Einführung in die Programmierung mit ABAP Objects

Objektorientiertes Programmieren ABAP

Prof. Dr. Peter Hohmann

Technische Hochschule Mittelhessen

FB MNI

Das objektorientierte Paradigma

- Objekt als "Weltbild".
- Objekte haben Attribute ("Speicherplätze") **und** Operationen (Methoden, "eigentliche Programme").
- Objekte tragen gekapselte Informationen in sich.
- Objekte können "Nachrichten empfangen und senden".
- Objekte haben untereinander Beziehungen.
- Objekte können ihre Eigenschaften vererben.
- In unterschiedlichen Objekten kann das Verhalten ein und derselben Nachricht verschieden sein (Polymorphie).

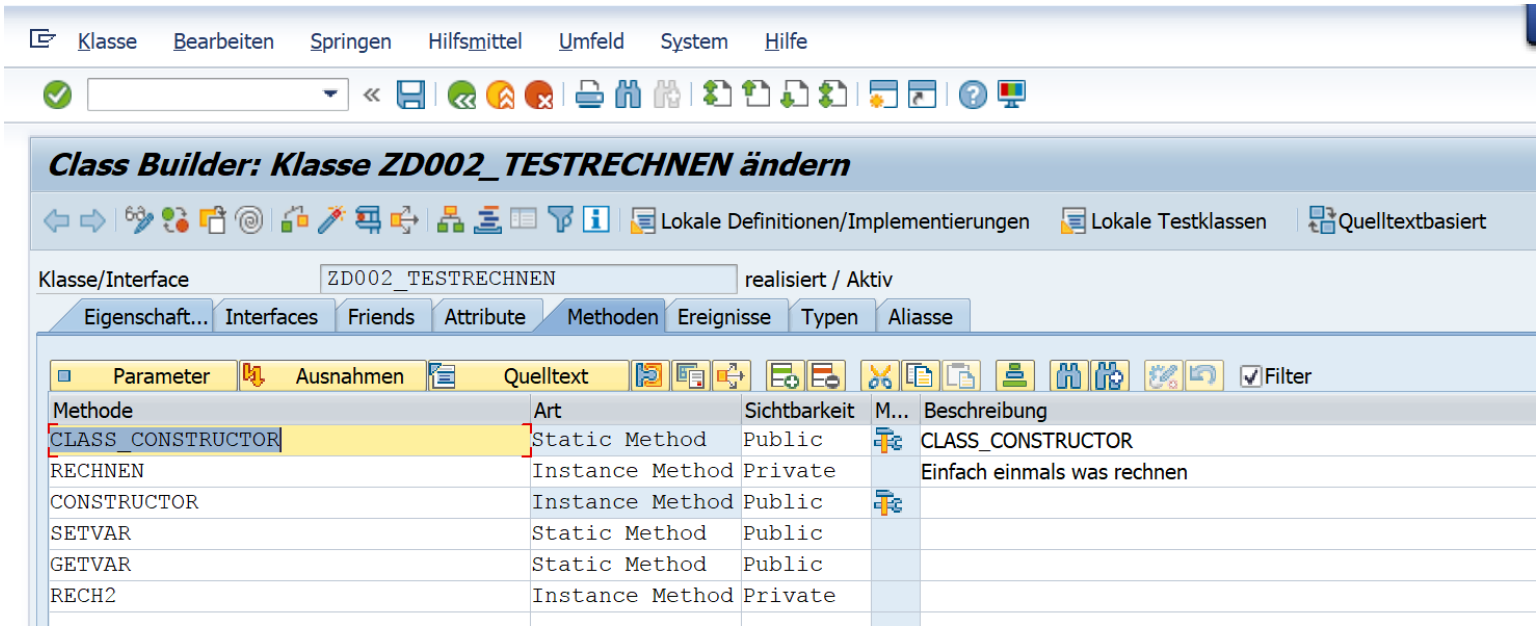
Objekte in SW-Systemen

- **Modellierung (UML) / Definition der Klasse**
  - Ergebnis einer Abstraktion
  - allgemeine Beschreibung gleichartiger Objekte (statisch) ➔ **Klasse**
  - Beschreibung der Eigenschaften ➔ **Speicherplätze**, **Variablen**
  - Beschreibung der Fähigkeiten / des Verhaltens (vgl. Prozeduren, Funktionen, "eigentliches Programm") ➔ **Methoden**

- **Erzeugung (Instanziierung)**
  - nur möglich, falls eine allgemeine Beschreibung existiert (**Klasse**)
  - **eindeutige Identifikation**: Adresse im Speicher (**"Referrenz"-Variable vom Typ der Klasse**)
  - **Zustand**: Belegung der Speicherplätze / Variablen mit **konkreten Werten** im privaten Speicher des Objekts
  - **Verhalten / Fähigkeiten**: "Programme", die das Objekt kennt (**Methoden**)

Klassenformen

- **Lokale Klassen im ABAP-Programm**
- **Globale Klassen mit dem CLASS-Builder (SE24)**



Konzepte von ABAP-Objects

- **Sichtbarkeit** (Public – Protected - Private)
- **Elemente einer Klasse (`DEFINITION und IMPLEMENTATION`)**
  - Methoden (Instanzmethoden, Statische Methoden)
  - Konstruktor (Klassen- und Instanzkonstruktur)
  - Attribute (Instanzattribute, Statische Attribute, Konstante)
  - Ereignisse/Events (Eventauslöser und Verarbeitungsmethode
- **Interfaces** (Gleiche Schnittstellen für Klassen, die das Interface verwenden)
- **Vererbung** (Public - Protected kann vererbt werden; FINAL beendet eine Vererbungshirachie)
- **Garbage-Collector** (Freigabe Hauptspeicher für nicht vorhandene Referenzen – DATA: ref_var type ref to classname. Create ref_var. Clear ref_var.)
- **Klassenbasierte Ausnahmen** (Fehlerklassen von CX_ROOT abgeleitet, TRY – ENDTRY, CATCH)

Sichtbarkeit

- **PUBLIC SECTION**: Bereich für alle öffentlichen Attribute und Methoden, voller Zugriff für alle äußeren Verwender (Klassen, Objekte), Schnittstelle der Klasse nach außen.

- **PROTECTED SECTION**: Geschützter Bereich für alle Attribute und Methoden in einer Vererbungshierarchie, Sichtbarkeit auf die Unterklassen beschränkt, Schnittstellen der Klasse zu ihren Unterklassen.

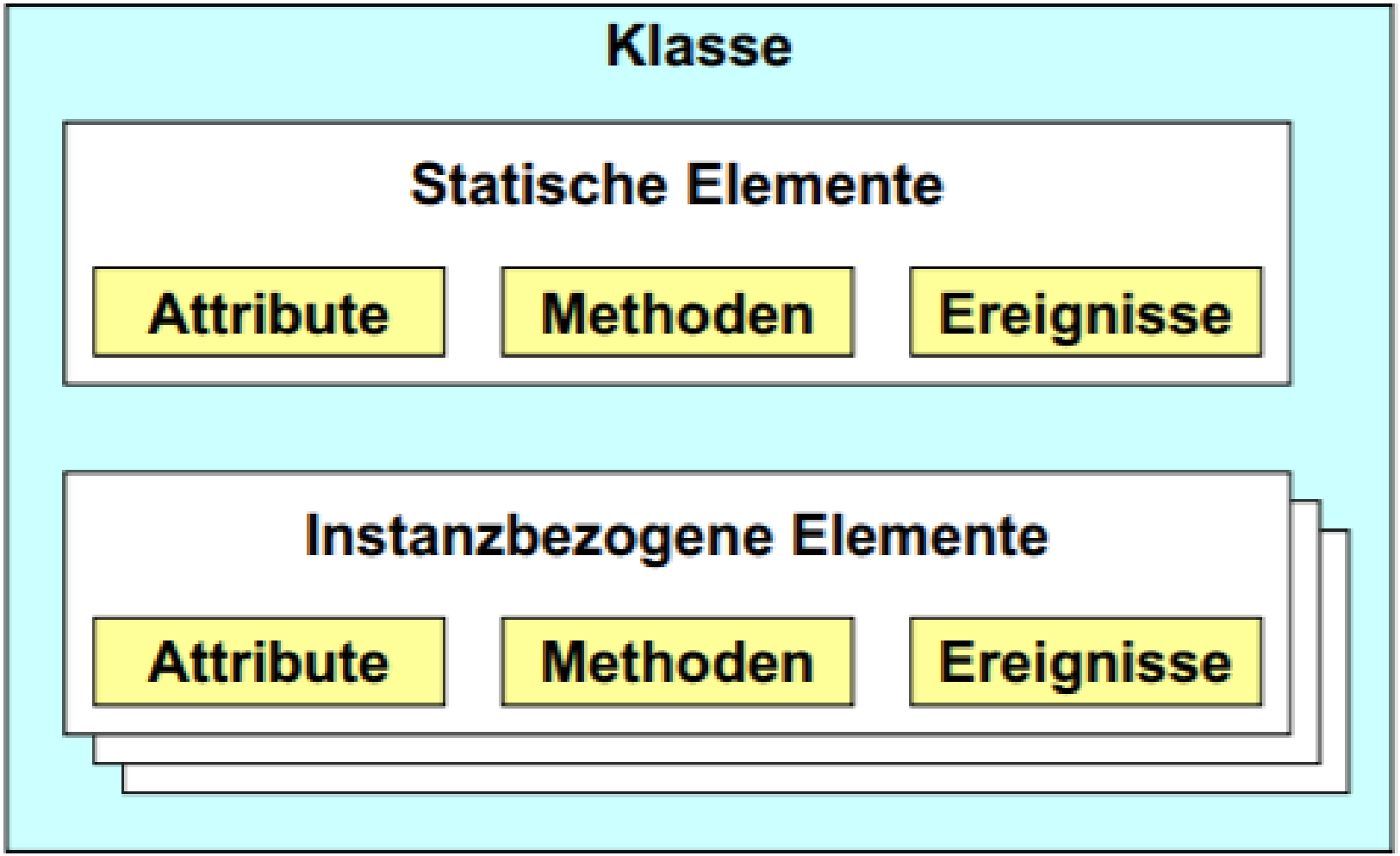
- **PRIVATE**: Geschützter Bereich für alle Attribute und Methoden der Klasse, Zugriff nur innerhalb der Klasse erlaubt.

Definition und –Implementierung von Klassen

```
* Definition ----------------------*
CLASS klassen_name DEFINITION .
    PUBLIC SECTION.
    ...
    PROTECTED SECTION.
    ...
    PRIVATE SECTION.
    ...
ENDCLASS.

* Implementierung -----------------*
CLASS klassen_name IMPLEMENTATION.
   ...
ENDCLASS.
```

Elemente einer Klasse



Attributdeklaration

Attribute sind interne Datenfelder für Objekte (Instanzattribute) und Klassen (Klassenattribute) von beliebigem ABAP-Datentyp.

```
* Klassenattribute / Instanzattribute
{ CLASS-DATA | DATA } name[(laenge)]
  {TYPE typ | LIKE name}
  [DECIMALS dezimal][VALUE wert][READ-ONLY].


* Datentypen und Konstanten sind instanzun-
* abhängig einmalig in der Klasse definiert
TYPES name[(laenge)] {TYPE typ | LIKE name}
                [DECIMALS dezimal].
CONSTANTS < siehe DATA-Anweisung > .
```

Instanziieren von Objekten mit Referenz-variable

Anlegen einer Referenzvariablen:

```
* Im Deklarationsteil des ABAP-Programms
DATA objekt_variable
                TYPE REF TO klassen_name .
```

Objekt erzeugen (Instanziieren):

```
* Im Anweisungsteil des ABAP-Programms
CREATE OBJECT objekt_variable .
```

ACHTUNG: Reihenfolgeabhängigkeit der Definition in Programmen

```
REPORT  z_aufgabe .

* DEFERRED: Klasse wird vor der Deklaration bekannt gemacht ----------*
CLASS lcl_Klasse_XYZ DEFINITION DEFERRED .

* Klasse wird „normal“ erstellt in der Verwendungsreihenfolge --------*
< Alle KlassenDefinitionen >

< Alle KlassenImplementationen >

< Datendeklarationen des Programms >
< Instanzvariable >

* Anweisungen des Programms ------------------------------------*
START-OF-SELECTION.
   < ABAP-Anweisungen >
```

Beispiel für DEFERRED

In diesem Beispiel verwendet die Klasse c1 die Klasse c2 und umgekehrt. Deshalb muss eine der Klassen vor ihrer eigentlichen Definition bekannt gemacht werden.

```
CLASS c1 DEFINITION DEFERRED.

CLASS c2 DEFINITION.
  PUBLIC SECTION.
    DATA c1ref TYPE REF TO c1.
ENDCLASS.

CLASS c1 DEFINITION.
  PUBLIC SECTION.
    DATA c2ref TYPE REF TO c2.
ENDCLASS.
```

Namenskonventionen Teil1

- Klassenbezeichner beginnen mit **cl**_ (global) bzw. **lcl**_ (lokal): *cl_projekt, lcl_projekt*.
- Interfaces beginnen mit **if**_ (global) bzw. **lif**_ (lokal): *if_projekt, lif_projekt*.
- Methoden beginnen mit einem Verb: *drive*. (Optional)
- Ereignisnamen werden in der Form <Substantiv><Verb in Partizipform> festgelegt: *projekt_stopped*.
- Bez. für Datentypen beginnen mit **ty**_: *ty_projekt_data*.
- Konstanten mit **co**_: *co_pi*.
- Variablennamen beginnen möglichst nicht mit einem Verb: *car_count*.

Namenskonventionen Teil2

OO-Namensgebung

- lcl_ = local class (Lokale Klasse, später lernen Sie auch globale Klassen kennen)
- im_ / ex_ / ch_ / re_ = Import- / Export- / Changing- / Returning-Parameter
- r_ = Referenz
- get_ = Getter (Methode zum Lesen von Attributen)
- set_ = Setter (Methode zum Schreiben von Attributen)

Funktionale-Namensgebung

- wa_ = workarea (Tabellenarbeitsbereich)
- it_ = interne Tabelle
- pa_ = Parameter
- so_ = select option (Selektionskriterium)

Methodendeklaration und Implementierung

Methodendeklaration und Implementierung

```
* Deklarationsteil
{CLASS-METHODS | METHODS} methoden_name
            [ABSTRACT][FINAL]
    [IMPORTING <Liste form. Parametern>]
    [EXPORTING <Liste form. Parametern>]
    [CHANGING <Liste form. Parametern>]
    [RETURNING VALUE(var) TYPE typ]
    [EXCEPTIONS <Liste mit Ausnahmen>] .

  * Implementierungsteil
  METHOD methoden_name .
     <ABAP-Anweisungen>
  ENDMETHOD .
```

Methodenparameter

Übergabe der Parameter per Referenz. Bei Wertübergabe muss der formale Parameter ***f_par*** durch **VALUE(*f_par*)** ersetzt werden.

```
* Liste der formalen Parameter:

{f_par | VALUE(f_par)} {TYPE typ | LIKE dobj}
      [OPTIONAL | DEFAULT d_wert]
   [ <nächster/weitere Parameter> ]...  .
```

Namenskonventionen: Methodenparameter beginnen mit

i_ bei IMPORTING-, ***e***_ bei EXPORTING-,
c_ bei CHANGING-, ***r***_ bei RESULT-Parameter,

um Verwechslungen mit den Attributen zu vermeiden.

Definition Referenz-/Wertübergabe

Referenzübergabe

Für die Referenzübergabe wird der Prozedur beim Aufruf eine Referenz auf den Aktualparameter übergeben, und sie arbeitet mit dem Aktualparameter selbst. Es wird kein lokales Datenobjekt für den Aktualparameter angelegt. Per Referenz übergebene Eingabeparameter können in der Prozedur nicht geändert werden (Ausnahme: **`USING`**-Parameter von Unterprogrammen).

WANN: **Referenzübergabe bei großen Datenmengen aus Gründen der Geschwindigkeit**

Wertübergabe

Bei der Wertübergabe wird für den Formalparameter ein typgerechtes lokales Datenobjekt als Kopie des Aktualparameters angelegt. Ausgabeparameter und Rückgabewerte werden beim Eintritt in die Prozedur initialisiert, und Eingabeparameter sowie Ein-/Ausgabeparameter erhalten den Wert des Aktualparameters übergeben. Ein geänderter Formalparameter wird nur bei fehlerfreier Beendigung der Prozedur an den Aktualparameter übergeben.

WANN: **Wertübergabe bei kleinen Datenmengen aus Gründen der Sicherheit**

Typisierung von Formalparameter

```
* vollständige Typangabe
TYPE { d | f | i | t | typ |
     LINE OF itab }

LIKE name
```

```
* Generische Typisierung
TYPE {ANY | c | n | p | x | csequence |
   [{ANY | INDEX | SORTED | HASHED}] TABLE}
```

Aufruf von Instanzmethoden

Aufruf von Instanzmethoden

```
CALL METHOD objekt_ref->methodenname
    [EXPORTING     f1 = a1 ... fn = an]
    [IMPORTING     f1 = a1 ... fn = an]
    [CHANGING      f1 = a1 ... fn = an]
    [RECEIVING     f1 = a1 ]
    [EXCEPTIONS    e1 = r1 ... en = rn]
    [OTHERS = r0] .
```

```
objekt_ref->methodenname (
    [EXPORTING     f1 = a1 ... fn = an]
    [IMPORTING     f1 = a1 ... fn = an]
    [CHANGING      f1 = a1 ... fn = an]
    [RECEIVING     f1 = a1 ]
    [EXCEPTIONS    e1 = r1 ... en = rn]
    [OTHERS = r0] ) .
```

Kurzformen: Aufruf von Instanzmethoden

```
objekt_ref->methodenname( ).

* falls nur IMPORTING-Parameter vorliegen
objekt_ref->methodenname( datenobjekt ).
objekt_ref->methodenname( f1 = a1 .. fn = an ).

* funktionale Methoden
variable = objekt_ref->methodenname( .... ).
```

Sonderformen: Aufruf von Instanzmethoden II

```
* Instanzmethoden innerhalb einer Klasse
  werden einfach mit deren Namen
  aufgerufen

methodenname( [<Parameter>] ) .

* ME ist eine Referenz auf die eigene
  Instanz (vergleiche this in Java)

ME->methodenname( [<Parameter>] ) .
```

Unterscheide IMPORTING – EXPORTING

- **Deklaration und Aufruf der Klassen**: Sichtweise des Entwicklers
  - IMPORTING: Die Methode erhält einen Parameter
  - EXPORTING: Methode gibt einen Parameter zurück
- **Aufruf von Methoden aus ABAP**: Sichtweise des Benutzers
  - EXPORTING: Programm übergibt einen Parameter an die Methode
  - IMPORTING: Programm erhält einen Parameter von der Methode
- **Sprechweise**: Mit IMPORTING- / EXPORTING-Parameter ist immer die Deklaration gemeint!

Beispiel: Einfache set-Methoden

```
REPORT zd002_oo_beispiel10.
CLASS lcl_bsp DEFINITION.
  PUBLIC SECTION.
    DATA name(20)   TYPE c.
    METHODS: set_name IMPORTING VALUE(im_name)
                                TYPE csequence .
ENDCLASS.
CLASS lcl_bsp IMPLEMENTATION.
  METHOD set_name .
    name = im_name .
  ENDMETHOD.
ENDCLASS.
DATA: r_beispiel TYPE REF TO lcl_bsp.

START-OF-SELECTION.
  CREATE OBJECT r_beispiel.
  CALL METHOD r_beispiel->set_name
    EXPORTING
      im_name = 'müller'.
```

Beispiel: REPORT zd002_oo_projekt

```
CLASS lcl_projekt DEFINITION.
  PUBLIC SECTION.
    METHODS:
      set_attribute
        IMPORTING
          im_projnr  TYPE zdev002_projnr
          im_projbez TYPE zdev002_projbez,
      ausgabe.
  PRIVATE SECTION.
    DATA: projnr  TYPE zdev002_projnr,
          projbez TYPE zdev002_projbez.
ENDCLASS.

CLASS lcl_projekt IMPLEMENTATION.
  METHOD set_attribute.
    projnr  = im_projnr.
    projbez = im_projbez.
  ENDMETHOD.
  METHOD ausgabe.
    WRITE: / projnr, projbez.
  ENDMETHOD.
ENDCLASS.
DATA: r_projekt TYPE REF TO lcl_projekt.
```


```
START-OF-SELECTION.
  CREATE OBJECT r_projekt.
  CALL METHOD r_projekt->set_attribute
    EXPORTING
      im_projnr  = '10'
      im_projbez = 'Sonderprojekt'.
  CALL METHOD r_projekt->ausgabe.
```

Klassenattribute und Klassenmethoden

Klassenattribute / Klassenmethoden I

```
* Deklaration Klassenattribute / -methoden
CLASS-DATA < siehe DATA-Anweisung >.

CLASS-METHODS methodenname
              < siehe Instanzmethoden > .


* Aufruf einer Klassenmethode
CALL METHOD klasse_name=>methodenname
                         [<Parameter>] .

klasse_name=>methodenname( [<Parameter>] ).
```

Klassenattribute / Klassenmethoden II

- **wiedersprechen dem objektorientierten Paradigma**

- **globale Typen / Konstanten / Variablen,**
  - für alle Instanzen der Klassen verfügbar.
  - Typen und Konstanten sind problemlos einsetzbar, da nicht veränderlich.
  - Klassenvariable problematisch, Einsatz nur in wenigen Fällen notwendig (siehe Instanzzähler, Singleton Pattern).

- **globale Methoden,**
  - benötigt keine Instanz, über den Klassenamen überall aufrufbar,
  - auch für alle Instanzen der Klasse verfügbar.
  - Einsatz nur in wenigen Fällen notwendig (Factory Pattern, in einer Funktionsbibliothek).

Beispiel: Klassenmethoden

```
REPORT ZD002_OBJEKT_LISTE_PROJEKT.

CLASS lcl_ausgabe DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS anzeige.
ENDCLASS.
*---------------------------------------------------------------------*
*       CLASS lcl_ausgabe IMPLEMENTATION
*---------------------------------------------------------------------*
CLASS lcl_ausgabe IMPLEMENTATION.
  METHOD anzeige.
    DATA: wa TYPE zdev002_projekt.
    SELECT * FROM zdev002_projekt INTO wa.
      WRITE: / wa-projnr, wa-projbez.
    ENDSELECT.
  ENDMETHOD.
ENDCLASS.
START-OF-SELECTION.
  lcl_ausgabe=>anzeige( ).
```

Funktionale Methoden

Funktionale Methoden

- Methoden mit IMPORTING- und nur einem Resultat-Parameter (Deklaration mit RETURNING …, Aufruf mit RECEIVING *f_parameter* = *a_parameter*).

- Direkte Weiterverarbeitung in
  - log. Ausdrücke (IF- CASE-, WHEN-Anweisungen, WHERE-Klauseln bei internen Tabellen),
  - arithmetische Ausdrücke,
  - Quellfeld einer Move-Anweisung.

```
IF neu_Kunde->get_Kredit_limit( ) < 10000 .
   <Kundenssatz verarbeiten>
ENDIF .
```

Beispiel: Funktionale Methoden

```
CLASS bsp DEFINITION.
PUBLIC SECTION.
  TYPES ty_name(20)   TYPE c.
  DATA name   TYPE ty_name READ-ONLY.

  METHODS: get_name RETURNING VALUE(r_name)
                        TYPE ty_name .
ENDCLASS.

CLASS bsp IMPLEMENTATION.
  METHOD get_name .
    r_name = name .
  ENDMETHOD.
ENDCLASS.
```

Konstruktoren

Konstruktoren

- Spezielle Methoden mit einem fest definierte Namen: CONSTRUCTOR bzw. CLASS_CONSTRUCTOR (sind geschützt).

- Konstruktoren müssen in der PUBLIC SECTION definiert werden, tatsächliche Zugriffsrechte werden bei der Klasse festgelegt (siehe erweiterte Zugriffsrechte).

- Instanzkonstruktoren: CONSTRUCTOR
  - Können nur IMPORTING-Parameter besitzen,
  - beim Werfen einer EXCEPTION wird die Instanz nicht angelegt,
  - wird automatisch einmal beim Instanziieren (CREATE OBJECT ....) aufgerufen.
  - CREATE OBJECT hat dieselben Zusätze (EXPORTING, EXCEPTION) wie der Methodenaufruf mit Parameter.

- KLassenkonstruktoren: CLASS_CONSTRUCTOR
  - Werden genau einmal beim ersten Zugriff auf die Klasse ausgeführt,
  - Haben keine Parameter und werfen keine EXCEPTION.

Instanziieren von Objekten mit Parameter für den Instanzkonstruktor

Anlegen einer Referenzvariablen:

```
* Im Deklarationsteil des ABAP-Programms
DATA objekt_ref
                TYPE REF TO klassen_name .
```

Objekt erzeugen (Instanziieren):

```
* Im Anweisungsteil des ABAP-Programms
Übergabe von EXPORT-Werten für den
Instanzkonstruktor
CREATE OBJECT objekt_ref
    [EXPORTING     f1 = a1 ... fn = an]
    [EXCEPTIONS    e1 = r1 ... en = rn].
```

Beispiel: Konstruktor mit 2 Parameter

```
CLASS bsp DEFINITION.
PUBLIC SECTION.
  DATA: name(25)     TYPE c READ-ONLY ,
        vorname(25)  TYPE c READ-ONLY .
  METHODS:
    constructor IMPORTING i_name    TYPE csequence
                          i_vorname TYPE csequence .
ENDCLASS.
CLASS bsp IMPLEMENTATION.
  METHOD constructor .
    name     = i_name .
    vorname  = i_vorname .
  ENDMETHOD.
ENDCLASS.
```

Erweiterte Zugriffsrechte

```
CLASS klassen_name DEFINITION
    [CREATE {PUBLIC | PROTECTED | PRIVATE}]
    [FRIENDS klasse_1 ... klasse_n].
```

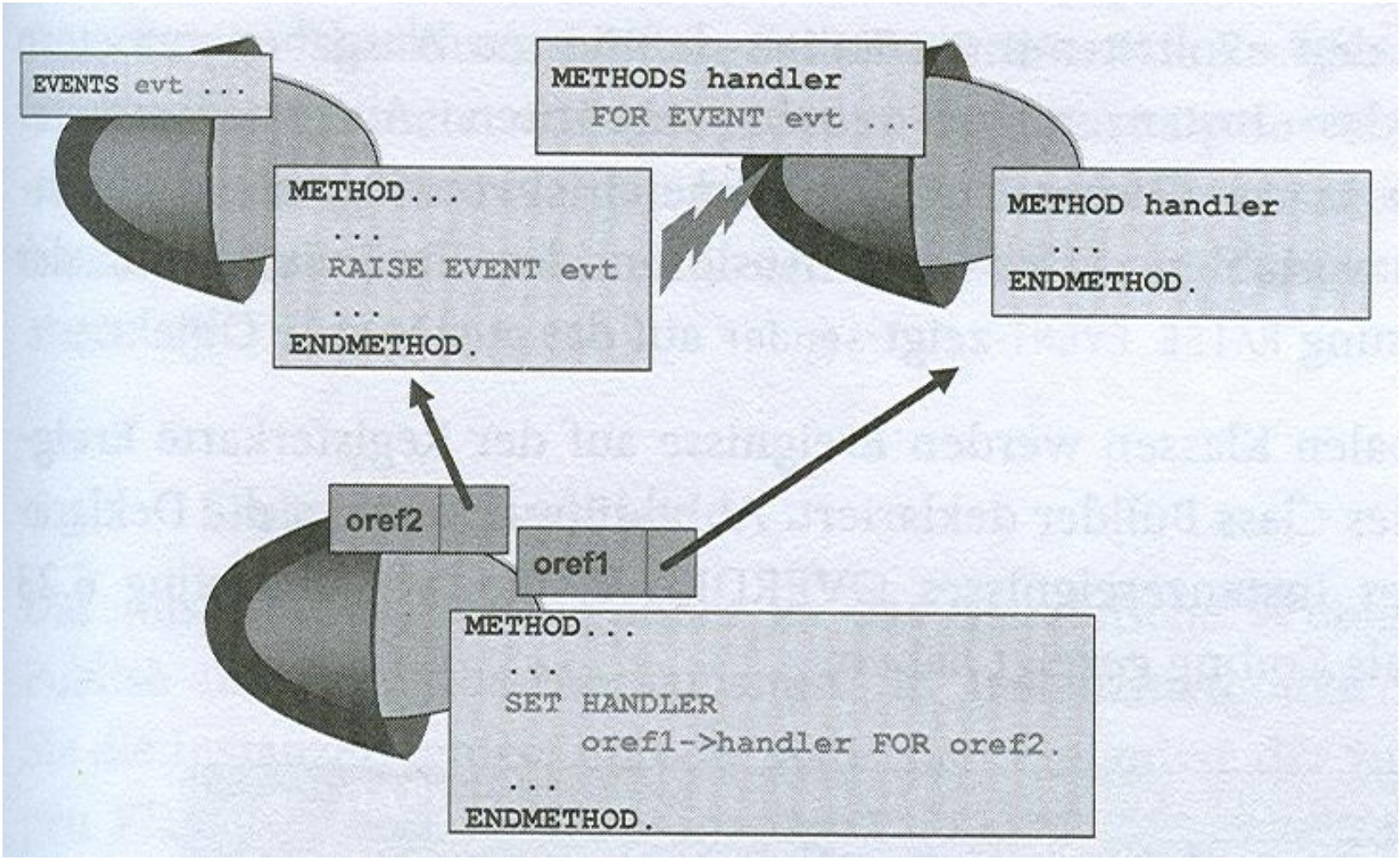
Steuerung der Instanziierung: Öffentliche, geschützte, private Instanziierbarkeit

Freundschaft unter Klassen ermöglicht den Zugriff auf private und geschützte Komponenten.

Freundschaft ist ein einseitiges Verhältnis, sie ist nicht vererbbar.

Ereignisse

Ereignisse


EVENTS evt ...
METHODS handler
FOR EVENT evt ...
METHOD...
...
RAISE EVENT evt
...
ENDMETHOD.
METHOD handler
...
ENDMETHOD.
oref2
oref1
METHOD...
...
SET HANDLER
oref1->handler FOR oref2.
...
ENDMETHOD.

Definition eines Ereignisses

```
* Ereignisse werden ähnlich wie Methoden
* definiert, die optionale Parameterschnitt-
* Stelle legt die Schnittstelle des Behandlers
* fest. Impliziter Parameter: sender.

CLASS <Klassenname> DEFINITION.
  ... SECTIONS.
    {EVENTS: | CLASS-EVENTS:}
       <Ereignis> [EXPORTING VALUE(f_i)
                             TYPE typ ...].
...

ENDCLASS.
```

Auslösen eines Ereignisses

```
* In einer Methode der Klasse wird das
* Ereignis ausgelöst. Die Parameterschnitt-
* Stelle muss mit der Schnittstelle des
* Ereignisses korrespondieren.

CLASS <Klassenname> IMPLEMENTATION.
  METHOD <methodenname>.
    ...
    RAISE EVENT <Ereignis>
       [EXPORTING e_i = a_i  ...].
    ...
  ENDMETHOD .
ENDCLASS.
```

Ereignisbehandler

```
* Jede Klasse kann Ereignisbehandler für die
* Ereignisse enthalten. Sind speziell ge-
* kennzeichnete Methoden. Schnittstelle muss
* mit der des Ereignisses korrespondieren.
* Eingabeparameter werden nicht typisiert,
* Typisierung wird von den Ausgabeparametern
* des Ereignisses übernommen. Typ des impliz.
* Parameters sender ist <Klassenname>.

[CLASS-]METHODS <Handlername>
      FOR EVENT <Eventname> OF <Klassenname>
      IMPORTING  ... ei ... [sender] .
```

Ereignisbehandler registrieren (abonnieren)

```
* Damit ein Ereignisbehandler auf ein ausge-
* löstes Ereignis reagiert, muss er zu Lauf-
* zeit registriert werden. Erst die Registrie-
* rung koppelt Ereignisbehandler an Auslöser.
* Die Kopplung kann jederzeit wieder aufge-
* hoben werden. Bei Angabe von act='X' wird
* registriert (Standard), bei act=' ' deregi-
* striert.

SET HANDLER handler1 handler2 ...
       FOR {object_ref | ALL INSTANCES}
          [ ACTIVATION act ] .
```

Erweiterungen

Lokale Klassen für Listen

Lokale Klassen für einen Taschenrechner

Globale Klasse für den Taschenrechner