# KAPSELUNG





## Kapselung

**Abstraktion für das Programmieren im Großen**

.. erlaubt das Aufteilen von Aufgaben in Moduln,
die von verschiedenen Programmierern implementiert werden

... erlaubt die Unterscheidung:
- Was soll ein Modul leisten?
- Wie soll er implementiert werden?

**Definition**

ein Modul ist eine Zusammenfassung von Vereinbarungen
die für einen bestimmten Zweck getroffen werden

der Modul kapselt seine Komponenten

typischerweise haben Moduln Schnittstellen (*interfaces*):
- einige Vereinbarungen werden exportiert
- die anderen sind nach außen verborgene Hilfsdefinitionen

**Arten von Moduln**
- einfache Pakete
- abstrakte Datentypen
- Objekte
- Klassen
- generische Pakete (parametrisierte Moduln)



## einfache Pakete

**einfache Zusammenfassung von Vereinbarungen**

verkörpert gekapselte Bindungen

Form (in Ada)

```
package /is
   D;
end /;
```

**ein ganz einfaches Beispiel (Ada)**

Vereinbarung

```
package physics is
   c : constant Float := 3.0e+8;
   G : constant Float := 6.7e-11;
   h : constant Float := 6.6e-34;
end physics;
```

Effekt: gekapselte Bindungen

{c |→ $3.0 \times 10^{8}$
G |→ $6.7 \times 10^{-11}$
h |→ $6.6 \times 10^{-34}$}

Benutzung

`physics.c` ... `physics.G` ... `physics.h`



## ein nicht ganz so einfaches Beispiel (Ada)

**“Raumschiff Erde”**

Vereinbarung

```
package Earth is
   type Continent is
      (Afr, Ame, Ant, Asi, Aus, Eur)
   radius : constant Float := 6.4e+6;
   area   : constant array (Continent) of Float :=
      (30.3e9, 42.7e9, 13.0e9,
       43.3e9,  7.7e9, 10.4e9);
   population : array (Continent) of Integer;
end Earth;
```

Effekt: gekapselte Bindungen

{ Continent |→ der Typ {Afr, Ame, Ant, Asi, Aus, Eur},
radius |→ $6.4 \times 10^{6}$
area |→ {Afr |→ $30.3 \times 10^{9}$, ... s, Eur |→ $10.4 \times 10^{9}$}
population |→ eine Feld-Variable}

Benutzung

```
for cont in Earth.Continent loop
   put (Earth.population(cont) / Earth.area(cont));
end loop;
```

**einfache Pakete in ML**

Strukturen

```
structure /=
   struct
    D
   end
```

## Schnittstellen (*interfaces, information hiding*)

**Moduln enthalten zweierlei Vereinbarungen**

- exportierte Vereinbarungen, die der Modul definiert
- verborgene Vereinbarungen, die der Modul nur intern benutzt

**die Schnittstelle definiert die exportierten Vereinbarungen**

Beispiel: in Ada ist dies eine Paket-Spezifikation

```
package trig is
   function sin (x : Float) return Float;
   function cos (x : Float) return Float;
end trig;
```

**der Rumpf beschreibt die Realisierung der Vereinbarungen**

Beispiel: in Ada ist dies ein Paketrumpf

```
package body trig is
   pi: constant Float := 3.1416...;
   function norm (x : Float) return Float is …;
   function sin (x : Float) return Float is …;
   function cos (x : Float) return Float is …;
end trig;
```

Effekt: gekapselte Bindungen

{ `sin` $\mapsto$ eine Funktions-Abstraktion, die den Sinus annähert,
`cos` $\mapsto$ eine Funktions-Abstraktion, die den Cosinus annäher}



## Verborgene Vereinbarungen in ML

**ML benutzt das allgemeinere Konzept der lokalen Vereinbarung**

Form

```
local
   D1
in
   D2
end
```

die Vereinbarungen $D_1$ gelten nur für $D_2$

**Beispiel**

Schnittstelle und Implementierung sind nicht vollständig separat

```
structure trig =
   struct
      local
         val pi = 3.1416...;
         fun norm (x : real) = …;
      in
         fun sin (x : real) = …;
         and cos (x : real) = …;
      end
   end
```

Benutzung

```
trig.sin (theta/2.0)
```



## Probleme mit konkreten Datentypen

**Ausgangspunkt**

Bisher haben wir Typen als Wertemengen aufgefaßt.
Für manche Problem-orientierte Typen ist diese Sicht unzureichend,
weil sie sich nicht gut mit vorhandene Typen repräsentieren lassen

**Beispiel: exakte rationale Zahlen in ML**

Darstellung als Zähler-Nenner-Paare ganzer Zahlen

```
datatype rational = rat of (int * int);
val   zero  = rat(0,1);
val   one   = rat(1,1);
fun op ++ (rat(m1,n1): rational,
           rat(m2,n2): rational) =
           rat(m1*n2 + m2*n1, n1 * n2)
```

Probleme:

- die Darstellung ist redundant: *rat*(2,3) ≡ *rat*(4,6) ≡ *rat*(-2,-3)
- manche Werte sind keine rationalen Zahlen: *rat*(3,0)
- vordefinierte Operationen für Tupel
  widersprechen der mathematischen Auffassung

```
one ++ rat(1, 2) = rat(3, 2) ~= rat(6, 4)
```

**Ursache**

`rational` hat die Werte { *rat*(*m*, *n*) | *m*, *n* ∈ Integer}
eigentlich wollen wir aber definieren:

{*rat*(*m*, *n*) | *m*, *n* ∈ Integer ; *n* > 0;
gcd(*m* , n) = 1}

aber einen solchen Typ können wir in ML nicht definieren
(wie in den meisten anderen Programmiersprachen)



## abstrakte Datentypen

**Definition**

ein abstrakter Datentyp
wird durch eine Menge von Operationen definiert.
typischerweise sind die Operationen
Konstanten, Funktionen und Prozeduren.
die Wertemenge eines abstrakten Typs ist nur indirekt gegeben
durch die Anwendung seiner Operationen,
beginnend bei den Konstanten
deshalb braucht man explizite Konstruktoroperationen

**rationale Zahlen als abstrakter Datentyp in ML**

```
abstype rational = rat of (int * int)
with
val zero = rat(0, 1);
val one  = rat(1, 1);
fun op // (m: int, n: int) =
      if n <> 0
      then rat(m,n)
      else error "no rational";
fun op ++ (rat(m1,n1): rational,
           rat(m2,n2): rational) =
           rat(m1*n2 + m2*n1, n1 * n2);
fun op == (rat(m1,n1): rational,
           rat(m2,n2): rational) =
           (m1*n2 = m2*n1)
```

# Eigenschaften abstrakter Datentypen

**Der abstrakte Datentyp `rat` erzeugt die Bindungen**

{ `rational` |→ ein abstrakter Datentyp,
`zero` |→ die rationale Zahl 0,
`one` |→ die rationale Zahl 1,
`by` |→ eine Operation, die rationale Zahlen konstruiert,
`plus` |→ eine Operation, die rationale Zahlen addiert,
`equal` |→ eine Operation, die rationale Zahlen vergleicht}

**Beobachtungen**

nur mit `zero`, `one`, `by` und `plus`
können `rat`'s aufgebaut werden (Operationserzeugtheit)

`plus` stellt sicher, daß der Nenner stets ungleich 0 bleibt

die Redundanz der Darstellung bleibt verborgen, denn
`one ++ (1//2) == (3//2) == (6//4)`

d.h. die Redundanz ist nicht beobachtbar,
sondern nur die gewünschten mathematischen Eigenschaften

die Darstellung von `rat` kann geändert werden,
ohne das der Benutzer es beobachten kann
(z. B. gekürzte Darstellung)

abstrakte Datentypen ähneln somit den vordefinierten Typen,
die auch nur über die vordefinierten Konstanten, Funktionen und Prozeduren beobachtet werden können,
und deren Darstellung ebenfalls verborgen bleibt.



# Abstrakte Datentypen in Ada

**rationale Zahlen**

Schnittstelle

```
package rational is
   type rat is private;
   zero : constant Rat;
   one  : constant Rat;
   function "/" (m,n: in Integer) return Rat;
   function "+" (r,s: in Rat) return Rat;
   function "=" (r,s: in Rat) return Boolean;
private
   type rat is record m,n: Integer; end;
   zero : constant Rat := (m=>0, n=>1);
   one  : constant Rat := (m=>1, n=>1);
end rational;
```

Implementierung

```
package body rational is
function "/" (a,b : in Integer) return Rat is
   begin
      return (m=>a, n=>b);
   end "/";
function "+" (r,s: in Rat) return Rat is
   begin
      return (m=>(r.m*s.n + s.m*r.n, n=>r.n * s.n);
   end "+";
function "=" (r,s: in Rat) return Boolean is
   begin
      return (r.m*s.n = s.m*r.n);
   end "=";
end rational;
```

**Beobachtung**

die Schnittstelle muß die Darstellung des Typs preisgeben
(wegen der getrennten Übersetzbarkeit)
die Darstellung ist außerhalb des Pakets nicht beobachtbar



# einzelne Objekte

**Definition**

Ein Objekt ist ein Modul, bestehend aus einer (verborgenen) Variablen,
auf die nur mit exportierten Operationen zugegriffen werden kann.
Typischerweise ist die Variable eine Tabelle o. ä.

die Darstellung der verborgenen Variablen kann geändert werden,
ohne das der Benutzer es beobachten kann

**Beispiel: Telefonverzeichnis in Ada**

```
package dir_object is
  procedure insert (name: in Name;
                    num: in Number);
  procedure lookup (name: in  Name;
                    num: out Number;
                    found: out Boolean);
end dir_object;
package body dir_object is
   type Dirptr is ...
   root: Dirptr;
   procedure insert … is begin … end insert;

   procedure lookup … is begin … end lookup;
begin
-- initialization
...
end dir_object;
```

Benutzung

```
dir_object.insert (me, 6041);
dir_object.lookup (me, mynumber, ok)
```



# Objektklassen

**Definition**

Eine Objektklasse ist eine Klasse von ähnlichen Objekten
die Darstellung der verborgenen Variablen kann geändert werden,
ohne das der Benutzer es beobachten kann

**Beispiel: Telefonverzeichnisse in Ada**

Objektklassen können mit generischen Paketen realisiert werden

```
generic package dir_class is
   procedure insert(name: in Name;
       num: in Number);
   procedure lookup(name: in  Name;
       num: out Number;
       found: out Boolean);
end dir_class;
package body dir_class is
   -- genau wie vorher
...
end dir_class;
```

Instanziieren der Objekte

```
package homedir is new dir_object;
package workdir is new dir_object;
```

Benutzung

```
homedir.insert (me, 6041);
workdir.insert (me, 8715);
workdir.lookup (me, mynumber, ok)
```

## Objektklassen und abstrakte Datentypen

**Beispiel: Telefonverzeichnis in Ada**

der Typ des Objektes wird ein abstrakter Datentyp

```
package dir_type is
   type Directory is limited private;
   procedure insert (dir: in Directory;
                     name: in Name;
                     num: in Number);
   procedure lookup (dir: in Directory;
                     name: in  Name;
                     num: out Number;
                     found: out Boolean);
private
   type Directory is …;
end dir_type;

package body dir_type is
   procedure insert … is begin … end insert;

   procedure lookup … is begin … end lookup;
   …
end dir_type;
```

Benutzung

```
use dir_type;
homedir, workdir : Directory;
insert (homedir, me, 6041);
insert (workdir, me, 8715);
lookup (workdir, me, mynumber, ok);
```

**Vorteile abstrakter Datentypen**

Werte sind erstklassig, Objekte nicht

Parametrisierung ist explizit

abstrakte Datentypen sind für alle Paradigmen sinnvoll



## Parametrisierte Moduln (*generics*)

**Generische Pakete sind Abstraktionen von Paketen (Vereinbarungen)**

Die Vereinbarung eines generische Pakete

abstrahiert von Vereinbarungen (einem Paket):

```
generic package dir_class is
   …
end dir_class;
package body dir_class is
   …
end dir_class;
```

Die Instanziierung eines generische Pakete

elaboriert den Pakerumpf und liefert Vereinbarungen

(ein neues Paket):

```
package homedir is new dir_object;
```

**der logische Schritt**

generische Pakete können auch parametrisiert werden

**mögliche generische Parameter**

- Konstanten
- Variablen
- Funktionen
- Prozeduren
- *Typen*



## generische Parameter (Konstanten)

**Beispiel Schlangen:**

Die Vereinbarung ist parametrisiert mit der Kapazität der Schlange:

```
generic
   capacity : in Positive;
package queue_class is
   procedure append (item : in  Character);
   procedure remove (item : out Character);
end queue_class;
package body queue_class is
   items : array (1..capacity) of Character;
   size, front, rear: Integer range 0..capacity;
   procedure append (item : in  Character) is
      …;
   procedure remove (item : out Character) is
      …;
begin
-- initialization
end queue_class;
```

Die Instanziierungen erzeugen Schlangen verschiedener Kapazitäten:

```
package line_buffer is new queue_class(120);
package terminal_buffer is new queue_class(80);
```



## generische Parameter (Typen)

**Beispiel Schlangen:**

Die Vereinbarung ist zusätzlich mit dem Element-Typ parametrisiert:

```
generic
   capacity : in Positive;
   type ITEM is private;
package queue is
   procedure append (i: in  ITEM);
   procedure remove (i: out ITEM);
end queue;
package body queue is
   items : array (1..capacity) of ITEM;
   size, front, rear: Integer range 0..capacity;
   procedure append (i: in  ITEM) is
   begin
      …; items(rear) := i; …;
   end append;
   procedure remove (i: out ITEM) is
   begin
      …; i := items(front); …;
   end remove;
begin
-- initialization
front := 1; rear:= 0;
end queue;
```

Die Instanziierungen erzeugen Schlangen verschiedenen Typs:

```
type Transaction is record … end record;
package audit_trail is
        new queue(120,Transaction);
```

## generische Typparameter

**Was ist zu beachten?**

Welche Operationen werden im Rumpf des generischen Paketes für den formalen Typparameter vorausgesetzt?

Im Beispiel `queue` ist das die Wertzuweisung

Dies muß für die generischen Typparameter spezifiziert werden

In Ada besagt die Vereinbarung

```
type ITEM is private;
```

daß `ITEM` eine Wertzuweisung und Gleichheitsabfrage besitzen soll.

**allgemeine Spezifikation von generischen Typparametern:**

**`type`** *`T`* **`is`** *Spezifikationen der Operationen für* `T`;

Für die generische Abstraktion muß überprüft werden, ob alle im Rumpf benutzen Operationen für *T* auch spezifiziert sind

Für jeder generische Instanziierung muß überprüft werden, ob das aktuelle Typargument tatsächlich die spezifizierten Operationen besitzt

Darüber hinaus möchte man eigentlich noch Eigenschaften der verfügbaren Operationen spezifizieren



## generische Parameter (Funktionen und Typen)

**Sortieren**

```
generic
   type ITEM is private;
   type SEQUENCE is
            array (Integer range <>) of ITEM;
   with function prec(x,y: Item) return Boolean;
package sorting is
   procedure sort (s: in out SEQUENCE);
   procedure merge(s1, s2: in SEQUENCE;
          s: out SEQUENCE);
end sorting;
package body sorting is
   procedure sort (s: in out SEQUENCE) is
   begin
      …
      if prec(s(j), s(i)) then ...
      …
   end sort;
   procedure merge(s1, s2: in SEQUENCE;
       s: out SEQUENCE) is
   begin
      ...
   end merge;
end sorting;
```

**die generischen Parameter hängen von einander ab:**

Das muß bei der Instanziierung berücksichtigt werden:

```
type FloatSequence is
   array (Integer range <>) of Float;
package ascending is
   new sorting(Float,FloatSequence, "<=");
package decending is
   new sorting(Float,FloatSequence, ">=");
```



## Übungen zu Kapselung

**Automatische Initialisierung von Moduln:**

Vorteile und Nachteile

**Implementiert den abstrakten Datentyp `rat` als Objekt-Klasse ??**

Schnittstelle

```
generic package rational is
   type rat is private;
   zero : constant Rat;
   one  : constant Rat;
   function "/" (m,n: in Integer) return Rat;
   function "+" (r,s: in Rat) return Rat;
   function "=" (r,s: in Rat) return Boolean;
private
   type rat is record m,n: Integer; end;
   zero : constant Rat := (m=>0, n=>1);
   one  : constant Rat := (m=>1, n=>1);
end rational;
```

Implementierung

```
package body rational is
function "/" (a,b : in Integer) return Rat is
   begin
      return (m=>a, n=>b);
   end "/";
function "+" (r,s: in Rat) return Rat is
   begin
      return (m=>(r.m*s.n + s.m*r.n, n=>r.n * s.n);
   end "+";
function "=" (r,s: in Rat) return Boolean is
   begin
      return (r.m*s.n = s.m*r.n);
   end "=";
end rational;
```

**Vervollständige das Paket `queue`**

```
generic
   capacity : in Positive;
```