

# Abstrakte Datentypen

## Sonderübung

Luisse Zieger & Lea Happel

PROG Sonderübung

December 4, 2018

1. Wiederholung
2. Schnittstellen (INTERFACE-Blöcke)
  - Generische Schnittstellen
  - Übungsaufgabe
3. ADT - Abstrakte Datentypen
  - Allgemeine Syntax
  - Beispiel
  - Nutzen von ADT
  - Komponentenzugriff
  - Beispiel
4. Übungsaufgabe
  - Typkonstruktor

# Was haben wir in der letzten Übung besprochen?

- **Funktionen**

- liefern ein Ergebnis, d. h. besitzen einen Rückgabewert
- Funktionsaufrufe stehen in Ausdrücken

- **Subroutinen**

- liefern **kein** Ergebnis
- werden mittels CALL-Anweisung aufgerufen

- **Rekursive UP** rufen sich selbst (direkt oder indirekt) auf

- In Fortran gibt es nur **Referenzparameter** (d.h. f. A. wird mit a. A. assoziiert = Alias)

- Attribute für formale Argumente sind:

**INTENT(IN), INTENT(OUT) und INTENT(INOUT)**

# Was haben wir in der letzten Übung besprochen?

## Vorteile von Modularisierung

- Strukturierung von Programmen
- Vermeidung von Coderedundanz
- Wiederverwendung unabhängiger Programmteile
- Kollaboration

# Schnittstellen (INTERFACE-Blöcke)

## Spezifische Schnittstellen

eindeutig definiert durch:

- Information, ob Subroutine oder Funktion
- Prozedurnamen/ Operatornamen bzw. -symbol/ Zuweisungssymbol (=)
- Liste der Argument- bzw. Operandentypen
- nur bei Funktionen: Ergebnistyp

## Generische Schnittstellen und Überladung

- Überladung = mehrfache Verwendung von Prozedurnamen bzw. Operatornamen/ -symbolen (+, −, \*, ..., auch: =) für verschiedene Prozeduren
- jede Überladung muss eindeutige **Schnittstelle** haben

# Generische Schnittstellen und Überladung

Bei der Überladung eines Operators, müssen die Funktionsargumente immer das Attribut `Intent(In)` haben!

```
INTERFACE < generi_name >  
    MODULE PROCEDURE <function_1> [, ... , <function_i>]  
                        bzw. <subroutine_1> [, ... , <subroutine_i>]  
END INTERFACE
```

```
INTERFACE OPERATOR (+)                                oder: -, *, /, <, .NOT., ... , .MYOPE.  
    MODULE PROCEDURE <function_1> [, ... , <function_i>]  
  
END INTERFACE
```

```
INTERFACE ASSIGNMENT (=)  
    MODULE PROCEDURE <subroutine_1> [, ... , <subroutine_i>]  
END INTERFACE
```

# Generische Schnittstellen und Überladung - ein Beispiel

```
MODULE shifted
  IMPLICIT NONE
  ...
  INTERFACE OPERATOR (.S.)
    MODULE PROCEDURE SHIFT
  END INTERFACE
  ...

CONTAINS

  FUNCTION shift (wort, anzahl)
    CHARACTER(*), INTENT(IN) :: wort
    INTEGER, INTENT(IN) :: anzahl
    CHARACTER(LEN(wort)) :: shift

    shift = ' '
    shift (anzahl + 1 : len(wort)) = wort(1 : len(wort) - anzahl)
    shift (1 : anzahl) = wort(len(wort) - anzahl + 1 : len(wort))
  END FUNCTION

  ...
END MODULE
```

# Übungsaufgabe

Überlade die Funktion, die den Binomialkoeffizient berechnet, mit dem Operator `.BIN. .`



# Abstrakte Datentypen (auch: benutzerdefinierte Typen)

- definiert heterogene Datenstruktur
- heterogen bedeutet: Komponenten können unterschiedliche Typen haben

## Allgemeine Syntax

```
TYPE < Typname >  
    [PRIVATE]
```

```
    < Typvereinbarung für Komponenten >
```

```
END TYPE
```

# Abstrakte Datentypen (auch: benutzerdefinierte Typen)

## Beispiel

```
TYPE fraction
  INTEGER :: nominator
  INTEGER :: denominator
END TYPE
```

Die Komponententypen können verschieden sein:

```
TYPE bike
  CHARACTER[15] :: brand
  INTEGER :: size
  ...
END TYPE
```

# Nutzen von ADT

- möglichst viel von der inneren Struktur verstecken
- Informationen, die zusammen gehören, aber unterschiedliche Datentypen haben, an einem Ort speichern (Zum Beispiel wird jeder Student über Name, s-Nummer und Matrikelnummer gekennzeichnet. Diese Infos sollten daher zusammen gespeichert werden.)
- erstellen von Datentypen, die in der Praxis gebraucht werden, aber nicht in F95 vorhanden sind (z.B. Rationale Zahlen, Intervallarithmetik, ...)

# Komponentenzugriff

**Problem:** Es gibt fast keine intrinsischen (d.h. vordefinierte) Funktionen dazu  
→ Alle gewünschten Funktionalitäten müsst ihr selbst programmieren!

Dazu benötigt ihr den **Komponentenzugriff**:

- die einzelnen Komponenten besitzen intrinsische Datentypen (d.h. Real, Integer, ...) und für diese gibt es intrinsische Funktionen  
→ ihr könnt damit arbeiten
- Allgemeine Syntax: **<NameDesADT> % <NameDerKomponenten>**

# Beispiel für Komponentenzugriff

## Beispiel

```
TYPE Student
  INTEGER :: Alter
  CHARACTER :: Vorname, Name
END TYPE
```

...

```
FUNCTION aelter (a,b)
  TYPE(Student) :: a,b
  LOGICAL :: aelter

  aelter = a%Alter > b%Alter
END FUNCTION
```

# Übungsaufgabe

Wir möchten ein Teilnehmerverzeichnis für unsere Sonderübung anlegen. Dabei sollen zu jedem Teilnehmer folgende Angaben gespeichert werden:

- Vorname (max. 30 Zeichen),
- Name (max. 30 Zeichen),
- Alter,
- Semester

Programmiere für diesen ADT folgende Funktionalitäten:

- eine Subroutine PUT die folgenden Satz ausgibt: "*Max Muster, 20 Jahre, ist Student im 1. Semester.*"
- eine Funktion, die die Studiendauer zweier Studenten vergleicht
- eine Funktion, die Nach- und Vornamen zweier Studenten lexikalisch vergleicht

# Typkonstruktor

Um im Programmablauf Variablen eures ADTs zu definieren, ohne sie einlesen zu müssen, eine selbstgeschriebene Funktion zu verwenden oder sie komponentenweise belegen zu müssen, könnt ihr den Typkonstruktor verwenden.

Allgemeine Syntax:  $\langle \text{NameDesADT} \rangle (\langle \text{Komponente1} \rangle, \langle \text{Komponente2} \rangle, \dots)$

## Beispiel

```
TYPE Student
  INTEGER :: Alter
  CHARACTER :: Vorname, Name
END TYPE

TYPE(Student) :: MaxMuster

MaxMuster = Student(20,Max,Muster)
```