default()` sollte ggf eine passende Fehlermeldung ausgeben.

```
as_Person <- function(x, ...) UseMethod("as_Person")
as_Person.Person <- function(x) x
```

```r
as_Person.character <- function(x, age) {
  Person(x, age)
}
as_Person.list <- function(x, ...) {
  p <- new_Person(x)
  validate_Person(p)
  p
}
as_Person(varys)
## [1] "Varys (51)"
as_Person(list(name="Arya", age=14))
## [1] "Arya (14)"
as_Person("Cersei", age=33)
## [1] "Cersei (33)"
```

Unterklassen sollte man in ihre Elternklasse wandeln können. Dies sollte mit einer entsprechenden Methode `as_CLASS.SUB_CLASS()` oder `as_CLASS.CLASS()` der Coercion-Generic `as_CLASS()` geschehen.

```r
as_Person.Noble <- function(x) {
  attr(x, "house") <- NULL
  class(x) <- "Person"
  x
}
tyrion
## [1] "Tyrion of house Lannister (38)"
as_Person(tyrion)
## [1] "Tyrion (38)"
```

2.4 Internal generics

Einige generische Funktionen rufen nicht `UseMethod()` auf, da sie in C implementiert sind und stattdessen entsprechende C-Funktionen aufrufen. Diese Funktionen heißen **internal Generics**. Siehe `?InternalMethods`.

```r
x <- sample(10)
s3_dispatch(x[1])
##    [.integer
##    [.numeric
##    [.default
## => [ (internal)
s3_dispatch(mtcars[1])
## => [.data.frame
##    [.default
## -> [ (internal)
`[[.Noble` <- function(x, i) {
  if (i == 1 || i == "name")
    return(sprintf("%s of house %s", x$name, attr(x, "house")))
  unclass(x)[[i]]
}
tyrion[["name"]]
## [1] "Tyrion of house Lannister"
tyrion[[1]]
## [1] "Tyrion of house Lannister"
tyrion[[2]]
## [1] 38
```

Internal Generics gibt es nur in Base-R. Internal Generics können nicht selbst definiert werden.

Internal Generics führen Method Dispatch nur bei expliziten Klassen durch, nicht bei impliziten.

```
f <- function(...) UseMethod("f")
f.Person <- function(...) return("f: Person")
f.numeric <- function(...) return("f: numeric")
f(tyrion) # method-dispatch for explicit class
## [1] "f: Person"
f(1) # method-dispatch for implicit class
## [1] "f: numeric"
# c() is internal generic
c.Person <- function(...) return("c: Person")
c.numeric <- function(...) return("c: numeric")
c(tyrion) # method-dispatch for explicit class
## [1] "c: Person"
c(1) # no method-dispatch for implicit class
## [1] 1
```

2.5 Group generics

Es gibt 4 Group-Generics: `Math`, `Ops`, `Summary` und `Complex`. Zu jeder Group-Generic gehört eine Gruppe interner generischer Funktionen. Zu `Summary` gehören etwa die Funktionen `all()`, `any()`, `sum()`, `prod()`, `min()`, `max()`, `range()`. Siehe `?groupGeneric`.

Funktionen, die zu einer Group-Generic gehören, haben in ihrer Dispatch-Liste zusätzlich Methoden der Group-Generics (hinter den eigenen Methoden). Ist keine spezifische Methode definiert, kann eine Group-Generic-Methode aufgerufen werden.

Group-Generics existieren nur in Base-R.

```
s3_dispatch(min(Sys.time()))
##    min.POSIXct
##    min.POSIXt
##    min.default
## => Summary.POSIXct
##    Summary.POSIXt
##    Summary.default
## -> min (internal)
min(Sys.time(), as.POSIXct("2020-02-20 20:20:20"))
## [1] "2020-02-20 20:20:20 CET"

df <- data.frame(a = c(-1,1), b = c(T,F))
s3_dispatch(sin(df))
##    sin.data.frame
##    sin.default
## => Math.data.frame
##    Math.default
##  * sin (internal)
sin(df)
##           a        b
## 1 -0.841471 0.841471
## 2  0.841471 0.000000
```

Als Beispiel implementieren wir die Group-Generic `Summary` für die Klasse `mass`.

```r
weight <- structure(1:3, class="mass", unit="kg")
weight
## [1] "1 kg" "2 kg" "3 kg"
sum(weight)
## [1] 6
s3_dispatch(sum(weight))
##    sum.mass
##    sum.default
##    Summary.mass
##    Summary.default
## => sum (internal)
Summary.mass <- function(x, ...) {
  m <- NextMethod()
  structure(m, class = "mass", unit = attr(x, "unit"))
}
s3_dispatch(sum(weight))
##    sum.mass
##    sum.default
## => Summary.mass
##    Summary.default
## -> sum (internal)
sum(weight)
## [1] "6 kg"
max(weight)
## [1] "3 kg"
```

2.5.1 Double Dispatch

Bei Generics der Group-Generic `Ops` (enthält `+`, `-`, `*`, ...) ist die ausgeführte Methode von der Klasse beider Argumente abhängig. Dies nennt man Multiple- oder Double-Dispatch.

Die Dispatch-Listen beider Argumente werden nach einer passenden Methode durchsucht. Falls die gefundenen Methoden gleich sind oder nur für ein Argument eine (externe) Methode gefunden wurde, wird diese angewendet. Ansonsten wird die interne Methode genutzt.

```r
john <- Person("John", 21)
`+.Person` <- function(x, y) {
  if ("Person" %in% class(x) && "Person" %in% class(y) &&
      all(sort(c(x$name, y$name)) == sort(c("John", "Daenerys"))))
    return("Happy End(?)")
  else
    return("...")
}
1 + 1
## [1] 2
1 + john
## [1] "..."
dany + 1
## [1] "..."
dany + john
## [1] "Happy End(?)"
```

3 S4

In S3 gibt es kaum Regeln. Viele unsinnige Konstrukte können erstellt werden.

```
x <- 1:5
class(x) <- "data.frame" # Unsinn
x
## NULL
## <0 rows> (or 0-length row.names)
```

Dies kann leicht zu unvorhersehbarem Verhalten und Fehlern im Code führen, die schwer aufzuspüren sind.

S4 ist in der Funktionsweise ähnlich zu S3, besitzt jedoch striktere Regeln, die einigen Unsinn von vorn herein unmöglich machen.

Wie S3 folgt auch S4 dem Paradigma der funktionalen OOP, ist dabei jedoch deutlich formaler und strikter als S3.

- S4-Klassen haben eine formale Definition via `setClass()`.
- S4-Klassen können mehrere (gleichberechtigte) Eltern haben (*multiple inheritance*).
- Felder von S4-Klassen heißen Slots. Auf sie wird mittels des Operators `@` zugegriffen.
- Methoden werden mittels `setMethod()` definiert.
- S4-Generics können Method-Dispatch in Abhängigkeit mehreren Argumente durchführen (*multiple dispatch*).

S4 ist im Paket `methods` implementiert. Dies ist in RStudio automatisch geladen, jedoch nicht unbedingt im Batch-Mode (Kommandozeilenprogramm `RScript`). Deswegen ist es sinnvoll bei der Nutzung von S4 das Paket explizit zu laden.

```
library(methods)
```

3.1 Klassen

S4-Klassen werden mit `setClass()` erzeugt. Dieser Funktion wird der Name der Klasse (typischerweise in UpperCamelCase), die Slots (Felder) und ggf die Eltern-Klasse (Argumentname `contains`) übergeben.

Slots werden als benannter `character`-Vektor übergeben. Die Namen werden zu den Namen der Slots. Die Werte sind die Klasse (auch implizite) der Slots.

```
.Person <- setClass("Person",
  slots = c(name = "character", age = "numeric")
)
.Employee <- setClass("Employee",
  contains = "Person",
  slots = c(boss = "Person")
)
```

Der Rückgabewert von `setClass()` ist der Basis-Konstruktors der Klasse. Er überprüft, ob die Klassen seiner Argumente den in `setClass()` geforderten Klassen entsprechen.

```
tyrion <- .Person(name = "Tyrion", age = 38)
.Person(name = "Night King", age = FALSE)
## Error in validObject(.Object): invalid class "Person" object: invalid object for slot "age" in class
```

Auf Slots kann mittels `@` oder `slot()` zugegriffen werden. Bei Zuweisungen wird Korrektheit von Typ/Klasse geprüft.

```
tyrion@name
## [1] "Tyrion"
slot(tyrion, "age")
```

```
## [1] 38
tyrion@age <- tyrion@age + 1
tyrion@name <- 42 # ERROR
## Error in (function (cl, name, valueClass) : assignment of an object of class "numeric" is not valid
```

`slotNames()` gibt die Namen aller Slots eines Objektes aus.

```
slotNames(tyrion)
## [1] "name" "age"
```

Um weitere Checks durchzuführen und mehr Kontrolle über die Erzeugung eines Objektes zu haben, werden weitere Konstruktoren definiert.

```
Person <- function(name, age = NA_real_, ...) {
  stopifnot(length(name) == 1 && length(age) == 1)
  .Person(name = name, age = age)
}
.Person() # Setzt Default-Werte.
## An object of class "Person"
## Slot "name":
## character(0)
##
## Slot "age":
## numeric(0)
Person()
## Error in stopifnot(length(name) == 1 && length(age) == 1): argument "name" is missing, with no defau
Person("Night King")
## An object of class "Person"
## Slot "name":
## [1] "Night King"
##
## Slot "age":
## [1] NA
```

Ein unbenanntes Argument des Basis-Konstruktors wird als Objekt der Eltern-Klasse interpretiert. Dadurch vereinfachen sich Konstruktoren von Unterklassen.

```
Employee <- function(name, age, boss) {
  person <- Person(name = name, age = age)
  .Employee(person, boss = boss)
}
```

Um die Klasse eines Objektes festzustellen, nutze `is()`.

```
is(tyrion)
## [1] "Person"
pod <- Employee("Podrick", 19, boss=tyrion)
is(pod)
## [1] "Employee" "Person"
is(pod, "Person")
## [1] TRUE
is(tyrion, "Imp")
## [1] FALSE
```

Der Typ eines S4 Objektes ist `S4`. Die Klasse ist die entsprechende S4-Klasse (ohne Eltern).

```
typeof(tyrion)
## [1] "S4"
class(tyrion)
## [1] "Person"
## attr(,"package")
## [1] ".GlobalEnv"
class(pod)
## [1] "Employee"
## attr(,"package")
## [1] ".GlobalEnv"
```

Das Paket `lubridate` stellt einige S4 Klassen bereit, zB `Period`.

```
library(lubridate)
##
## Attaching package: 'lubridate'
## The following objects are masked from 'package:base':
##
##     date, intersect, setdiff, union
time <- hms("1:23:45")
time
## [1] "1H 23M 45S"
class(time)
## [1] "Period"
## attr(,"package")
## [1] "lubridate"
typeof(time)
## [1] "double"
otype(time)
## [1] "S4"
slotNames(time)
## [1] ".Data"  "year"   "month"  "day"    "hour"   "minute"
time@minute
## [1] 23
time@.Data
## [1] 45
is(time, "Period")
## [1] TRUE
```

3.2 Generische Funktionen und Methoden

Eine neue generische S4 Funktion wird mit `setGeneric()` erzeugt. Dabei wird der Name der generischen Funktion übergeben und eine Funktion, die `standardGeneric()` aufruft.

```
setGeneric("myGeneric", function(x) standardGeneric("myGeneric"))
```

Methoden werden mit `setMethod(f, signature, definition)` hinzugefügt. `f` ist der Name der Generic als String, `signature` die Klasse dieser Methode, und `definition` die Funktion, die zu der entsprechenden Methode gemacht werden soll.

```
setMethod("myGeneric", "Person", function(x) {
  # method implementation
})
```

3.2.1 Show

S4-Objekte werden nicht durch `print()` ausgegeben sondern durch `show()`.

Um die Argumente von `show()` herauszufinden, finden wir diese Generic mittels `getGeneric()` und geben die Argumente mittels `formals()` aus.

```
names(formals(getGeneric("show")))
## [1] "object"
```

```
setMethod("show", "Person", function(object) {
  cat(is(object)[[1]], "\n",
      "  Name: ", object@name, "\n",
      "  Age:  ", object@age, "\n",
      sep = ""
  )
})
tyrion
## Person
##   Name: Tyrion
##   Age:  39
```

3.2.2 Zugriffsmethoden

Häufig wird ein Programmierstil verwendet, bei dem ein Nutzer einer Klasse nicht mit `@` oder `slot()` auf Solts zugreift, sondern mittels eigens erzeugter Zugriffsmethoden. Dadurch erhält der Erzeuger der Klasse mehr Kontrolle und kann die Validität der Objekte der Klasse sichern.

Zugriffsmethoden unterteilen sich in *Getter* (geben Wert zurück) und *Setter* (Zuweisung eines neuen Wertes).

Zugriffsmethoden können einfache Funktionen sein. Oft bietet es sich jedoch an sie als Methoden einer generischen Funktion zu erstellen.

```
setGeneric("name", function(x) standardGeneric("name"))
## [1] "name"
setMethod("name", "Person", function(x) x@name) # getter
name(tyrion)
## [1] "Tyrion"
```

```
setGeneric("name<-", function(x, value) standardGeneric("name<-"))
## [1] "name<-"
setMethod("name<-", "Person", function(x, value) { # setter
  stopifnot(length(value) == 1) # validity test
  x@name <- value
  x
})
name(tyrion) <- "Tyrion Lannister"
name(tyrion)
## [1] "Tyrion Lannister"
```

3.2.3 Coercion

Coercion wird mit `as()` durchgeführt.

Für verwandte Klassen gibt es eine natürliche Umwandlung.

```
as(pod, "Person")
## Person
```

```
##   Name: Podrick
##   Age:  19
```

```
tyrion_employee <- as(tyrion, "Employee")
tyrion_employee@boss
## Person
##   Name:
##   Age:
```

Um eigene Methoden zur Typenumwandlung der generischen Funktion `as()` hinzuzufügen, verwende `setAs()`.

```
setAs("Person", "Employee", function(from) {
  stop("Can not coerce an Person to an Employee", call. = FALSE)
})
as(tyrion, "Employee")
## Error: Can not coerce an Person to an Employee
```

3.2.4 Introspection

Zu einer Generic oder Klasse gehörenden Methoden können mit `sloop::s4_methods_generic()` bzw `sloop::s4_methods_class()` aufgelistet werden.

```
s4_methods_generic("initialize")
## # A tibble: 15 x 4
##    generic    class             visible source
##    <chr>      <chr>             <lgl>   <chr>
##  1 initialize .environment      TRUE    ""
##  2 initialize ANY               TRUE    "methods"
##  3 initialize array             TRUE    ""
##  4 initialize environment       TRUE    ""
##  5 initialize envRefClass       TRUE    "methods"
##  6 initialize externalRefMethod TRUE    ""
##  7 initialize matrix            TRUE    ""
##  8 initialize MethodsList       TRUE    ""
##  9 initialize Module            TRUE    "Rcpp"
## 10 initialize mts               TRUE    ""
## 11 initialize oldClass          TRUE    ""
## 12 initialize Period            TRUE    "lubridate"
## 13 initialize signature         TRUE    ""
## 14 initialize traceable         TRUE    ""
## 15 initialize ts                TRUE    ""
s4_methods_class("Person")
## # A tibble: 6 x 4
##   generic  class  visible source
##   <chr>    <chr>  <lgl>   <chr>
## 1 coerce   Person TRUE    R_GlobalEnv
## 2 coerce   Person TRUE    R_GlobalEnv
## 3 coerce<- Person TRUE    R_GlobalEnv
## 4 name     Person TRUE    R_GlobalEnv
## 5 name<-   Person TRUE    R_GlobalEnv
## 6 show     Person TRUE    R_GlobalEnv
```

3.3 Method-Dispatch

Der Method-Dispatch kann in S4 etwas komplizierter werden als in S3 wegen 2 zusätzlicher Features: *multiple Inheritance* (Klasse kann mehrere Eltern haben) und *multiple Dispatch* (Dispatch abhängig von Klassen mehrerer Argumente).

Für mehr Informationen zu Method-Dispatch in S4, siehe https://adv-r.hadley.nz/s4.html#s4-dispatch

3.4 S4 and existing code

Für die Argumente `slots` und `contains` der Funktion `setClass()` können S4-Klassen, S3-Klassen und implizite Klassen verwendet werden. Um eine S3 Klasse verwenden zu können, muss sie jedoch erst mittels `setOldClass()` registriert werden, zB `setOldClass("data.frame")`.

Die S3-Klassen aus Base-R werden in den Paketen von Base-R registriert. Wir müssen sie also nicht noch einmal selbst registrieren. Registrierung ist in erster Line für selbst erstellte S3-Klassen nötig.

Erbt eine S4-Klasse von einer S3-Klasse oder einer impliziten Klasse, ist das vererbte Objekt über den automatisch generierten Slot `.Data` abrufbar.

```
RangedNumeric <- setClass(
  "RangedNumeric",
  contains = "numeric",
  slots = c(min = "numeric", max = "numeric")
)
rn <- RangedNumeric(1:10, min = 1, max = 10)
rn@min
## [1] 1
rn@.Data
##  [1]  1  2  3  4  5  6  7  8  9 10
```

4 R6

R6 ist ein weiteres OOP-System für R. Die wichtigsten Unterschiede zu S3 und S4 sind:

- R6 setzt encapsulated OO um: Es gib keine generischen Funktionen; Methoden gehören zu Objekten.
- Keine Copy-on-modify-Semantik sondern Referenzsemantik.
- R6 gehört nicht zu Base-R. Es muss durch das Paket `"R6"` geladen werden.

```
library(R6)
```

4.1 Klassen und Methoden

Um R6-Klassen zu erzeugen, wird die Funktion `R6::R6Class()` benutzt. Dies ist die einzige Funktion aus dem Paket `R6`, die wir benötigen.

Das erste Argument von `R6::R6Class()` ist der Name der Klasse, nach Konvention in UpperCamelCase.

Das zweite Argument von `R6::R6Class()` trägt den Namen `public`. Wir übergeben damit eine Liste aller Methoden und Feldern, die einem Nutzer der erzeugten Klasse zur Verfügung stehen sollen. Namen von Methoden und Feldern werden nach Konvention in snake_case (alles klein, Wörter durch `_` getrennt) angegeben.

Methoden können auf andere Methoden und Slots mittels `self$ELEMENT` zugreifen. `self` entspricht dem aktuellen Objekt.

```
Accumulator <- R6Class("Accumulator", list(
  sum = 0, # a field
```

```
  add = function(x = 1) { # a method
    self$sum <- self$sum + x # method acesses field "sum" via "self$"
  })
)
```

Der Rückgabewert von `R6::R6Class()` ist ein Objektgenerator der erzeugten Klasse.

```
Accumulator
## <Accumulator> object generator
##   Public:
##     sum: 0
##     add: function (x = 1)
##     clone: function (deep = FALSE)
##   Parent env: <environment: R_GlobalEnv>
##   Locked objects: TRUE
##   Locked class: FALSE
##   Portable: TRUE
```

Mit der Methode `new()` eines Objektgenerators können Objekte erzeugt werden. Sie wird mittels `OBJECT_GENERATOR$new()` aufgerufen.

```
x <- Accumulator$new()
x
## <Accumulator>
##   Public:
##     add: function (x = 1)
##     clone: function (deep = FALSE)
##     sum: 0
```

Auf Felder und Methoden eines R6-Objektes wird mit `$` zugegriffen.

```
x$sum
## [1] 0
x$add(4)
x$sum
## [1] 4
```

Für R6-Objekte ist das `class` Attribut gesetzt. Alle R6-Objekte erben von der Klasse `R6`.

```
attributes(x)
## $class
## [1] "Accumulator" "R6"
```

4.2 Referenzsemantik

R6-Objekte und Objektgeneratoren sind Umgebungen. Dies erklärt den Zugriff auf Elemente mittels `$`.

```
typeof(Accumulator)
## [1] "environment"
typeof(x)
## [1] "environment"
```

Da R6-Objekte Umgebungen sind, folgen sie nicht der Copy-on-Modify-Semantik sondern der Referenzsemantik.

```
y <- x
y$sum <- 0
c(x$sum, y$sum)
```

```
## [1] 0 0
y$add(100)
c(x$sum, y$sum)
## [1] 100 100
```

Jedes R6-Objekt hat eine Methode namens `clone()`. Wegen der Referenzsemantik ist sie nötig, um Kopien eines Objektes zu erstellen.

```
z <- x$clone()
z$add(10)
c(x$sum, z$sum)
## [1] 100 110
```

Achtung: Enthalten R6-Objekte andere R6-Objekte als Felder, muss `clone(deep=TRUE)` aufgerufen werden, um diese ebenfalls zu kopieren (*shallow copy* vs *deep copy*).

Referenzsemantik angewendet auf Argumente von Funktionen wird mit *Call-by-Reference* (im Gegensatz zu *Call-by-Value*) bezeichnet.

```
x$sum
## [1] 100
nil_sum <- function(acc) {
  acc$sum <- 0
}
nil_sum(x)
x$sum
## [1] 0
```

Aus der Perspektive der funktionalen Programmierung erzeugen wir damit starke Nebeneffekte von Funktionen, die dazu führen, dass ihre Wirkung schwer vorhersagbar werden.

Für weitere Fähigkeiten von R6 (Konstruktor, Vererbung, private Elemente, ...) siehe https://adv-r.hadley.nz/r6.html.

4.3 Interna

Wird eine Methode namens `print()` definiert, wird die Ausgabe auf der Konsole durch diese Methode vollzogen.

```
Accumulator <- R6Class("Accumulator", list(
  sum = 0,
  add = function(x = 1) self$sum <- self$sum + x,
  print = function() cat("Sum is", self$sum, "\n"))
)
x <- Accumulator$new()
x
## Sum is 0
```

Bemerkung: R6 nutzt hierbei S3 Funktionalität: `print(x)` ruft `print.R6(x)` auf. Dabei ist `print.R6` zwar nicht im aktuellen Suchpfad vorhanden, jedoch suchen generische S3-Funktionen nach Methoden auch in der Umgebung `.__S3MethodsTable__.`, die zum Base-Paket gehört.

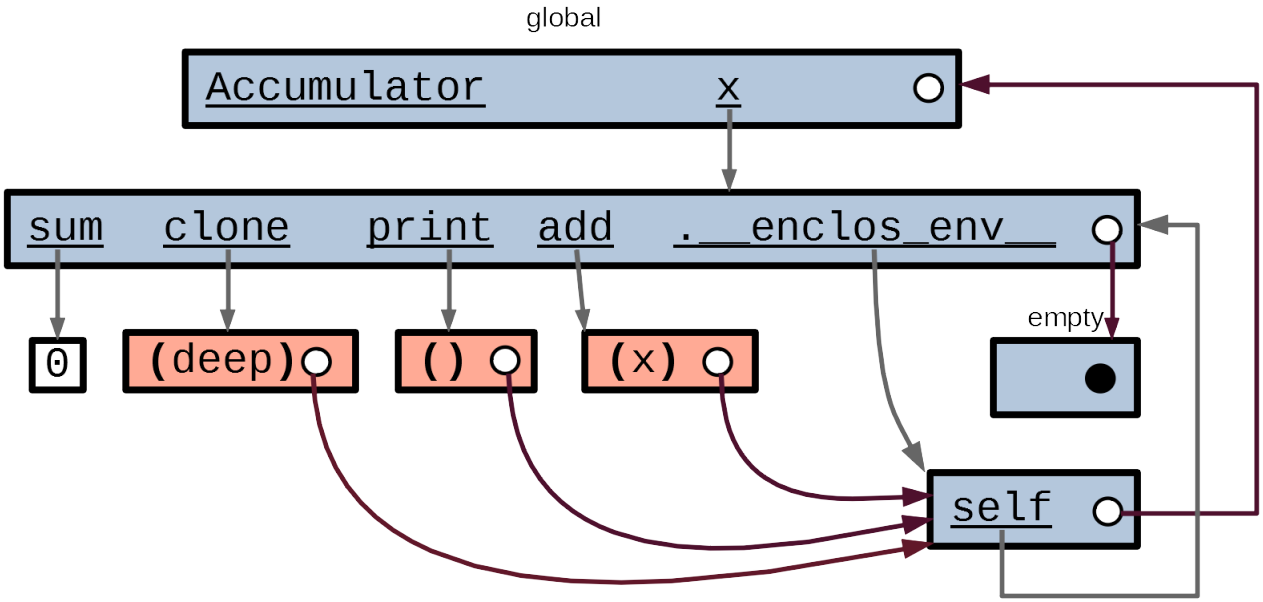
```
library(rlang)
class(x)
## [1] "Accumulator" "R6"
x # ruft print(x) auf
## Sum is 0
# print(x) ruft print.R6(x) auf
```

```
print.R6 # not in search path
## Error in eval(expr, envir, enclos): object 'print.R6' not found
rlang::base_env()$.__S3MethodsTable__.$print.R6
## function (x, ...)
## {
##     if (is.function(.subset2(x, "print"))) {
##         .subset2(x, "print")(...)
##     }
##     else {
##         cat(format(x, ...), sep = "\n")
##     }
##     invisible(x)
## }
## <bytecode: 0x000000001bd2f3a0>
## <environment: namespace:R6>
# .subset2(x,i) ist im Wesentlichen x[[i]]
# print.R6(x) ruft x[["print"]]() auf
```

Wir untersuchen den inneren Aufbau eines R6-Objektes.

```
typeof(x)
## [1] "environment"
env_print(x)
## <environment: 000000001BBA67E0> [L]
## parent: <environment: empty>
## class: Accumulator, R6
## bindings:
##  * .__enclos_env__: <env>
##  * sum: <dbl>
##  * clone: <fn> [L]
##  * print: <fn> [L]
##  * add: <fn> [L]
env_print(x$.__enclos_env__)
## <environment: 000000001BBA6498>
## parent: <environment: global>
## bindings:
##  * self: <Accumltr>
identical(x$.__enclos_env__$self, x)
## [1] TRUE
identical(fn_env(x$add), x$.__enclos_env__)
## [1] TRUE
```

Der Zugriff auf Elemente des Objektes mittels `self` im Inneren von Methoden, lässt sich also durch eine geschickt gewählte Funktionsumgebung erklären.


global
Accumulator
x
sum
clone
print
add
.__enclos_env__
empty
0
(deep)
()
(x)
self