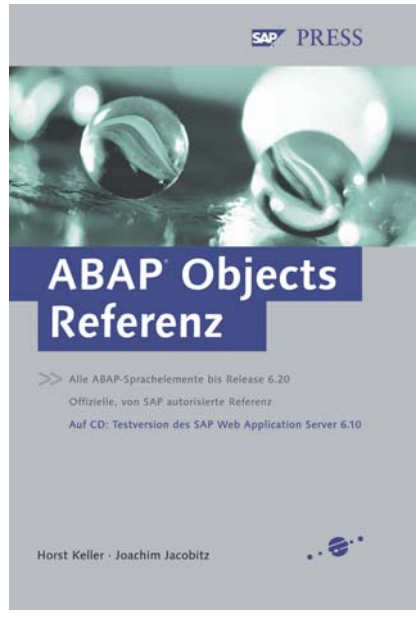
Horst Keller, Joachim Jacobitz

ABAP Objects-Referenz



Galileo Press

Inhalt

Vorwort 25

1 Einführung und Übersicht 29

1.1 Ziel des Buches 29
1.2 Beschriebene Releases 29
1.3 SAP-Basis und Web Application Server 30
1.4 ABAP und Unicode 30
1.5 Aufbau des Buches 33
1.6 Suchmöglichkeiten 36
1.7 Syntaxdiagramme 36
1.8 Weitere Informationen zum Buch 38

Teil 1 Syntax 39

2 ABAP-Syntax 41

2.1 ABAP-Anweisungen 41
2.2 ABAP-Sprachelemente 41
2.2.1 Operatoren 42
2.2.2 Operanden 43
2.2.3 Bezeichner für Operanden 43
2.2.4 Operanden statisch oder dynamisch angeben 49
2.2.5 Datenobjekte in Operandenpositionen 50
2.3 Namenskonventionen 54
2.4 Kettensätze 55
2.5 Kommentare 56

Teil 2 Programmaufbau 57

3 Programmeinleitende Anweisungen 59

3.1 Übersicht 59
3.2 Ausführbare Programme 61
3.2.1 Zusätze für die Grundliste des Programms 61
3.2.2 Zusatz für die Nachrichtenklasse 63
3.2.3 Zusatz für die Definition einer logischen Datenbank 63

3.3 Modul-Pools und Subroutinen-Pools 64
3.4 Funktionsgruppen 64
3.5 Class-Pools 65
3.6 Interface-Pools 65
3.7 Typgruppen 66

4 Modularisierungsanweisungen 67

4.1 Übersicht 67
4.2 Prozeduren 68
4.2.1 Methoden 68
4.2.2 Funktionsbausteine 69
4.2.3 Unterprogramme 70
4.3 Dialogmodule 75
4.4 Ereignisblöcke 76
4.4.1 Programmkonstruktor 76
4.4.2 Reporting-Ereignisse 77
4.4.3 Selektionsbildereignisse 83
4.4.4 Listenereignisse 88
4.5 Quelltextmodularisierung 95
4.5.1 Include-Programme 95
4.5.2 Makros 96

5 Eingebaute Typen, Datenobjekte und Funktionen 99

5.1 Übersicht 99
5.2 Eingebaute Datentypen 99
5.2.1 Eingebaute ABAP-Typen 99
5.2.2 Generische ABAP-Typen 103
5.2.3 Eingebaute Typen im ABAP Dictionary 104
5.2.4 Der eingebaute Datentyp cursor 108
5.3 Eingebaute Datenobjekte 108
5.3.1 Die Konstante space 108
5.3.2 Die Selbstreferenz me 108
5.3.3 Systemfelder 108
5.3.4 Die Struktur screen 113
5.4 Eingebaute Funktionen 114
5.4.1 Mathematische Funktionen 114
5.4.2 Beschreibungsfunktionen 116

Teil 3 Deklarative Anweisungen 119

6 Deklarative Anweisungen für Datentypen und Datenobjekte 121

6.1 Übersicht 121
6.1.1 Gültigkeit und Sichtbarkeit 121
6.1.2 Datentypen 122
6.1.3 Datenobjekte 122
6.1.4 Absolute Typnamen 124
6.1.5 Anweisungen zur Deklaration von Datentypen und -objekten 127

6.2 Typgruppen einbinden 127

6.3 Datentypen definieren 128
6.3.1 Typdefinitionen mit eingebauten ABAP-Typen 129
6.3.2 Typdefinition durch Bezug auf vorhandene Typen 130
6.3.3 Definition von Referenztypen 130
6.3.4 Definition von strukturierten Typen 132
6.3.5 Definition von Tabellentypen 134
6.3.6 Definition eines Ranges-Tabellentyps 137

6.4 Variablen deklarieren 138
6.4.1 Zusätze für Datenobjekte 139
6.4.2 Elementare Datenobjekte eingebauter ABAP-Typen 141
6.4.3 Datenobjekte bereits vorhandener Typen 141
6.4.4 Deklaration von Referenzvariablen 142
6.4.5 Deklaration von Strukturen 143
6.4.6 Deklaration interner Tabellen 144
6.4.7 Definition einer Ranges-Tabelle 146

6.5 Statische Attribute von Klassen deklarieren 147

6.6 Konstante Datenobjekte deklarieren 148

6.7 Statische Datenobjekte in Prozeduren deklarieren 149

6.8 Strukturkomponenten übernehmen 150

6.9 Tabellenarbeitsbereiche deklarieren 152
6.9.1 Flacher Tabellenarbeitsbereich 152
6.9.2 Beliebiger Tabellenarbeitsbereich 153

6.10 Feldsymbole deklarieren 156
6.10.1 Feldsymbol typisieren 157
6.10.2 Aufprägung einer Struktur 157

6.11 Extraktdatenbestand deklarieren 158

7 Definition von Klassen und Interfaces 161

7.1 Übersicht 161

7.2 Definition von Klassen 162
7.2.1 Deklarationsteil 162
7.2.2 Implementierungsteil 169
7.2.3 Klassen bekannt machen 171

7.3 Definition von Interfaces 172
7.3.1 Deklarationsteil 172
7.3.2 Interfaces bekannt machen 174

7.4 Deklaration von Komponenten in Klassen und Interfaces 174
7.4.1 Methoden 174
7.4.2 Ereignisse 193
7.4.3 Interfaces implementieren bzw. einbinden 196

8 Typisierung 203

8.1 Übersicht 203

8.2 Syntax der Typisierung 203
8.2.1 Generische Typisierung 204
8.2.2 Vollständige Typisierung 205

8.3 Typisierung überprüfen 206
8.3.1 Allgemeine Regeln 206
8.3.2 Literale als Aktualparameter 206

Teil 4 Objekte erzeugen 211

9 Datenobjekte und Objekte erzeugen 213

9.1 Übersicht 213

9.2 Datenobjekte erzeugen 213
9.2.1 Datentyp implizit festlegen 214
9.2.2 Datentyp über eingebaute ABAP-Typen festlegen 215
9.2.3 Datentyp über vorhandenen Typ festlegen 217
9.2.4 Referenzvariablen erzeugen 219
9.2.5 Interne Tabellen erzeugen 220
9.2.6 Behandelbare Ausnahmen beim Erzeugen von Datenobjekten 221

9.3 Objekte in ABAP Objects erzeugen 222
9.3.1 Klasse implizit festlegen 223
9.3.2 Klasse explizit festlegen 223
9.3.3 Statische Parameterübergabe 224
9.3.4 Dynamische Parameterübergabe 225
9.3.5 Behandelbare Ausnahmen beim Erzeugen von Objekten 226

Teil 5 Programmeinheiten aufrufen und verlassen 229

10 ABAP-Programme aufrufen 231

10.1 Übersicht 231

10.2 Ausführbare Programme aufrufen 232
10.2.1 Ablauf des Programmaufrufs 233
10.2.2 Zusätze für das Selektionsbild 235
10.2.3 Zusätze für die Grundliste 244
10.2.4 Zusätze für die Hintergrundverarbeitung 248

10.3 Transaktionen aufrufen 251
10.3.1 Aufruf einer Transaktion und Rückkehr zum Aufrufer 251
10.3.2 Aufruf einer Transaktion ohne Rückkehr zum Aufrufer 258

11 Verarbeitungsblöcke aufrufen 261

11.1 Übersicht 261

11.2 Prozeduren aufrufen 262
11.2.1 Übersicht 262
11.2.2 Methodenaufruf 265
11.2.3 Funktionsbausteinaufruf 277
11.2.4 Unterprogrammaufruf 286

11.3 Ereignisbehandler aufrufen 292
11.3.1 Ereignisse auslösen 292
11.3.2 Ereignisbehandler registrieren 293

11.4 Ereignisblöcke aufrufen 297
11.4.1 Ereignisse einer logischen Datenbank auslösen 297
11.4.2 Listenereignisse auslösen 299

12 Programmeinheiten verlassen 301

12.1 Übersicht 301

12.2 Programme verlassen 301

12.3 Verarbeitungsblöcke verlassen 302
12.3.1 Schema zum Verlassen von Verarbeitungsblöcken 302
12.3.2 Verarbeitungsblöcke mit RETURN verlassen 303
12.3.3 Verarbeitungsblöcke mit EXIT verlassen 304
12.3.4 Verarbeitungsblöcke mit CHECK verlassen 305
12.3.5 GET-Verarbeitungsblöcke mit REJECT verlassen 306
12.3.6 Verarbeitungsblöcke ausführbarer Programme mit STOP verlassen 308

12.4 Schleifen verlassen 309
12.4.1 Schleifen mit EXIT verlassen 309
12.4.2 Schleifendurchlauf mit CONTINUE verlassen 309
12.4.3 Schleifendurchlauf mit CHECK verlassen 310

Teil 6 Programmablaufsteuerung 311

13 Logische Ausdrücke 313

13.1 Übersicht 313

13.2 Logische Ausdrücke mit Vergleichsoperatoren 313
13.2.1 Vergleichsoperatoren für alle Datentypen 313
13.2.2 Vergleichsoperatoren für zeichenartige Datenobjekte 318
13.2.3 Vergleichsoperatoren für byteartige Datentypen 320
13.2.4 Vergleichsoperatoren für Bit-Muster 321

13.3 Intervallzugehörigkeit feststellen 322

13.4 Zustände überprüfen 323
13.4.1 Zuweisung an ein Feldsymbol überprüfen 323
13.4.2 Gültigkeit einer Referenz überprüfen 324
13.4.3 Datenobjekt auf Initialwert überprüfen 325
13.4.4 Ausgabeparameter auf Aktualparameter überprüfen 325
13.4.5 Formalparameter auf Aktualparameter überprüfen 326

13.5 Selektionstabelle auswerten 327

13.6 Boolesche Operatoren und Klammerung 330
13.6.1 Verknüpfung logischer Ausdrücke mit AND 330
13.6.2 Verknüpfung logischer Ausdrücke mit OR 330
13.6.3 Negierung eines logischen Ausdrucks mit NOT 330
13.6.4 Klammerung logischer Ausdrücke 331

14 Kontrollstrukturen 333

14.1 Übersicht 333

14.2 Verzweigungen 333
14.2.1 Verzweigung mit IF 334
14.2.2 Verzweigung mit CASE 335

14.3 Schleifen 336
14.3.1 Unbedingte Schleifen mit DO 336
14.3.2 Bedingte Schleifen mit WHILE 339

15 Ausnahmebehandlung 341

15.1 Übersicht 341
15.1.1 Behandelbare Ausnahmen vor Release 6.10 341
15.1.2 Behandelbare Ausnahmen seit Release 6.10 341
15.1.3 Anweisungen für Ausnahmen 342

15.2 Klassenbasierte Ausnahmen 342
15.2.1 Ausnahmeklassen 342
15.2.2 Klassenbasierte Ausnahmen auslösen 343

15.2.3 Systemverhalten nach einer klassenbasierten Ausnahme 344
15.2.4 Klassenbasierte Ausnahmen behandeln 346

15.3 Nicht-klassenbasierte Ausnahmen 349
15.3.1 Nicht-klassenbasierte Ausnahmen definieren 349
15.3.2 Nicht-klassenbasierte Ausnahmen auslösen 349
15.3.3 Nicht-klassenbasierte Ausnahmen behandeln 350

15.4 Abfangbare Laufzeitfehler 351
15.4.1 Definition abfangbarer Laufzeitfehler 351
15.4.2 Auslösen abfangbarer Laufzeitfehler 351
15.4.3 Abfangbare Laufzeitfehler behandeln 351

Teil 7 Zuweisungen 355

16 Wertzuweisungen 357

16.1 Übersicht 357

16.2 Zuweisung von Datenobjekten 358
16.2.1 Zuweisung mit Schlüsselwort 358
16.2.2 Zuweisung mit Zuweisungsoperator 358
16.2.3 Mehrfachzuweisungen 359

16.3 Zuweisung von Strukturkomponenten 360

16.4 Formatierte Zuweisung 361

16.5 Konvertierung einer gepackten Zahl 363

17 Referenzen setzen 365

17.1 Übersicht 365

17.2 Datenobjekte Feldsymbolen zuweisen 365
17.2.1 Angabe des Speicherbereichs 366
17.2.2 Angabe des Datentyps 372
17.2.3 Angabe der Bereichsgrenzen 375

17.3 Feldsymbol initialisieren 379

17.4 Datenreferenz besorgen 379

18 Datenobjekte initialisieren 381

18.1 Übersicht 381

18.2 Beliebige Datenobjekte initialisieren 381
18.2.1 Mit Initialwert initialisieren 381
18.2.2 Mit anderen Inhalten initialisieren 382

18.3 Interne Tabelle initialisieren 383

18.4 Speicher freigeben 384

Teil 8 Interne Daten bearbeiten 385

19 Rechenausdrücke 387

19.1 Übersicht 387

19.2 Die Anweisung COMPUTE 387

19.3 Arithmetische Ausdrücke 387
19.3.1 Arithmetische Operatoren 388
19.3.2 Klammerung 389
19.3.3 Priorität von funktionalen Methoden 389
19.3.4 Rechentyp 390
19.3.5 Behandelbare Ausnahmen in arithmetischen Ausdrücken 392

19.4 Bit-Ausdrücke 392
19.4.1 Bit-Operatoren 393
19.4.2 Klammerung 394

20 Rechenanweisungen 395

20.1 Übersicht 395

20.2 Addition 395

20.3 Subtraktion 396

20.4 Multiplikation 396

20.5 Division 397

21 Byte- und Zeichenkettenverarbeitung 399

21.1 Übersicht 399
21.1.1 Byte- und Zeichenketten 399
21.1.2 Anweisungen zur Byte- und Zeichenkettenverarbeitung 399
21.1.3 Operanden in der Byte- und Zeichenkettenverarbeitung 400
21.1.4 Behandlung schließender Leerzeichen in der Zeichenkettenverarbeitung 401

21.2 Verketten 402

21.3 Zerlegen 403
21.3.1 In Einzelfelder zerlegen 404
21.3.2 In Zeilen einer internen Tabelle zerlegen 404

21.4 Verschieben 405
21.4.1 Anzahl der Stellen 406
21.4.2 Richtung der Verschiebung 407
21.4.3 Verschieben eines Musters aus dem Feld 407

21.5 Verdichten 409

21.6 Konvertieren 410

21.7 Durchsuchen 411
21.7.1 Durchsuchen mit FIND 411
21.7.2 Durchsuchen mit SEARCH 414
21.8 Ersetzen 418
21.8.1 Positionsbasiertes Ersetzen 418
21.8.2 Musterbasiertes Ersetzen 420
21.9 Bits setzen und lesen 423
21.9.1 Einzelbit setzen 423
21.9.2 Einzelbit lesen 425
21.10 Überlagern 426
21.11 Umsetzen 427
21.11.1 Behandlung der Schreibweise 427
21.11.2 Beschreibung der Substitution 428

22 Interne Tabellen bearbeiten 429
22.1 Übersicht 429
22.2 Interne Tabellen auslesen 430
22.2.1 Einzelne Zeilen lesen 430
22.2.2 Schleifenverarbeitung interner Tabellen 441
22.2.3 Gruppenstufenverarbeitung 445
22.2.4 Summierung in Gruppenstufen 447
22.3 Interne Tabellen füllen 449
22.3.1 Zeilen einfügen 449
22.3.2 Zeilen verdichtet einfügen 454
22.3.3 Zeilen anhängen 456
22.4 Tabellenzeilen bearbeiten 460
22.4.1 Zeilen ändern 460
22.4.2 Zeilen löschen 466
22.4.3 Sortieren 472
22.5 Angabe von Komponenten 475
22.6 Eigenschaften interner Tabellen bestimmen 476
22.7 Interne Tabellen durchsuchen 478
22.7.1 Suchoptionen 479
22.8 Verarbeitung spezieller interner Tabellen 480
22.9 Tabelle editieren 485

23 Extraktdatenbestände bearbeiten 487
23.1 Einführung 487
23.2 Zeilenstruktur festlegen 487
23.3 Extraktdatenbestand füllen 489

23.4 Extraktdatenbestand sortieren 490
23.5 Extraktdatenbestand auslesen 492
23.6 Gruppenstufenverarbeitung 493

24 Eigenschaften von Datenobjekten 497

24.1 Übersicht 497
24.2 Eigenschaften beliebiger Datenobjekte 497
24.2.1 Länge des Datenobjekts 498
24.2.2 Ausgabelänge 498
24.2.3 Datentyp und Komponentenanzahl 500
24.2.4 Nachkommastellen 501
24.2.5 Konvertierungsroutine 501
24.2.6 Hilfetext 501
24.3 Eigenschaften interner Tabellen 503
24.3.1 Tabellenart 504
24.3.2 Initialer Speicherbedarf 504
24.3.3 Anzahl der Zeilen 504
24.4 Abstände von Datenobjekten 505

Teil 9 Benutzerdialoge 509

25 Dynpros 511

25.1 Übersicht 511
25.2 Anweisungen der Dynpro-Ablauflogik 513
25.2.1 Ereignisblöcke der Dynpro-Ablauflogik 513
25.2.2 Aufruf von Dialogmodulen 515
25.2.3 Datenübergabe steuern 517
25.2.4 Verarbeitungsketten 521
25.2.5 Table Controls 522
25.2.6 Aufruf eines Subscreens 527
25.3 ABAP-Anweisungen für Dynpros 530
25.3.1 Dynpro-Folge aufrufen 530
25.3.2 Folge-Dynpro setzen 532
25.3.3 GUI-Status setzen 532
25.3.4 GUI-Status feststellen 534
25.3.5 GUI-Titel setzen 536
25.3.6 Eigenschaften von Dynpro-Feldern bestimmen 538
25.3.7 Eigenschaften von Dynpro-Feldern modifizieren 540
25.3.8 Cursor setzen 542
25.3.9 Cursor-Position bestimmen 544
25.3.10 Control deklarieren 546
25.3.11 Control initialisieren 553

25.3.12 Steploop verlassen 553
25.3.13 Feldinhalte bewahren 554
25.3.14 Anzeige unterdrücken 555
25.3.15 Dynpro verlassen 555

26 Selektionsbilder 557

26.1 Übersicht 557
26.1.1 Selektionsbilder als Dynpros 557
26.1.2 Aufgaben von Selektionsbildern 557
26.1.3 Generierung von Selektionsbildern 557
26.1.4 Selektionsbilder und logische Datenbanken 558
26.1.5 Anweisungen für Selektionsbilder 558

26.2 Selektionsbilder anlegen und gestalten 558
26.2.1 Selektionsbilder anlegen 559
26.2.2 Selektionsbilder gestalten 563
26.2.3 Elemente anderer Selektionsbilder übernehmen 577
26.2.4 Varianten und Zusätze für Selektionsbilder logischer Datenbanken 581

26.3 Parameter definieren 586
26.3.1 Datentyp des Parameters 587
26.3.2 Eigenschaften der Bildschirmelemente 590
26.3.3 Eigenschaften des Werts und der Wertübergabe 596
26.3.4 Zusätze für Selektionsbilder logischer Datenbanken 599

26.4 Selektionskriterien definieren 601
26.4.1 Datentyp der Spalten LOW und HIGH der Selektionstabelle 605
26.4.2 Eigenschaften der Bildschirmelemente 607
26.4.3 Eigenschaften des Werts und der Wertübergabe 610
26.4.4 Zusätze für Selektionsbilder logischer Datenbanken 612

26.5 Selektionsbilder aufrufen 614
26.5.1 Aufruf über SUBMIT 614
26.5.2 Aufruf über Reporttransaktion 614
26.5.3 Aufruf über Dialogtransaktion 614
26.5.4 Aufruf im Programm 615
26.5.5 Selektionsbildverarbeitung 616

27 Listen 619

27.1 Übersicht 619
27.1.1 Listen als Bildschirmbilder 619
27.1.2 Listen im ABAP-Programm 619
27.1.3 Grundliste 619
27.1.4 Verzweigungslisten 620
27.1.5 Aufbau einer Liste 620
27.1.6 Drucklisten 621
27.1.7 Listen und ABAP Objects 621
27.1.8 Anweisungen zur Listenverarbeitung 622

27.2 Listen erstellen 623
27.2.1 Daten in Listen schreiben **623**
27.2.2 Horizontale Linien erzeugen **644**
27.2.3 Listen abschnittsweise formatieren **645**
27.2.4 Anzeige von Leerzeilen **652**
27.2.5 Vertikale Positionierung des Listen-Cursors **653**
27.2.6 Horizontale Positionierung des Listen-Cursors **658**
27.2.7 Fixbereich beim horizontalen Blättern **659**
27.2.8 Seitenumbruch und Drucklistenerstellung **660**
27.2.9 Bedingter Seitenumbruch **669**
27.2.10 Variablen mit Listenzeilen speichern **670**
27.2.11 Seitenrand von Drucklisten **671**
27.2.12 Steuerung von Drucklisten **672**

27.3 Listen im Listenpuffer bearbeiten 675
27.3.1 Listenzeilen lesen **675**
27.3.2 Listenzeilen modifizieren **678**
27.3.3 Listen blättern **680**
27.3.4 Listeneigenschaften auslesen **683**

27.4 Angezeigte Liste an Cursor-Position auswerten 686

27.5 Anzeigeeigenschaften von Bildschirmlisten 688
27.5.1 GUI-Status einer Bildschirmliste **688**
27.5.2 Titel einer Bildschirmliste **690**
27.5.3 Cursor setzen **690**
27.5.4 Liste im Dialogfenster **692**

27.6 Listenanzeige aufrufen und verlassen 694
27.6.1 Anzeige der Grundliste aufrufen **695**
27.6.2 Anzeige von Listen verlassen **696**

28 Nachrichten 699

28.1 Übersicht 699
28.1.1 Ablage von Nachrichten **699**
28.1.2 Nachrichtentypen **699**

28.2 Nachrichten senden 702
28.2.1 Variante 1 **703**
28.2.2 Variante 2 **703**
28.2.3 Variante 3 **703**
28.2.4 Variante 4 **704**
28.2.5 Zusätze options **705**

Teil 10 Externe Daten bearbeiten 709

29 Open SQL 711

29.1 Übersicht 711
29.1.1 Umfang von Open SQL 711
29.1.2 Datenbankschnittstelle 711
29.1.3 Datenbankzugriff 711
29.1.4 Mandantenbehandlung 711
29.1.5 SAP-Pufferung 711
29.1.6 LUW 712
29.1.7 Die Anweisungen von Open SQL 712

29.2 Daten aus Datenbanktabellen lesen 713
29.2.1 Struktur der Ergebnismenge bestimmen 715
29.2.2 Auszulesende Datenbanktabellen angeben 722
29.2.3 Zielbereich angeben 729
29.2.4 Ergebnismenge einschränken 734
29.2.5 Zeilen zusammenfassen 746
29.2.6 Zusammengefasste Zeilen einschränken 747
29.2.7 Zeilen der Ergebnismenge sortieren 749

29.3 Daten aus Datenbanktabellen über Cursor lesen 751
29.3.1 Cursor öffnen 752
29.3.2 Daten über Cursor lesen 753
29.3.3 Cursor schließen 754

29.4 Daten in Datenbanktabellen einfügen 756
29.4.1 Angabe der Datenbanktabelle 757
29.4.2 Angabe der Quelle 758

29.5 Daten in Datenbanktabellen ändern 760
29.5.1 Angabe der Datenbanktabelle 761
29.5.2 Angabe der Änderungen 762

29.6 Daten in Datenbanktabellen einfügen oder ändern 768
29.6.1 Angabe der Datenbanktabelle 768
29.6.2 Angabe der Quelle 769

29.7 Daten in Datenbanktabellen löschen 770
29.7.1 Angabe der Datenbanktabelle 771
29.7.2 Angabe der Zeilen 771

29.8 Arbeitsbereiche in Open-SQL-Anweisungen 774

29.9 Behandelbare Ausnahmen in Open-SQL-Anweisungen 775

30 Native SQL 777

30.1 Übersicht 777

30.2 Native SQL einbinden 777
30.2.1 Hostvariablen 779
30.2.2 Cursor-Verarbeitung 781

30.2.3 Datenbankprozeduren aufrufen 782
30.2.4 Datenbankverbindung festlegen 784
30.2.5 Implizite Cursor-Verarbeitung 787

30.3 Native SQL verlassen 788

30.4 Behandelbare Ausnahmen in Native SQL 789

31 Daten-Cluster 791

31.1 Übersicht 791

31.2 Daten-Cluster erstellen 792
31.2.1 Definition des Daten-Clusters 792
31.2.2 Bestimmung der Ablage 793
31.2.3 Komprimierung steuern 799
31.2.4 Behandelbare Ausnahmen beim Export von Daten-Clustern 799

31.3 Daten-Cluster lesen 799
31.3.1 Angabe der einzulesenden Datenobjekte 800
31.3.2 Bestimmung der Ablage 801
31.3.3 Konvertierungszusätze 805
31.3.4 Behandelbare Ausnahmen beim Importieren aus Daten-Clustern 812

31.4 Inhaltsverzeichnis eines Daten-Clusters lesen 812

31.5 Löschen eines Daten-Clusters 814

31.6 Löschen eines Daten-Clusters im ABAP Memory 815

32 Die ABAP-Dateischnittstelle 817

32.1 Übersicht 817
32.1.1 Adressierung von Dateien 817
32.1.2 Berechtigungen für Dateizugriffe 817
32.1.3 Sperren 820
32.1.4 Die Dateischnittstelle in Unicode-Programmen 820
32.1.5 Dateigröße 821
32.1.6 Die Anweisungen der Dateischnittstelle 821

32.2 Datei öffnen 822
32.2.1 Definition der Zugriffsart 823
32.2.2 Definition der Ablageart 824
32.2.3 Positionsangabe 829
32.2.4 Betriebssystemabhängige Zusätze 830
32.2.5 Fehlerbehandlung 832
32.2.6 Behandelbare Ausnahmen beim Öffnen von Dateien 834

32.3 Datei schreiben 834
32.3.1 Einfluss der Zugriffsart 835
32.3.2 Einfluss der Ablageart 836
32.3.3 Anzahl der übertragenen Zeichen bzw. Bytes einschränken 837
32.3.4 Behandelbare Ausnahmen beim Schreiben in Dateien 838

32.4 Datei lesen 838
32.4.1 Einfluss der Zugriffsart 839
32.4.2 Einfluss der Ablageart 839
32.4.3 Anzahl der eingelesenen Zeichen bzw. Bytes einschränken 841
32.4.4 Anzahl der eingelesenen Zeichen bzw. Bytes feststellen 842
32.4.5 Behandelbare Ausnahmen beim Lesen von Dateien 843

32.5 Eigenschaften einer geöffneten Datei bestimmen 843
32.5.1 Position des Dateizeigers feststellen 844
32.5.2 Weitere Eigenschaften feststellen 845
32.5.3 Behandelbare Ausnahmen beim Feststellen von Dateieigenschaften 847

32.6 Dateieigenschaften einer geöffneten Datei ändern 847
32.6.1 Position des Dateizeigers festlegen 848
32.6.2 Weitere Eigenschaften ändern 849
32.6.3 Behandelbare Ausnahmen beim Ändern von Dateieigenschaften 851

32.7 Datei schließen 852

32.8 Datei löschen 853

33 Datenkonsistenz 855

33.1 Übersicht 855

33.2 Datenbank-LUW 856
33.2.1 Datenbank-Commit 856
33.2.2 Datenbank-Rollback 857

33.3 SAP-LUW 858
33.3.1 SAP-Commit 859
33.3.2 SAP-Rollback 861
33.3.3 Lokale Verbuchung 862

33.4 Datenbanksperren 863

33.5 SAP-Sperren 863
33.5.1 SAP-Sperren verhängen 863
33.5.2 SAP-Sperren aufheben 864

33.6 Berechtigungsprüfung 865

Teil 11 Programmparameter 869

34 Parameter im SAP Memory 871

34.1 Übersicht 871

34.2 Parameter setzen 871

34.3 Parameter lesen 872

35 Sprachumgebung 873

35.1 Übersicht 873

35.2 Sprache von Textelementen setzen 873

35.3 Textumgebung setzen 874

35.4 Textumgebung feststellen 876

35.5 Länderkennung setzen 877

36 Zeitstempel 879

36.1 Übersicht 879

36.2 Aktueller Zeitstempel 880

36.3 Zeitstempel bearbeiten 881
36.3.1 Zeitstempel-Auflösung 881
36.3.2 Zeitstempel-Erstellung 881
36.3.3 Erklärung des Zusatzes option 882

36.4 Aktuelle Uhrzeit 884

Teil 12 Programmbearbeitung 885

37 Programme testen und prüfen 887

37.1 Übersicht 887

37.2 Haltepunkte setzen 887

37.3 Laufzeitmessung 888
37.3.1 Relative Programmlaufzeit 888
37.3.2 Zeitauflösung festlegen 889
37.3.3 Messstrecke für Laufzeitanalyse 890

37.4 Erweiterte Programmprüfung umgehen 891

38 Dynamische Programmerstellung 893

38.1 Übersicht 893

38.2 Dynamischer Subroutinen-Pool 894

38.3 Einlesen eines ABAP-Programms 897

38.4 Syntaxüberprüfung 898

38.5 Anlegen eines ABAP-Programms 900

38.6 Einlesen eines Text-Pools 903

38.7 Anlegen eines Text-Pools 904

38.8 ABAP Editor aufrufen 906

Teil 13 Externe Programmierschnittstellen 907

39 Remote Function Call 909

39.1 Übersicht 909
39.1.1 Einleitung 909
39.1.2 Destination 910
39.1.3 Systemfelder beim RFC 911
39.1.4 Ausnahmen beim RFC 911
39.1.5 RFC-Berechtigung 912
39.1.6 Unzulässige Aktionen 912
39.1.7 Anweisungen der RFC-Schnittstelle 912

39.2 Remote Funktionsaufruf 913
39.2.1 Synchroner Remote Function Call 913
39.2.2 Asynchroner Remote Function Call 914
39.2.3 Transaktionaler Remote Function Call 921

40 XSLT-Transformationen 923

40.1 Übersicht 923

40.2 Aufruf 923
40.2.1 Angabe der Transformation 924
40.2.2 Transformationsquelle 924
40.2.3 Transformationsergebnis 925
40.2.4 Erklärung der Zusätze options 926

Teil 14 Obsolete Anweisungen 929

41 Obsolete Anweisungen 931

41.1 Übersicht 931

41.2 Obsolete Syntax 931

41.3 Obsolete Modularisierung 932
41.3.1 Obsoleter Ereignisblock 932

41.4 Obsolete Deklarationen 932
41.4.1 Obsolete Schnittstellen-Arbeitsbereiche 932
41.4.2 Obsolete Deklarationen interner Standardtabellen 935
41.4.3 Obsolete Deklarationen spezieller interner Tabellen 937
41.4.4 Obsoleter Hinweis für die erweiterte Programmprüfung 940

41.5 Obsolete Objekterzeugung 940

41.6 Obsoleter Programmaufruf 943

41.7 Obsoletes Verlassen eines Programms 945

41.8 Obsolete Programmablaufsteuerung 946
41.8.1 Obsolete Vergleichsoperatoren **946**
41.8.2 Obsolete Verzweigung **946**

41.9 Obsolete Zuweisungen 948
41.9.1 Obsolete Zuweisung eines prozentualen Teilfeldes **948**
41.9.2 Obsolete Konvertierung **949**
41.9.3 Obsoletes Zwischenspeichern von Datenobjekten **950**

41.10 Obsolete Rechenanweisungen 951
41.10.1 Addition von Feldfolgen im Speicher **951**
41.10.2 Komponentenweise addieren **953**
41.10.3 Komponentenweise subtrahieren **954**
41.10.4 Komponentenweise multiplizieren **955**
41.10.5 Komponentenweise dividieren **956**
41.10.6 Obsolete Berechnungen während der Listenerstellung **957**

41.11 Obsolete Zeichenkettenverarbeitung 960
41.11.1 Obsolete Umsetzung **960**
41.11.2 Obsoletes Ersetzen **962**
41.11.3 Neunerkomplement eines Datums **963**

41.12 Obsolete Verarbeitung interner Tabellen 965
41.12.1 Obsolete Schlüsselangaben beim Lesen von Zeilen **965**
41.12.2 Obsolete Zuweisung an Tabellenzeilen **968**
41.12.3 Obsolete Form der Anweisung PROVIDE **970**

41.13 Contexte 972
41.13.1 Übersicht **972**
41.13.2 Instanzen von Contexten erzeugen **973**
41.13.3 Contexte mit Schlüsselwerten versorgen **974**
41.13.4 Contexte abfragen **975**

41.14 Obsolete Anweisungen der Dynpro-Ablauflogik 977
41.14.1 Werteüberprüfung in der Ablauflogik **977**
41.14.2 Verarbeitung von Steploops **979**

41.15 Obsolete Anweisungen der Listenverarbeitung 985
41.15.1 Obsolete Formatierungsanweisungen **985**
41.15.2 Obsolete Druckparameter **986**

41.16 Obsolete Datenbankzugriffe 988
41.16.1 Obsoletes Lesen einer Zeile **988**
41.16.2 Obsoletes sequenzielles Lesen mehrerer Zeilen **991**
41.16.3 Obsoletes Lesen mehrerer Zeilen in eine interne Tabelle **992**
41.16.4 Obsolete Kurzformen in Open SQL **994**

41.17 Obsolete externe Programmierschnittstellen 995
41.17.1 CPI-C-Schnittstelle **995**
41.17.2 OLE-Schnittstelle **1002**

Anhang 1013

A Konvertierungsregeln für Zuweisungen 1015

A.1 Übersicht 1015

A.2 Konvertierungsregeln für elementare Datentypen 1015
A.2.1 Darstellung von numerischen Werten in zeichenartigen Feldern 1016
A.2.2 Konvertierungstabelle für Quellfeld Typ c 1017
A.2.3 Konvertierungstabelle für Quellfeld Typ d 1018
A.2.4 Konvertierungstabelle für Quellfeld Typ f 1020
A.2.5 Konvertierungstabelle für Quellfeld Typ i, b oder s 1021
A.2.6 Konvertierungstabelle für Quellfeld Typ n 1022
A.2.7 Konvertierungstabelle für Quellfeld Typ p 1023
A.2.8 Konvertierungstabelle für Quellfeld Typ string 1024
A.2.9 Konvertierungstabelle für Quellfeld Typ t 1025
A.2.10 Konvertierungstabelle für Quellfeld Typ x 1026
A.2.11 Konvertierungstabelle für Quellfeld Typ xstring 1027

A.3 Konvertierungsregeln für Strukturen 1028
A.3.1 Konvertierung zwischen flachen Strukturen 1029
A.3.2 Konvertierung zwischen flachen Strukturen und Einzelfeldern 1032

A.4 Konvertierungsregeln für interne Tabellen 1033

A.5 Zuweisungen zwischen Referenzvariablen 1034
A.5.1 Statischer und dynamischer Typ 1034
A.5.2 Narrowing Cast und Widening Cast 1035
A.5.3 Zuweisungen zwischen Datenreferenzvariablen 1036
A.5.4 Zuweisungen zwischen Objektreferenzvariablen 1037

B Sprachnahe Klassen und Interfaces 1039

B.1 Hilfsklassen 1039
B.1.1 Klassen der Run Time Type Identification 1039
B.1.2 Klassen für Konvertierungen externer Datenformate 1039
B.1.3 Klasse für Extremwerte von Datenobjekten 1040
B.1.4 Klasse für die Eigenschaften von Zeichen 1040
B.1.5 Klasse für mathematische Operationen 1040
B.1.6 Klasse für Zeitstempel 1041
B.1.7 Klassen für Daten-Cluster 1041
B.1.8 Klasse für Transaktionen 1042

B.2 Object Services 1042

B.3 JavaScript-Integration 1042

C Sprachnahe Funktionsbausteine 1045

C.1 Funktionsbausteine für Druckparameter 1045
C.1.1 GET_PRINT_PARAMETERS 1045
C.1.2 SET_PRINT_PARAMETERS 1048
C.2 Funktionsbausteine für Dateien auf dem Präsentationsserver 1048
C.3 Funktionsbaustein für den Aufruf logischer Datenbanken 1048

D Vordefinierte behandelbare Ausnahmen 1051

D.1 Übersicht 1051
D.2 Vordefinierte Ausnahmeklassen 1051
D.3 Abfangbare Laufzeitfehler 1054
D.3.1 Ausnahmegruppe für arithmetische Fehler 1054
D.3.2 Ausnahmegruppe für Konvertierungsfehler 1055
D.3.3 Ausnahmegruppe für Fehler bei der Erzeugung von Datenobjekten 1056
D.3.4 Ausnahmegruppe für Fehler bei der Erzeugung von Instanzen von Klassen 1056
D.3.5 Ausnahmegruppe für Fehler beim Zugriff auf Datenobjekte 1056
D.3.6 Ausnahmegruppe für Fehler beim dynamischen Methodenaufruf 1057
D.3.7 Ausnahmegruppe für Fehler beim Dateizugriff 1058
D.3.8 Ausnahmegruppe für Fehler beim Zugriff auf Daten-Cluster 1058
D.3.9 Ausnahmegruppe für Fehler in der Sprachumgebung 1059
D.3.10 Ausnahmegruppe für Fehler beim Remote Function Call 1059
D.3.11 Nicht zugeordnete abfangbare Laufzeitfehler 1059

E Hinweise zu den CD-ROMs 1061

F Glossar 1065

Index 1113

Vorwort

Gleich nachdem die Programmiersprache ABAP mit der Einführung von ABAP Objects zu Release 4.6 einen großen Schritt in Richtung zeitgemäße Programmierung vollzogen hatte, stellte sich für die Weiterentwicklung der Sprache bereits die nächste Herausforderung: Um den Anforderungen eines vom Internet bestimmten, internationalen Programmierumfeldes gerecht zu werden, musste der Web Application Server als Nachfolger der SAP-Basis und mit ihm die Programmiersprache ABAP unicodefähig gemacht werden. Diese Anforderung wurde mit Release 6.10 erfüllt.

Mit der Implementierung von Release 6.10 bzw. 6.20 wird jedoch niemand gezwungen, seine Programme anzupassen, solange das SAP-System nicht auf Unicode umgestellt wird. Unabhängig davon, ob ein Programm in einem Unicode- oder einem Nicht-Unicode-System eingesetzt werden soll, bedeutet die Verwendung der im Zusammenhang mit Unicode eingeführten Neuerungen dennoch einen wichtigen Schritt in Richtung sicherer und fehlerfreier Programmierung.

Daher war vor allem die Einführung von Unicode ein wichtiger Beweggrund für dieses Buch. Die im System vorhandene ABAP-Schlüsselwortdokumentation wuchs mit der Sprache, indem Neues hinzugefügt und Altes ungeändert belassen wurde. Es gibt dort beispielsweise ein Teilgebiet »ABAP Objects« und seit Release 6.10 eben auch ein Teilgebiet »Unicode«, während die Anweisungen des klassischen Reporting nach wie vor so beschrieben sind, als wäre jedes ABAP-Programm ein Report, der mit einer logischen Datenbank verknüpft ist. Da diese Art der Dokumentation als Direkthilfe zwar vertretbar, für eine einheitliche Gesamtbeschreibung aber nicht mehr ausreichend ist, haben wir die Gelegenheit ergriffen, der Einführung in die SAP-Programmierung mit ABAP Objects eine ABAP-Referenz folgen zu lassen.

Das Ziel, das wir uns gesteckt haben, ist die vollständige Neufassung und Vereinheitlichung der ABAP-Dokumentation. Dies betrifft nicht nur die Gliederung und Zusammenfassung der einzelnen Anweisungen, sondern auch die inhaltliche Aufbereitung und stilistische Anpassung der Beschreibung. Speziell für die Darstellung der Syntaxdiagramme wurde daher eine neue Sichtweise in Form einer Pseudosyntax eingeführt, die jede Anweisung ausgehend von ihrer Grundform schrittweise in ihre einzelnen Bestandteile auflöst. Erstmals werden auch die Anforderungen an alle in einer ABAP-Anweisung vorkommenden Operanden vollständig und in einer einheitlichen Form beschrieben. Unser Anspruch ist die umfassende

Beschreibung aller im aktuellen Release 6.20 zur Verwendung freigegebenen ABAP-Sprachelemente. Dies beinhaltet auch eine detaillierte Beschreibung aller zwar als obsolet gekennzeichneten, aber nur in ABAP Objects verbotenen Sprachelemente. Die ABAP-Referenz soll den Leser in die Lage versetzen, jedes ABAP-Programm zu analysieren und neue Programme zu schreiben. Sie kann und will zwar kein Best-Practices-Buch sein, wir hoffen aber, dass Sinn und Zweck mancher Anweisung durch die genaue Beschreibung klarer wird, als es bisher der Fall war.

Es ist leichter gesagt als getan, eine Sprache mit etwa 500 wesentlichen Sprachelementen durchgehend auf gleich hohem Niveau und in gleichem Detaillierungsgrad zu beschreiben, zumal ABAP seit zwanzig Jahren ständig weiterentwickelt wird und aus Kompatibilitätsgründen keine Entwicklung jemals wieder zurückgenommen wurde. Dies erklärt auch die wiederholten Verschiebungen des Erscheinungstermins dieses Buches und wir danken allen, die so lange gewartet haben. Als Entschädigung halten Sie jetzt aber ein Buch in Händen, das das aktuellste Release der Sprache ABAP beschreibt und lange gültig bleiben wird. Da wir jedoch trotz sorgfältiger Durchsicht aller Kapitel nicht ausschließen können, dass dieses Buch noch kleinere Unstimmigkeiten enthält, freuen wir uns auf alle diesbezüglichen Anregungen und Hinweise. Sie können sie auf den Internetseiten von SAP PRESS unter *www.sap-press.de* einreichen.

Ohne die mittelbare und unmittelbare Hilfe vieler Personen bei der Erstellung und Prüfung des Manuskripts wäre das Buch in der vorliegenden Form nicht zustande gekommen. An dieser Stelle gilt der Dank den Kollegen Masoud Aghadavoodi, Thomas Bareiss, Adrian Goerler, Christian Jendel, Gerd Kluger, Björn Mielenhausen, Andreas Simon Schmitt und Christoph Stöck. Erhardt Vortanz danken wir für seine Mitarbeit bei der Erstellung des Manuskripts. Dem Development Manager der Gruppe Business Programming Languages, Andreas Blumenthal, danken wir, dass er uns dieses Projekt überhaupt ermöglichte und weitgehend freie Hand in der Gestaltung ließ. Dem unermüdlichen Michael Demuth ist es zu verdanken, dass diesem Buch wieder zwei CDs mit einem SAP-System, dieses Mal in Form eines Web Application Server 6.10, beiliegen. Dank gilt schließlich auch allen Mitarbeitern von Galileo Press – insbesondere Iris Warkus und Florian Zimniak – für die gute Zusammenarbeit bei der Korrektur des Manuskripts und dafür, dass sie die Hoffnung nie aufgaben, dass wir unser Manuskript irgendwann überhaupt noch einmal einreichen würden.

Horst Keller dankt ganz besonders seiner Frau Ute, die ihn in den letzten Monaten fast nur noch über dem aufgeklappten Laptop zu Gesicht bekam, für ihre Geduld und ihr Verständnis dafür, dass ein großer Anteil der ohnehin spärlichen gemeinsamen Freizeit schon wieder für ein Buchprojekt geopfert wurde.

Wir wünschen nun allen Leserinnen und Lesern viel Spaß beim Durchstöbern dieser ABAP-Referenz.

Walldorf, im Juli 2002
Horst Keller, Joachim Jacobitz

1 Einführung und Übersicht

1.1 Ziel des Buches

Das vorliegende Buch enthält eine vollständige Beschreibung der Programmiersprache ABAP, die seit 1982 bei der SAP AG entwickelt und in den meisten SAP-Anwendungsprogrammen eingesetzt wird. Es behandelt Aufbau und Wirkungsweise aller zur Verwendung freigegebenen ABAP-Anweisungen in detaillierter Form und ist damit ein umfassendes Nachschlagewerk für alle Release-Stände bis einschließlich 6.20.

Das Buch ist als Arbeitsbuch für den täglichen Einsatz jedes ABAP-Entwicklers gedacht. Für eine Einführung in die Programmiersprache ABAP verweisen wir auf das Buch »ABAP Objects – Einführung in die SAP-Programmierung« von Horst Keller und Sascha Krüger, das ebenfalls bei SAP PRESS erschienen ist.

1.2 Beschriebene Releases

Wir beschreiben den Stand der Sprache ABAP zu Release 6.20. Neuerungen, die sich von Release 4.6 zu Release 6.10 und 6.20 ergeben haben, sind in der Randspalte mit Ikonen gekennzeichnet, so dass das Buch uneingeschränkt auch für Release 4.6 verwendbar ist. Folgende wesentliche Änderungen haben sich von Release 4.6 nach 6.10 ergeben:

- ABAP-Programme können unicode-fähig gemacht werden.
- Es wurde ein neues, klassenbasiertes Ausnahmekonzept eingeführt.
- Es gibt neue Sprachelemente für die dynamische Programmierung, wie z.B. neue generische Datentypen, typisierte Datenreferenzen und die Möglichkeit, Open-SQL-Anweisungen weitgehend dynamisch anzugeben.

Selbstverständlich werden mit der Beschreibung von Release 6.20 bzw. Release 4.6 auch alle vorhergehenden Releases abgedeckt.[1] Da Release 4.6 aber der inzwischen verbreitete Standard ist, wird bei den Sprachelementen, die vor Release 6.10 eingeführt wurden, nicht vermerkt, ab welchem Release sie gültig sind. Hierfür sei auf die ABAP-Schlüsselwortdokumentation verwiesen, in der dies unter dem Knoten **ABAP · Releaseabhängige Änderungen** seit Release 3.0 dokumentiert ist.

1 Manches betagte Sprachelement wurde sogar bisher noch nie so detailliert wie hier beschrieben.

1.3 SAP-Basis und Web Application Server

Zu Release 6.10 wurde die SAP-Basis zum Web Application Server weiterentwickelt. Der Web Application Server enthält die Funktionalität der SAP-Basis und damit die ABAP-Laufzeitumgebung. Unter anderem unterscheiden folgende Komponenten den Web Application Server von der alten SAP-Basis:

- Der Web Application Server unterstützt HTTP(S) und kann damit direkt, d.h. ohne eine Zwischenstufe wie sie beispielsweise der Internet Transaction Server (ITS) darstellt, als Webserver eingesetzt werden.
- Für die Programmierung eines Webservers wurden Business Server Pages (BSP) eingeführt. Business Server Pages sind HTML-Seiten, die ein Server Side Scripting mit ABAP und JavaScript als Script-Sprachen erlauben. Durch die Einbindung von ABAP als Script-Sprache hat man in Business Server Pages direkten Zugriff auf den gesamten Sprachumfang von ABAP und damit insbesondere über Open SQL Zugriff auf die Datenbank des Web Application Server.
- Der Web Application Server enthält einen XSLT-Prozessor, der die Umwandlung von mit XML beschriebenen Daten in ABAP-Datentypen und umgekehrt ermöglicht.
- Der Web Application Server enthält einen JavaScript-Prozessor, der es ermöglicht, in JavaScript geschriebene Programme auszuführen. Variablen des Scripts können an Datenobjekte eines ABAP-Programms angebunden werden.

In diesem Buch beschreiben wir ausschließlich die Sprache ABAP. Für weitergehende Informationen zum Web Application Server, insbesondere zu Business Server Pages, verweisen wir auf das Buch »SAP Web Application Server – Entwicklung von Web-Anwendungen« von Frédéric Heinemann und Christian Rau, das ebenfalls bei SAP PRESS erscheint.

1.4 ABAP und Unicode

Vor Release 6.10 wurden von SAP unterschiedliche Codes für die Darstellung von Zeichen verschiedener Schriften verwendet, wie zum Beispiel ASCII, EBCDIC oder Double-Byte-Codepages:

- ASCII (*American Standard Code for Information Interchange*) verschlüsselt jedes Zeichen durch ein Byte. Damit lassen sich höchstens 256 Zeichen darstellen.[2] Gebräuchliche Codepages sind zum Beispiel ISO88591 für westeuropäische oder ISO88595 für kyrillische Schriften.

2 Genaugenommen verschlüsselt Standard-ASCII ein Zeichen nur durch 7 Bit und kann daher nur 128 Zeichen abbilden.

- EBCDIC (*Extended Binary Coded Decimal Interchange*) verschlüsselt ebenfalls jedes Zeichen durch ein Byte, womit sich wiederum 256 Zeichen abbilden lassen. EBCDIC 0697/0500 ist beispielsweise ein IBM-Format, das zum Beispiel auf der Plattform AS400 für westeuropäische Schriften verwendet wird.
- Double-Byte-Codepages benötigen 1 bis 2 Byte je Zeichen. Dadurch lassen sich 65536 Zeichen darstellen, wobei in der Regel nur 10000 bis 15000 Zeichen belegt sind. Als Codepages werden zum Beispiel SJIS für die japanische und BIG5 für die traditionelle chinesische Schrift verwendet.

Mit diesen Zeichensätzen können in einem SAP-System alle Sprachen einzeln abgedeckt werden. Schwierigkeiten ergeben sich, wenn in einem zentralen System Texte aus verschiedenen inkompatiblen Zeichensätzen gemischt werden sollen. Auch der Austausch von Daten zwischen Systemen mit inkompatiblen Zeichensätzen kann zu Problemen führen.

Die Lösung dieses Problems ist die Verwendung eines Zeichensatzes, der alle Zeichen auf einmal umfasst; dies wird durch Unicode (ISO/IEC 10646) verwirklicht. Für den Unicode-Zeichensatz gibt es verschiedene Unicode-Zeichendarstellungen, in denen ein Zeichen zwischen ein und vier Bytes belegen kann.

Der Web Application Server unterstützt ab Release 6.10 sowohl Nicht-Unicode- als auch Unicode-Systeme. Nicht-Unicode-Systeme sind herkömmliche SAP-Systeme, bei denen in der Regel ein Zeichen durch ein Byte repräsentiert wird. Unicode-Systeme sind SAP-Systeme, die auf einer Unicode-Zeichendarstellung basieren und denen ein entsprechendes Betriebssystem samt Datenbank zugrunde liegt.

Vor Release 6.10 gingen viele ABAP-Programmiertechniken davon aus, dass ein Zeichen einem Byte entspricht. Bevor ein System auf Unicode umgestellt wird, müssen ABAP-Programme deshalb überall dort geändert werden, wo eine explizite oder implizite Annahme über die interne Länge eines Zeichens gemacht wird.

ABAP unterstützt diese Umstellung durch neue Syntaxregeln und neue Sprachkonstrukte, wobei größter Wert darauf gelegt wurde, so viel vorhandenen Quelltext wie nur möglich zu erhalten. Als Vorbereitung auf eine Umstellung auf Unicode – aber auch unabhängig davon, ob ein System tatsächlich auf Unicode umgestellt werden soll – kann in den Programmeigenschaften das Ankreuzfeld **Unicodeprüfungen aktiv** markiert werden. Die Transaktion UCCHECK unterstützt das Einschalten dieser

Prüfung für vorhandene Programme. Falls diese Eigenschaft gesetzt ist, wird das Programm als Unicode-Programm bezeichnet. In einem Unicode-Programm wird eine andere, strengere Syntaxprüfung durchgeführt als in Nicht-Unicode-Programmen. In einigen Fällen müssen Anweisungen auch um neue Zusätze erweitert werden. Das vorliegende Buch beschreibt die Regeln für Unicode- und Nicht-Unicode-Programme. Ein syntaktisch korrektes Unicode-Programm läuft in der Regel mit gleicher Semantik und gleichen Ergebnissen in Unicode- und Nicht-Unicode-Systemen.[3] Programme, die in beiden Systemen laufen sollen, sollten daher auch auf beiden Plattformen getestet werden.

In einem Unicode-System können nur Unicode-Programme ausgeführt werden. Vor der Umstellung auf ein Unicode-System muss der Profilparameter ABAP/UNICODE_CHECK auf »ON« gesetzt werden, so dass nur noch die Ausführung von Unicode-Programmen erlaubt ist. Nicht-Unicode-Programme können nur in Nicht-Unicode-Systemen ausgeführt werden. Alle Sprachkonstrukte, die für Unicode-Programme eingeführt wurden, können aber auch in Nicht-Unicode-Programmen verwendet werden.

Es hat sich herausgestellt, dass Programme, die bisher schon sauber programmiert wurden, die neuen Unicode-Regeln größtenteils schon automatisch erfüllen und deshalb kaum Änderungsaufwand hervorrufen. Umgekehrt deuten alle Stellen, an denen größere Änderungen vorzunehmen sind, meistens auf einen fehleranfälligen und daher fragwürdigen Programmierstil hin. Selbst wenn es nicht geplant ist, auf ein Unicode-System umzustellen, sind Unicode-Programme die besseren Programme, weil sie besser wartbar und weniger fehleranfällig sind. So wie in ABAP Objects veraltete und gefährliche Sprachkonstrukte als obsolet erklärt wurden und nicht mehr verwendet werden dürfen, bieten die Regeln für Unicode-Programme erhöhte Sicherheit beim Programmieren, beispielsweise beim Arbeiten mit Zeichenfeldern und gemischten Strukturen. Dies gilt insbesondere auch bei der Ablage externer Daten wie z.B. über die Dateischnittstelle, die für die Verwendung in Unicode-Programmen vollständig überarbeitet wurde. Es wird deshalb empfohlen, beim Anlegen eines neuen Programms dieses immer als Unicode-Programm zu kennzeichnen und ältere Programme schrittweise auf Unicode umzustellen.

3 Ausnahmen von dieser Regel sind systemnahe Programme, die die Anzahl der Bytes pro Zeichen abfragen und auswerten.

1.5 Aufbau des Buches

Das Buch ist in Kapitel unterteilt, die thematisch zusammengehörige Anweisungen beschreiben. Jedes dieser Kapitel enthält eine Übersicht, die in das Thema einführt. Wenn ein einziges ABAP-Schlüsselwort semantisch unterschiedliche Anweisungen einleitet, wie zum Beispiel `GET` oder `CALL`, haben wir diese nicht in einer Syntaxform zusammengefasst, sondern entsprechend der Semantik aufgeteilt und in den thematisch zugehörigen Abschnitten angeordnet.

Abbildung 1.1 zeigt den allgemeinen Aufbau eines ABAP-Programms. Welche Komponenten im konkreten Fall tatsächlich verwendet werden, richtet sich nach den Anforderungen an das Programm und nach seinem Programmtyp.

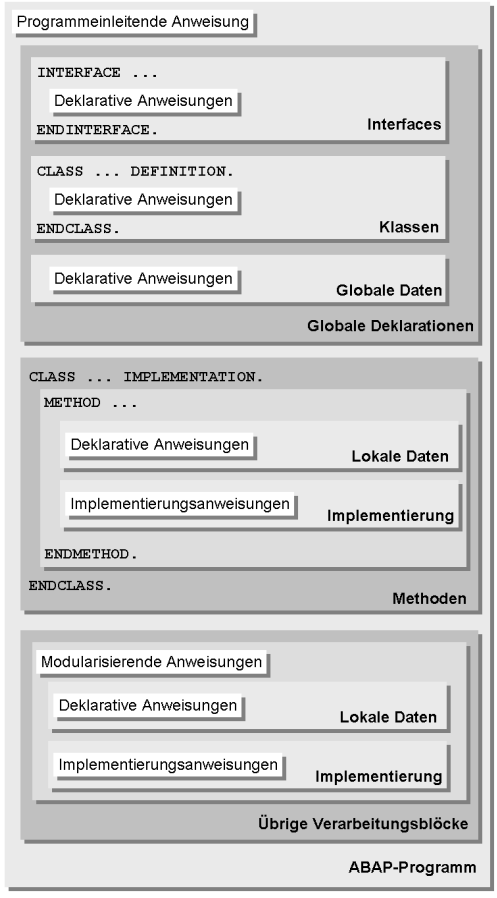


Abbildung 1.1 Allgemeiner Aufbau eines ABAP-Programms

An der Verwendung der beschriebenen Anweisungen in ABAP-Programmen orientiert sich auch die Reihenfolge der Kapitel dieses Buches:

- In **Kapitel 2** beschreiben wir die allgemeine *ABAP-Syntax*. Wir definieren Begriffe wie *Anweisung*, *Schlüsselwort*, *Sprachelement* etc. und gehen insbesondere darauf ein, wie Datenobjekte in Operandenpositionen angegeben werden.
- Jedes ABAP-Programm beginnt mit einer *programmeinleitenden Anweisung*. Diese beschreiben wir in **Kapitel 3**.
- Die Funktionalität eines ABAP-Programms ist in *Verarbeitungsblöcken* implementiert. Verarbeitungsblöcke werden mit *Modularisierungsanweisungen* definiert. In Verarbeitungsblöcken vom Typ »Prozedur« können mit deklarativen Anweisungen lokale Datentypen und Datenobjekte angelegt werden. Alle anderen Verarbeitungsblöcke haben keinen lokalen Datenbereich und deklarative Anweisungen wirken programmglobal. Eine Sonderrolle spielen *Methoden*. Methoden sind Verarbeitungsblöcke, die im Implementierungsteil ihrer Klasse implementiert werden müssen. Die Reihenfolge von Verarbeitungsblöcken bzw. Implementierungsteilen spielt keinerlei Rolle für die Programmausführung, sollte aber die Lesbarkeit eines Programms unterstützen. Die Modularisierungsanweisungen beschreiben wir in **Kapitel 4**.
- In jedem ABAP-Programm stehen vordefinierte *Datentypen* und *Datenobjekte* sowie vordefinierte *Funktionen* zur Verfügung. Diese führen wir in **Kapitel 5** auf.
- Nach der programmeinleitenden Anweisung enthält jedes Programm einen globalen *Deklarationsteil*, in dem Definitionen und Deklarationen vorgenommen werden, die im gesamten Programm gültig und sichtbar sind. Dies sind zum einen die Deklaration von *Datentypen* und *Datenobjekten* und zum anderen die Definition von *Interfaces* oder der Deklarationsteil von *Klassen* in ABAP Objects. Die Definitionen von Interfaces und Klassen enthalten die Deklarationen ihrer *Komponenten*. Die Reihenfolge der einzelnen Definitionen und Deklarationen ist zwar nicht prinzipiell festgelegt, muss sich aber nach der Tatsache richten, dass man sich in einer ABAP-Anweisung immer nur auf vorhergehende Definitionen und Deklarationen beziehen kann. Beispielsweise kann sich eine Referenzvariable nur auf eine zuvor definierte Klasse beziehen und diese wiederum nur ein zuvor definiertes Interface implementieren. Nach der Einleitung einer Prozedur können Datentypen und Datenobjekte deklariert werden, die innerhalb der Prozedur sichtbar sind. Wir beschreiben die deklarativen Anweisungen für Daten-

typen und Datenobjekte in **Kapitel 6** und die Anweisungen zur Definition von Klassen und Interfaces in **Kapitel 7**. Zur Deklaration von Datentypen gehören in weiterem Sinn auch Typisierungen von Objekten, deren Datentyp bei der Programmerstellung noch nicht feststeht. Diese beschreiben wir in **Kapitel 8**.

- Alle übrigen Anweisungen eines ABAP-Programms sind *Implementierungsanweisungen*, die immer einem Verarbeitungsblock zugeordnet werden können. Mit den Implementierungsanweisungen wird die Funktionalität eines Verarbeitungsblocks implementiert. In den **Kapiteln 9 bis 40** beschreiben wir die Implementierungsanweisungen geordnet nach ihrer Funktionalität. Beachten Sie, dass die Funktionalität aller Verarbeitungsblöcke im Wesentlichen mit den gleichen Anweisungen implementiert wird. Aus Gründen der Abwärtskompatibilität sind in Verarbeitungsblöcken außerhalb von ABAP Objects einige Syntaxvarianten möglich, die in Methoden verboten sind. In der Regel sollten diese Varianten prinzipiell nicht mehr verwendet werden. Wir beschreiben die ABAP-Syntax, wie sie in Methoden gültig ist, und weisen gegebenenfalls auf die obsoleten Varianten hin.
- Da ABAP eine historisch gewachsene Sprache ist, enthält es einige ältere Anweisungen oder Zusätze zu Anweisungen, die durch bessere Sprachkonstrukte ersetzt wurden, aus Gründen der Abwärtskompatibilität aber nicht abgeschafft werden können. In ABAP Objects, d.h. bei der Implementierung von Methoden, sind die *obsoleten Sprachelemente* in der Regel syntaktisch verboten. Sie sollten diese Sprachelemente in neuen Programmen zwar prinzipiell nicht mehr verwenden, werden sie aber in älteren Programmen weiterhin vorfinden. Wir beschreiben die obsoleten Anweisungen in **Kapitel 41**. Obsolete Zusätze zu weiterhin aktuellen Anweisungen beschreiben wir in der Regel entsprechend gekennzeichnet bei der Anweisung selbst, teilweise aber auch in Kapitel 41.
- Im **Anhang** werden die *Konvertierungsregeln* für Zuweisungen, einige *sprachnahe Klassen* und *Funktionsbausteine* und die *Ausnahmen* der ABAP-Laufzeitumgebung dargestellt.
- Die Beschreibung der ABAP-Sprachelemente wird durch ein umfangreiches lexikalisch aufgebautes **Glossar** ergänzt, das wichtige Begriffe zentral definiert und Verweise auf die zugehörigen Abschnitte enthält. Ein im Text *kursiv* gesetzter Ausdruck verweist auf einen Eintrag im Glossar.

Hinweis
Begriffe in der männlichen Form wie »Benutzer« oder »Entwickler« werden in diesem Buch lediglich im grammatikalischen Sinn und nur der Einfachheit halber verwendet und sollen keinesfalls Benutzerinnen oder Entwicklerinnen ausschließen.

1.6 Suchmöglichkeiten

Für das Arbeiten mit der ABAP-Referenz bieten sich drei Zugangsmöglichkeiten an:

1. **Suche nach einem Sprachelement**
   Die häufigste Anwendung dieses Buches wird wahrscheinlich sein, dass Sie die genaue Bedeutung eines ABAP-Sprachelements nachschlagen wollen. Hierfür ist der ausführliche zweistufige Index vorgesehen, der ausschließlich ABAP-Sprachelemente enthält. Zusätze zu Schlüsselwörtern werden auf oberster Ebene dargestellt und darunter finden Sie die Verweise auf die Schlüsselwörter, zu denen die Zusätze gehören.
2. **thematische Suche**
   Wenn Sie sich über alle zu einem bestimmten Thema gehörigen Sprachelemente informieren wollen, suchen Sie am besten im Inhaltsverzeichnis nach dem gewünschten Thema. In der Übersicht zum Thema finden Sie eine Liste aller im betreffenden Kapitel beschriebenen ABAP-Anweisungen.
3. **Suche über einen Begriff**
   Wenn Sie sich über einen Begriff aus dem ABAP-Umfeld und die zugehörigen Sprachelemente informieren wollen, bietet es sich an, den Begriff im ABAP-Glossar nachzuschlagen. Bei allen Begriffen des Glossars, die in Bezug zu einem Sprachelement stehen, gibt es einen entsprechenden Verweis.

Natürlich können Sie neben der gezielten Suche einfach auch einmal auf eine kleine Entdeckungsreise durch die ABAP-Referenz gehen.

1.7 Syntaxdiagramme

Tabelle 1.1 zeigt die in den Syntaxdiagrammen dieses Buches verwendete Notation. Der wesentliche Unterschied zur Darstellung in der ABAP-Schlüsselwortdokumentation ist die Verwendung von Pseudosyntax, um Anweisungen mit zahlreichen Varianten in einer einzigen Grundform darstellen zu können. Pseudosyntax wird weiterhin dazu verwendet, um bei Anweisungen mit vielen Zusätzen diese thematisch zu ordnen.

Darstellung	Bedeutung
`ABAP`	ABAP-Wörter werden in Großbuchstaben und in Fettdruck dargestellt.
`operand`	Operanden werden in Kleinbuchstaben dargestellt.
`pseudo_syntax`	Teile von Anweisungen, deren Syntaxdiagramme an späterer Stelle gezeigt werden, sind als Pseudosyntax in Kleinbuchstaben und Fettdruck dargestellt.
`. : , ()`	Punkte, Kommata, Doppelpunkte und runde Klammern werden normal dargestellt. Sie sind Teil der ABAP-Syntax.
+, -, *, / ...	Operatoren werden normal dargestellt. Sie sind Teil der ABAP-Syntax.
`[]`	Teile von Anweisungen, die verwendet werden können, aber nicht müssen, sind in eckige Klammern gesetzt. Eine Aufzählung von Anweisungsteilen in eckigen Klammern bedeutet, dass alle oder einzelne Teile verwendet werden dürfen. Wenn mindestens ein Teil verwendet werden muss, ist dies im Text vermerkt. Eckige Klammern sind nicht Teil der ABAP-Syntax.
|	Striche zwischen Anweisungsteilen bedeuten, dass nur einer der aufgeführten Teile innerhalb einer Anweisung verwendet werden darf. Striche sind nicht Teil der ABAP-Syntax.
`{}`	Geschweifte Klammern fassen zusammengehörige Teile von Anweisungen zusammen, beispielsweise rechts oder links von Strichen (|). Geschweifte Klammern sind nicht Teil der ABAP-Syntax.
`...`	Punkte bedeuten, dass an dieser Stelle andere Teile der Anweisung stehen können.
$operand_i$	Operanden mit tiefergestellten Indizes bedeuten, dass an dieser Stelle eine Liste von Operanden aufgeführt werden kann. Die einzelnen Operanden sind nur durch Leerzeichen getrennt.
{ ... $operand_i$ ...} [... $operand_i$ ...]	Befinden sich Operanden mit Index innerhalb einer geschweiften oder eckigen Klammer, bedeutet das, dass der gesamte Inhalt der Klammer als Liste mit unterschiedlichen Operanden aufgeführt werden kann.
(dobj$_i$)	Befinden sich Operanden mit Index innerhalb einer runden Klammer, bedeutet das, dass in der Klammer eine Liste von durch Kommata getrennten Operanden aufgeführt werden kann.

Tabelle 1.1 Syntaxkonventionen

Die Syntaxdiagramme dieses Buches zeigen die Sprachelemente in einer syntaktisch korrekten Reihenfolge. In vielen Anweisungen sind auch andere Reihenfolgen möglich, werden aber nicht erwähnt. Wir haben in der Regel die Reihenfolge gewählt, die am besten zur Semantik der Anweisung passt und mit der sich die verschiedenen Varianten einer Anweisung einheitlich beschreiben lassen. Es kann deshalb vorkommen, dass Sie in bestehenden Programmen oder der ABAP-Dokumentation eine andere Reihenfolge vorfinden.

Hinweis
Die Wahl von Großbuchstaben für ABAP-Worte und von Kleinbuchstaben für Operanden orientiert sich an einer gängigen Pretty-Printer-Einstellung des ABAP Editor, wie sie auch in der ABAP-Dokumentation verwendet wird. Eine andere Einstellung des Pretty Printer, die seit Release 6.10 auswählbar ist, ermöglicht auch die umgekehrte Darstellung.

1.8 Weitere Informationen zum Buch

Weitere Informationen zum Buch, eventuelle Korrekturlisten, ein Diskussionsforum und Updates finden Sie im Internet unter *www.sap-press.de*.

7 Definition von Klassen und Interfaces

7.1 Übersicht

Dieser Abschnitt beschreibt die Definition von Klassen und Interfaces sowie ihrer Komponenten. Klassen und Interfaces sind die Grundlage von ABAP Objects, dem objektorientierten Bestandteil der Sprache ABAP. Klassen und Interfaces können in ABAP-Programmen folgender Programmtypen (siehe Tabelle 3.1) definiert werden:

- In einem *Class-Pool* kann mit dem Werkzeug *Class Builder* der ABAP Workbench genaus eine globale Klasse der Klassenbibliothek zur Verwendung in allen übrigen ABAP-Programmen definiert werden. Weiterhin können in einem Class-Pool lokale Klassen und Interfaces zur Verwendung im Class-Pool selbst definiert werden.
- In einem *Interface-Pool* kann mit dem Werkzeug *Class Builder* der ABAP Workbench genau ein globales Interface der *Klassenbibliothek* zur Verwendung in allen übrigen ABAP-Programmen definiert werden. In einem Interface-Pool können keine lokalen Klassen und Interfaces definiert werden.
- In allen übrigen ABAP-Programmen außer *Typgruppen* können lokale Klassen und Interfaces zur Verwendung im Programm selbst definiert werden.

Innerhalb von Klassen und Interfaces können außer den in Kapitel 6 beschriebenen Deklarationen weitere Komponenten deklariert werden, die wir ebenfalls in diesem Abschnitt beschreiben.

Die Anweisungen zur Deklaration von Klassen und Interfaces sind:

Anweisung	Abschnitt
`CLASS`	7.2
`INTERFACE`	7.3
`METHODS, CLASS-METHODS`	7.4.1
`EVENTS, CLASS-EVENTS`	7.4.2
`INTERFACES, ALIASES`	7.4.3

7.2 Definition von Klassen

```
CLASS
```

Die vollständige Definition einer Klasse besteht aus einem *Deklarationsteil* und einem *Implementierungsteil*, die beide mit CLASS eingeleitet werden. Im Deklarationsteil werden die *Eigenschaften* der Klasse festgelegt und ihre Komponenten deklariert. Im Implementierungsteil werden die *Methoden* der Klasse implementiert. Weitere Varianten von CLASS dienen dem Bekanntmachen von Klassen in einem Programm.

7.2.1 Deklarationsteil

Definition einer Klasse und Deklaration ihrer Komponenten.

Syntax

```
CLASS class DEFINITION [PUBLIC]
                       [INHERITING FROM superclass]
                       [ABSTRACT]
                       [FINAL]
                       [CREATE {PUBLIC|PROTECTED|PRIVATE}]
                       [[GLOBAL] FRIENDS [class_i] [ifac_i]].
  [PUBLIC SECTION].
    [components]
  [PROTECTED SECTION].
    [components]
  [PRIVATE SECTION].
    [components]
ENDCLASS.
```

Der Anweisungsblock CLASS class DEFINITION – ENDCLASS definiert eine Klasse class. Zwischen CLASS und ENDCLASS werden die Komponenten components der Klasse deklariert. Jede Komponente muss hinter einer der Anweisungen PUBLIC SECTION, PROTECTED SECTION oder PRIVATE SECTION, und diese müssen in der angegebenen Reihenfolge aufgeführt werden. Eine Klasse muss nicht alle SECTION-Anweisungen enthalten.

Mit den Zusätzen der Anweisung CLASS kann eine Klasse global in der *Klassenbibliothek* veröffentlicht, eine *Vererbungsbeziehung* definiert, die Klasse *abstrakt* oder *final* gemacht, die *Instanzierbarkeit* gesteuert und anderen Klassen oder Interfaces die *Freundschaft* angeboten werden.

7.2.1.1 Sichtbarkeitsbereiche

Öffentlicher Sichtbarkeitsbereich

Syntax

```
PUBLIC SECTION.
```

Diese Anweisung definiert den öffentlichen Sichtbarkeitsbereich der Klasse `class`. Alle Komponenten der Klasse, die im Bereich hinter der Anweisung `PUBLIC SECTION` deklariert werden, sind von außerhalb der Klasse, in ihren Unterklassen und in der Klasse selbst ansprechbar.

Geschützter Sichtbarkeitsbereich

Syntax

```
PROTECTED SECTION.
```

Diese Anweisung definiert den geschützten Sichtbarkeitsbereich der Klasse `class`. Alle Komponenten der Klasse, die im Bereich hinter der Anweisung `PROTECTED SECTION` deklariert werden, sind in den Unterklassen der Klasse und in der Klasse selbst ansprechbar.

Privater Sichtbarkeitsbereich

Syntax

```
PRIVATE SECTION.
```

Diese Anweisung definiert den privaten Sichtbarkeitsbereich der Klasse `class`. Alle Komponenten der Klasse, die im Bereich hinter der Anweisung `PRIVATE SECTION` deklariert werden, sind nur in der Klasse selbst ansprechbar.

7.2.1.2 Klassenkomponenten

Syntax

```
components.
```

In den Sichtbarkeitsbereichen werden die Komponenten der Klassen definiert. Folgende Deklarationsanweisungen sind für `components` möglich:

- `TYPE-POOLS`, `TYPES`, `DATA`, `CLASS-DATA`, `CONSTANTS` für Datentypen und Datenobjekte (siehe Kapitel 6)

- METHODS, CLASS-METHODS, EVENTS, CLASS-EVENTS für Methoden und Ereignisse (siehe Abschnitt 7.4)
- INTERFACES zur Implementierung von *Interfaces* und ALIASES für Alias-Namen von *Interfacekomponenten* (siehe Abschnitt 7.4)

Hinweis
Alle Komponenten einer Klasse liegen in einem Namensraum. Innerhalb einer Klasse muss der Name einer Komponente unabhängig von seiner Art (Datentyp, Attribut, Methode oder Ereignis) eindeutig sein. Die Komponenten eines implementierten Interfaces unterscheiden sich durch das Präfix ifac~ (Name des Interfaces mit Interfacekomponenten-Selektor) von den direkt deklarierten Komponenten der Klasse.

7.2.1.3 Globale Klassen

Syntax

```
... PUBLIC ...
```

Durch den Zusatz PUBLIC wird die Klasse class zu einer globalen Klasse der *Klassenbibliothek*. Der Zusatz PUBLIC ist nur bei genau einer Klasse eines *Class-Pools* möglich und wird beim Anlegen einer globalen Klasse vom *Class Builder* erzeugt. Alle Klassen ohne den Zusatz PUBLIC sind lokale Klassen ihres Programms.

Hinweis
Im öffentlichen Sichtbarkeitsbereich globaler Klassen dürfen keine eigenen Datentypen mit der Anweisung TYPES deklariert werden. Außerdem können dort nur global bekannte Datentypen verwendet werden.

7.2.1.4 Definition von Unterklassen

Syntax

```
... INHERITING FROM superclass ...
```

Durch den Zusatz INHERITING FROM wird die Klasse class durch *Vererbung* von der *Oberklasse* superclass abgeleitet und dadurch zu ihrer direkten *Unterklasse*. Die Oberklasse superclass kann eine beliebige nicht-*finale,* an dieser Stelle sichtbare Klasse sein.

Jede Klasse kann nur eine direkte Oberklasse, aber mehrere direkte Unterklassen haben (*Einfachvererbung*). Jede Klasse ohne den Zusatz INHERITING FROM erbt implizit von der vordefinierten leeren Klasse

`object`. Alle Klassen in ABAP Objects bilden einen *Vererbungsbaum*, in dem es von jeder Klasse einen eindeutigen Pfad zum *Wurzelknoten* `object` gibt.

Die Klasse `class` übernimmt alle Komponenten von `superclass`, ohne deren *Sichtbarkeitsbereich* zu ändern. In der Unterklasse sind nur die Komponenten des *öffentlichen* und *geschützten Sichtbarkeitsbereichs* der Oberklasse sichtbar. Die Eigenschaften der übernommenen Komponenten können nicht verändert werden. In einer Unterklasse können zusätzliche Komponenten deklariert und geerbte Methoden redefiniert, d.h. ohne Änderung der Schnittstelle neu implementiert werden.

Hinweis
Die *öffentlichen* und *geschützten* Komponenten aller Klassen innerhalb eines Pfads des *Vererbungsbaums* liegen im gleichen Namensraum. In einer Unterklasse dürfen neue Komponenten nicht genauso heißen wie *öffentliche* oder *geschützte* Komponenten, die von den Oberklassen geerbt sind.

7.2.1.5 Abstrakte Klassen

Syntax

```
... ABSTRACT ...
```

Durch den Zusatz `ABSTRACT` wird eine *abstrakte Klasse* `class` definiert. Von einer abstrakten Klasse können keine *Instanzen* erzeugt werden. Um die *Instanzkomponenten* einer abstrakten Klasse zu verwenden, muss eine nicht-abstrakte Unterklasse der Klasse *instanziert* werden.

7.2.1.6 Finale Klassen

Syntax

```
... FINAL ...
```

Durch den Zusatz `FINAL` wird eine *finale Klasse* `class` definiert. Von einer finalen Klasse können keine *Unterklassen* abgeleitet werden. Alle *Methoden* einer finalen Klasse sind implizit final und dürfen nicht explizit als final deklariert werden.

Hinweis
In Klassen, die gleichzeitig abstrakt und final sind, sind nur die statischen Komponenten verwendbar. Es können zwar Instanzkomponenten deklariert werden, diese sind aber nicht verwendbar.

Beispiel

Im folgenden Beispiel werden eine *abstrakte* Klasse `c1` und eine *finale* Klasse `c2` definiert, wobei `c2` von `c1` erbt. `c1` ist implizit Unterklasse der leeren Klasse `object`. In `c2` kann auf `m1`, aber nicht auf `a1` zugegriffen werden.

```
CLASS c1 DEFINITION ABSTRACT.
  PROTECTED SECTION.
    METHODS m1.
  PRIVATE SECTION.
    DATA a1 TYPE string
            VALUE `Attribute A1 of class C1`.
ENDCLASS.

CLASS c2 DEFINITION INHERITING FROM c1 FINAL.
  PUBLIC SECTION.
    METHODS m2.
ENDCLASS.

CLASS c1 IMPLEMENTATION.
  METHOD m1.
    WRITE / a1.
  ENDMETHOD.
ENDCLASS.

CLASS c2 IMPLEMENTATION.
  METHOD m2.
    m1( ).
  ENDMETHOD.
ENDCLASS.

...

DATA oref TYPE REF TO c2.

CREATE OBJECT oref.
oref->m2( ).
```

7.2.1.7 Instanzierbarkeit

Syntax

```
... CREATE {PUBLIC|PROTECTED|PRIVATE} ...
```

Der Zusatz CREATE legt fest, in welchem *Kontext* die Klasse class instanziert wird, d.h., wo die Anweisung CREATE OBJECT für diese Klasse ausgeführt werden kann.

- Eine Klasse mit dem Zusatz CREATE PUBLIC kann überall dort instanziert werden, wo die Klasse sichtbar ist.
- Eine Klasse mit dem Zusatz CREATE PROTECTED kann nur in Methoden ihrer Unterklassen und der Klasse selbst instanziert werden.
- Eine Klasse mit dem Zusatz CREATE PRIVATE kann nur in Methoden der Klasse selbst instanziert werden. Das bedeutet insbesondere, dass sie auch nicht als vererbter Bestandteil von Unterklassen instanziert werden kann.

Die Instanzierbarkeit einer Klasse hängt wie folgt von der direkten *Oberklasse* ab:

- Direkte Unterklassen von object oder einer Klasse mit dem Zusatz CREATE PUBLIC erben implizit den Zusatz CREATE PUBLIC. Explizit können alle CREATE-Zusätze angegeben werden.
- Direkte Unterklassen einer Klasse mit dem Zusatz CREATE PROTECTED erben implizit den Zusatz CREATE PROTECTED. Explizit können alle CREATE-Zusätze angegeben werden.
- Direkte Unterklassen einer Klasse mit dem Zusatz CREATE PRIVATE, die keine *Freunde* der Klasse sind, erhalten implizit einen Zusatz CREATE NONE. Sie sind nicht instanzierbar und es dürfen keine expliziten CREATE-Zusätze angegeben werden. Direkte Unterklassen, die *Freunde* der Klasse sind, erben implizit den Zusatz CREATE PRIVATE. Explizit können bei Freunden privat instanzierbarer Oberklassen alle CREATE-Zusätze angegeben werden.

Hinweis
Es empfiehlt sich, eine privat instanzierbare Klasse gleichzeitig *final* zu machen, da ihre Unterklassen nicht instanziert werden können, wenn sie keine Freunde der Klasse sind.

7.2.1.8 Freunde

Syntax

AB RELEASE 6.10

```
... [GLOBAL] FRIENDS [class_i] [ifac_i].
```

Mit dem Zusatz FRIENDS macht die Klasse class die Klassen $class_i$ bzw. die Interfaces $ifac_i$ zu ihren *Freunden*. Gleichzeitig werden sämtli-

che Unterklassen der Klassen $class_i$, sämtliche Klassen, die eines der Interfaces $ifac_i$ implementieren und sämtliche Interfaces, die eines der Interfaces $ifac_i$ als *Komponenteninterface* enthalten, zu Freunden der Klasse class.

Die Freunde einer Klasse haben unbeschränkten Zugriff auf die *geschützten* und *privaten* Komponenten der Klasse und können uneingeschränkt *Instanzen* der Klasse erzeugen.

Die Freunde von class sind nicht automatisch auch Freunde der *Unterklassen* von class. Die Klasse class wird durch den Zusatz FRIENDS nicht zum Freund ihrer Freunde.

Ohne die Zusätze GLOBAL oder LOCAL können für $class_i$ und $ifac_i$ alle an dieser Stelle sichtbaren Klassen und Interfaces angegeben werden. Falls globale Klassen und Interfaces der Klassenbibliothek zu Freunden gemacht werden, ist zu beachten, dass in diesen die lokalen Klassen anderer ABAP-Programme nicht sichtbar sind. Ein statischer Zugriff auf die Komponenten der Klasse class ist in solchen Freunden nicht möglich.

Der Zusatz GLOBAL ist nur bei gleichzeitiger Verwendung des Zusatzes PUBLIC, also für die globale Klasse eines *Class-Pool*s erlaubt. Hinter GLOBAL FRIENDS können andere globale Klassen und Interfaces der Klassenbibliothek aufgeführt werden. Dieser Zusatz wird gegebenenfalls beim Anlegen einer globalen Klasse vom Class Builder erzeugt.

Beispiel
In folgendem Beispiel ist die Klasse c2 Freund des Interfaces i1 und damit auch der implementierenden Klasse c1. Sie kann diese instanzieren und auf ihre private Komponente a1 zugreifen.

```
INTERFACE i1.
  ...
ENDINTERFACE.

CLASS c1 DEFINITION CREATE PRIVATE FRIENDS i1.
  PRIVATE SECTION.
    DATA a1(10) TYPE c VALUE 'Class 1'.
ENDCLASS.

CLASS c2 DEFINITION.
  PUBLIC SECTION.
    INTERFACES i1.
    METHODS m2.
ENDCLASS.
```

```
CLASS c2 IMPLEMENTATION.
  METHOD m2.
    DATA oref TYPE REF TO c1.
    CREATE OBJECT oref.
    WRITE  oref->a1.
  ENDMETHOD.
ENDCLASS.
```

Lokale Freunde globaler Klassen

Hierfür gibt es eine eigene Anweisung.

Syntax

AB RELEASE 6.10

```
CLASS class DEFINITION LOCAL FRIENDS [class_i] [ifac_i].
```

Diese Anweisung macht die lokalen Klassen und Interfaces $class_i$ und $ifac_i$ eines *Class-Pool*s zu Freunden seiner globalen Klasse `class`.

Diese Anweisung leitet keinen Deklarationsteil ein und darf nicht mit `ENDCLASS` abgeschlossen werden. Sie muss im *Class-Pool* vor der Definition der lokalen Klassen und Interfaces $class_i$ und $ifac_i$ aufgeführt werden, wenn diese Zugriff auf die *privaten* und *geschützten* Komponenten der globalen Klasse `class` haben sollen.

7.2.2 Implementierungsteil

Implementierung der *Methoden* einer Klasse.

Syntax

```
CLASS class IMPLEMENTATION.
  ...
  METHOD ...
    ...
  ENDMETHOD.
  ...
ENDCLASS.
```

Im Anweisungsblock `CLASS class IMPLEMENTATION – ENDCLASS` müssen folgende Methoden einer Klasse `class` in beliebiger Reihenfolge implementiert werden:

- alle nicht-*abstrakten* Methoden, die mit `METHODS` oder `CLASS-METHODS` im Deklarationsteil der Klasse deklariert sind
- alle nicht-*abstrakten* Methoden von Interfaces, die mit der Anweisung `INTERFACES` im Deklarationsteil der Klasse aufgeführt sind

- alle von *Oberklassen* geerbten Methoden, die im Deklarationsteil der Klasse mit der Anweisung `METHODS ... REDEFINITION` aufgeführt sind

Die Implementierung jeder Methode entspricht einem Verarbeitungsblock `METHOD` – `ENDMETHOD` (siehe Abschnitt 4.2.1). Außerhalb von Methodenimplementierungen sind im Implementierungsteil keine Anweisungen erlaubt. In einer Methodenimplementierung kann in Instanzmethoden auf alle Komponenten und in statischen Methoden auf alle statischen Komponenten der eigenen Klasse zugegriffen werden. Für die Adressierung von Komponenten der eigenen Klasse ist kein Komponentenselektor notwendig. Innerhalb der Implementierung jeder *Instanzmethode* steht zur Laufzeit eine implizit erzeugte lokale *Referenzvariable* namens `me` zur Verfügung, die auf die aktuelle *Instanz* der Methode zeigt.

Hinweis
Eine Klasse, die aufgrund ihres Deklarationsteils keine Methoden implementieren muss, hat entweder einen leeren oder gar keinen Implementierungsteil.

Beispiel
Im folgenden Beispiel müssen drei Methoden der Klasse `c2` implementiert werden. Die Methode `m1` in `c1` ist abstrakt und muss nicht implementiert werden.

```
INTERFACE i1.
  METHODS m1.
ENDINTERFACE.

CLASS c1 DEFINITION ABSTRACT.
  PROTECTED SECTION.
    METHODS m1 ABSTRACT.
ENDCLASS.

CLASS c2 DEFINITION INHERITING FROM c1.
  PUBLIC SECTION.
    INTERFACES i1.
    METHODS m2.
  PROTECTED SECTION.
    METHODS m1 REDEFINITION.
ENDCLASS.

CLASS c2 IMPLEMENTATION.
  METHOD m1.
```

```
    ...
  ENDMETHOD.
  METHOD m2.
    ...
  ENDMETHOD.
  METHOD i1~m1.
    ...
  ENDMETHOD.
ENDCLASS.
```

7.2.3 Klassen bekannt machen

In einigen Fällen muss die Definition einer Klasse vor ihrer Verwendung explizit bekannt gemacht werden.

Syntax

```
CLASS class DEFINITION { DEFERRED | LOAD }.
```

Diese beiden Varianten der Anweisung `CLASS` dienen dazu, die Klasse `class` unabhängig vom Ort der eigentlichen Definition im Programm bekannt zu machen. Sie leiten keinen Deklarationsteil ein und dürfen nicht mit `ENDCLASS` abgeschlossen werden. Die Zusätze haben folgende Bedeutung:

- Die Variante mit dem Zusatz `DEFERRED` macht eine lokale Klasse vor ihrer eigentlichen Definition im Programm bekannt. Das Programm muss an späterer Stelle einen Deklarationsteil für `class` enthalten. Die Anweisung ist notwendig, wenn man sich auf eine Klasse beziehen will, bevor sie definiert wird. Ein Zugriff auf einzelne Komponenten ist nicht vor der eigentlichen Definition möglich.
- Die Variante mit dem Zusatz `LOAD` lädt eine globale Klasse `class` aus der *Klassenbibliothek*. Die Anweisung ist notwendig, wenn in einem Programm auf eine der *statischen Komponenten* von `class` zugegriffen oder ein *Ereignisbehandler* für `class` deklariert werden soll, bevor `class` automatisch geladen wurde. Alle anderen Zugriffe laden `class` automatisch.

Beispiel

Im folgenden Beispiel verwendet die Klasse `c1` die Klasse `c2` und umgekehrt. Deshalb muss eine der Klassen vor ihrer eigentlichen Definition bekannt gemacht werden. Weiterhin wird die globale Klasse `cl_gui_cfw` vor der Verwendung eines ihrer statischen Attribute explizit geladen.

```
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.

CLASS c1_gui_cfw DEFINITION LOAD.

DATA state LIKE c1_gui_cfw=>system_state.
```

7.3 Definition von Interfaces

```
INTERFACE
```

Die Definition eines Interfaces besteht aus einem Deklarationsteil, der mit INTERFACE eingeleitet wird. Ein Interface hat im Gegensatz zu Klassen keinen Implementierungsteil. Weitere Varianten von INTERFACE dienen dem Bekanntmachen von Interfaces in einem Programm.

7.3.1 Deklarationsteil

Definition eines Interfaces und Deklaration seiner Komponenten.

Syntax

```
INTERFACE ifac.
  [components]
ENDINTERFACE.
```

Der Anweisungsblock INTERFACE – ENDINTERFACE definiert ein Interface ifac. Zwischen INTERFACE und ENDINTERFACE werden die Komponenten components des Interfaces deklariert.

7.3.1.1 Interfacekomponenten

Syntax

```
components
```

Folgende Deklarationsanweisungen sind für `components` möglich:

- `TYPE-POOLS`, `TYPES`, `DATA`, `CLASS-DATA`, `CONSTANTS` für Datentypen und Datenobjekte (siehe Kapitel 6)
- `METHODS`, `CLASS-METHODS`, `EVENTS`, `CLASS-EVENTS` für Methoden und Ereignisse (siehe Abschnitt 7.4)
- `INTERFACES` zur Einbindung von *Komponenteninterfaces* und `ALIASES` für Alias-Namen für deren Komponenten (siehe Abschnitt 7.4)

Hinweis

Alle Komponenten eines Interfaces liegen in einem Namensraum. Innerhalb eines Interfaces muss der Name einer Komponente unabhängig von seiner Art (Datentyp, Attribut, Methode oder Ereignis) eindeutig sein. Die Komponenten eines eingebundenen Interfaces unterscheiden sich durch das Präfix `ifac~` (Name des Interfaces mit Interfacekomponenten-Selektor) von den direkt deklarierten Komponenten.

Beispiel

Im folgenden Beispiel wird ein Interface `i1` mit drei Interfacekomponenten `a1`, `m1` und `e1` deklariert. Die Klasse `c1` implementiert das Interface, wodurch die Interfacekomponenten zu *öffentlichen* Komponenten der Klasse werden, die über den Interfacekomponenten-Selektor (~) ansprechbar sind.

```
INTERFACE i1.
  DATA    a1 TYPE string.
  METHODS m1.
  EVENTS  e1 EXPORTING value(p1) TYPE string.
ENDINTERFACE.

CLASS c1 DEFINITION.
  PUBLIC SECTION.
    INTERFACES i1.
ENDCLASS.

CLASS c1 IMPLEMENTATION.
  METHOD i1~m1.
    RAISE EVENT i1~e1 EXPORTING p1 = i1~a1.
  ENDMETHOD.
ENDCLASS.
```

7.3.2 Interfaces bekannt machen

In einigen Fällen muss die Definition eines Interfaces vor seiner Verwendung explizit bekannt gemacht werden.

Syntax

```
INTERFACE ifac { DEFERRED | LOAD }.
```

Diese beiden Varianten der Anweisung `INTERFACE` dienen dazu, das Interface `ifac` unabhängig vom Ort der eigentlichen Definition im Programm bekannt zu machen. Sie leiten keinen Deklarationsteil ein und dürfen nicht mit `ENDINTERFACE` abgeschlossen werden. Die Bedeutung der beiden Anweisungen für Interfaces ist identisch mit der Bedeutung der Anweisungen `CLASS` in Abschnitt 7.2.3 für Klassen.

7.4 Deklaration von Komponenten in Klassen und Interfaces

Im Deklarationsteil von Klassen und Interfaces werden die Komponenten von Klassen und Interfaces deklariert, die wesentliche Komponenten einer Klasse sind:

- Attribute
- Methoden
- Ereignisse

Attribute sind die Datenobjekte einer Klasse. Daneben ist auch die Deklaration klasseneigener Datentypen möglich. Die Deklaration von Attributen und Datentypen erfolgt mit den allgemeinen Anweisungen `TYPE-POOLS`, `TYPES`, `DATA`, `CLASS-DATA` und `CONSTANTS`, die mit Ausnahme von `CLASS-DATA` auch in anderen Kontexten möglich und in Kapitel 6 beschrieben sind.

Die Deklaration von *Methoden* und *Ereignissen* erfolgt mit speziellen Anweisungen, die nur im Deklarationsteil von Klassen und Interfaces möglich sind.

7.4.1 Methoden

Methoden werden mit den Anweisungen `METHODS` und `CLASS-METHODS` deklariert. Sie bestimmen das Verhalten einer Klasse. Bei der Deklaration wird die Schnittstelle einer Methode definiert. Für spezielle Aufgaben gibt es unterschiedliche Arten von Methoden:

- allgemeine Methoden
- funktionale Methoden
- Konstruktoren
- Ereignisbehandler

7.4.1.1 Instanzmethoden

```
METHODS
```

Die Anweisung METHODS deklariert *Instanzmethoden*. Instanzmethoden sind an Objekte gebunden. Um Instanzmethoden zu verwenden, muss zunächst ein Objekt der Klasse erzeugt werden. Instanzmethoden können ohne *Komponentenselektor* auf alle Komponenten der eigenen Klasse zugreifen.

7.4.1.2 Allgemeine Instanzmethoden

Die allgemeinste Form der METHODS-Anweisung erlaubt die Definition von Instanzmethoden mit beliebigen *Ein-* und *Ausgabeparametern*.

Syntax

```
METHODS meth [ABSTRACT|FINAL]
  [IMPORTING {parameter_i} [PREFERRED PARAMETER p]]
  [EXPORTING {parameter_i}]
  [CHANGING  {parameter_i}]
  [{RAISING|EXCEPTIONS} {exc_i}].
```

Diese Anweisung deklariert eine allgemeine Instanzmethode meth. Mit den Zusätzen ABSTRACT und FINAL wird die Methode abstrakt bzw. final gemacht.

Die übrigen Zusätze definieren die Parameterschnittstelle der Methode und legen fest, welche *Ausnahmen* die Methode propagieren bzw. auslösen kann.

Parameterschnittstelle

Die Zusätze IMPORTING, EXPORTING und CHANGING definieren die *Signatur* der Methode meth:

- IMPORTING definiert *Eingabeparameter*
  Beim Aufruf der Methode muss für jeden nicht-optionalen Eingabeparameter ein passender Aktualparameter angegeben werden. Der Inhalt

des Aktualparameters wird beim Aufruf an den Eingabeparameter übergeben.

- `EXPORTING` definiert *Ausgabeparameter*
  Beim Aufruf der Methode kann für jeden Ausgabeparameter ein passender Aktualparameter angegeben werden. Der Inhalt des Ausgabeparameters wird bei fehlerfreier Beendigung der Methode an den Aktualparameter übergeben.
- `CHANGING` definiert *Ein-/Ausgabeparameter*
  Beim Aufruf der Methode muss für jeden nicht-optionalen Ein-/Ausgabeparameter ein passender Aktualparameter angegeben werden. Der Inhalt des Aktualparameters wird beim Aufruf an den Ein-/Ausgabeparameter übergeben und bei Beendigung der Methode wird der Inhalt des Ein-/Ausgabeparameters an den Aktualparameter übergeben.

Hinter jedem Zusatz werden die entsprechenden Formalparameter durch die Angabe einer Liste $\texttt{parameter}_i$ definiert.

Syntax von $\texttt{parameter}_i$

```
... { {VALUE(pi)}|[REFERENCE](pi)}|{pi} {typing}
      [OPTIONAL|{DEFAULT defi}] } ...
```

Mit `VALUE` oder `REFERENCE` wird festgelegt, ob ein Parameter $\texttt{p}_i$ per Wert oder per Referenz übergeben wird. Wenn nur ein Name $\texttt{p}_i$ angegeben wird, wird der Parameter standardmäßig per Referenz übergeben. Ein per Referenz übergebener Eingabeparameter darf in der Methode nicht verändert werden.

Mit dem Zusatz `typing` muss jeder Formalparameter typisiert werden. Die Syntax von `typing` ist in Abschnitt 8.2 beschrieben. Die Typisierung eines Formalparameters bewirkt, dass bei der Übergabe eines Aktualparameters dessen Datentyp gegen die Typisierung geprüft wird (siehe Abschnitt 8.3). Weiterhin legt die Typisierung fest, an welchen Operandenpositionen der Formalparameter in der Methode verwendet werden kann.

Mit `OPTIONAL` oder `DEFAULT` können *Eingabeparameter* und *Ein-/Ausgabeparameter* als optionale Parameter definiert werden, wobei mit `DEFAULT` ein Ersatzparameter $\texttt{def}_i$ angegeben werden kann. Für einen optionalen Parameter muss beim Aufruf der Methode kein Aktualparameter angegeben werden. Während ein Formalparameter mit dem Zusatz `OPTIONAL` dann typgerecht initialisiert wird, übernimmt ein Formalparameter mit dem Zusatz `DEFAULT` den Wert des Ersatzparameters

def_i. Als Ersatzparameter def_i kann jedes vom Typ passende Datenobjekt angegeben werden, das an dieser Stelle sichtbar ist.

Mit PREFERRED PARAMETER kann ein Eingabeparameter p der Liste $parameter_i$ hinter IMPORTING als bevorzugter Parameter gekennzeichnet werden. Diese Angabe ist nur dann sinnvoll, wenn alle *Eingabeparameter* optional sind. Bei einem Aufruf der Methode mit der Syntax

```
[CALL METHOD] meth( a ).
```

wird dann der Aktualparameter a dem bevorzugten Parameter p zugeordnet.

Hinweis
Falls ein Formalparameter als Referenzvariable typisiert ist und nicht in der Prozedur verändert werden kann, wird die Typprüfung wie für einen *Narrowing Cast* ausgeführt. Wenn er in der Prozedur verändert werden kann, muss der Aktualparameter kompatibel zum Formalparameter sein.

Beispiel
Die Methode read_spfli_into_table des folgenden Beispiels hat einen Eingabe- und einen Ausgabeparameter, die durch Bezug auf das ABAP Dictionary vollständig typisiert sind.

```
CLASS flights DEFINITION.
  PUBLIC SECTION.
    METHODS read_spfli_into_table
       IMPORTING VALUE(id)  TYPE spfli-carrid
       EXPORTING flight_tab TYPE spfli_tab.
       ...
ENDCLASS.
```

Klassenbasierte Ausnahmen

Syntax

AB RELEASE 6.10

```
... RAISING {exc_i} ...
```

Mit dem Zusatz RAISING können klassenbasierte *Ausnahmen* exc_i an den Aufrufer weitergereicht werden, die *Unterklassen* von CX_STATIC_CHECK und CX_DYNAMIC_CHECK sind und in der Methode von der ABAP-Laufzeitumgebung oder mit der Anweisung RAISE EXCEPTION ausgelöst bzw. in dieses propagiert, aber nicht in einem TRY-Block behandelt werden.

Für exc$_i$ können alle an dieser Stelle sichtbaren Ausnahmeklassen, die Unterklassen obiger Oberklassen sind, angegeben werden. Die Ausnahmeklassen müssen in aufsteigender Reihenfolge bezüglich ihrer Vererbungshierarchie angegeben werden.

Tritt in der Methode eine Ausnahme dieser *Oberklassen* auf, die weder behandelt noch weitergereicht wird, führt dies entweder zu einem Syntaxfehler oder zu einer vom Aufrufer behandelbaren Ausnahme CX_SY_NO_HANDLER.

Hinweis

In einer Methode, die mit dem Zusatz RAISING klassenbasierte Ausnahmen propagiert, kann die Anweisung CATCH SYSTEM-EXCEPTIONS nicht verwendet werden. Stattdessen sind die entsprechenden behandelbaren Ausnahmen in einem TRY-Block abzufangen (siehe Kapitel 15).

Beispiel

In der folgenden Klasse math propagiert die Methode divide_1_by alle Ausnahmen, die durch die Klasse CX_SY_ARITHMETIC_ERROR und deren Unterklassen repräsentiert werden. Falls beispielsweise dem Eingabeparameter operand beim Aufruf der Wert 0 übergeben wird, wird die Ausnahme CX_SY_ZERODIVIDE ausgelöst, propagiert und kann – wie im Beispiel gezeigt – vom Aufrufer in einem TRY-Block behandelt werden.

```
CLASS math DEFINITION.
  PUBLIC SECTION.
    METHODS divide_1_by
       IMPORTING operand TYPE I
       EXPORTING result  TYPE f
       RAISING   cx_sy_arithmetic_error.
ENDCLASS.

CLASS math IMPLEMENTATION.
  METHOD divide_1_by.
    result = 1 / operand.
  ENDMETHOD.
ENDCLASS.

...

DATA oref TYPE REF TO math.
DATA exc  TYPE REF TO cx_sy_arithmetic_error.

DATA result TYPE f.
DATA text   TYPE string.
```

```
CREATE OBJECT oref.
TRY.
    oref->divide_1_by( EXPORTING operand = ...
                       IMPORTING result = result ).
    text = result.
  CATCH cx_sy_arithmetic_error INTO exc.
    text = exc->get_text( ).
ENDTRY.
MESSAGE text TYPE 'I'.
```

Ausnahmen vor Release 6.10

Syntax

```
... EXCEPTIONS {exc_i} ...
```

Mit dem Zusatz EXCEPTIONS wird eine Liste von Ausnahmen exc_i definiert, die mit den Anweisungen RAISE oder MESSAGE RAISING in der Methode ausgelöst werden können. Diese Ausnahmen sind – ähnlich wie Formalparameter – an die Methode gebunden und können nicht propagiert werden.

Wird eine solche Ausnahme in einer Methode ausgelöst und nicht mit dem gleichnamigen Zusatz EXCEPTIONS der Anweisung CALL METHOD beim Methodenaufruf behandelt, kommt es zu einem *Laufzeitfehler*.

Hinweis

Die beiden Zusätze RAISING und EXCEPTIONS können nicht gleichzeitig verwendet werden. Für Neuentwicklungen ab Release 6.10 wird empfohlen, mit klassenbasierten Ausnahmen zu arbeiten, die unabhängig von der konkreten Methode sind.

Beispiel

In der folgenden Klasse math ist für die Methode divide_1_by eine eigene Ausnahme arith_error definiert, die in der Methode mit der Anweisung RAISE ausgelöst wird, wenn ein arithmetischer Fehler auftritt. Falls beispielsweise dem Eingabeparameter operand beim Aufruf der Wert 0 übergeben wird, wird die Ausnahme arith_error bei der methodeninternen Behandlung der Ausnahme CX_SY_ZERODIVIDE ausgelöst und beim Aufruf der Methode behandelt.

```
CLASS math DEFINITION.
  PUBLIC SECTION.
    METHODS divide_1_by
```

```
       IMPORTING  operand TYPE I
       EXPORTING  result  TYPE f
       EXCEPTIONS arith_error.
ENDCLASS.

CLASS math IMPLEMENTATION.
  METHOD divide_1_by.
    TRY.
        result = 1 / operand.
      CATCH cx_sy_arithmetic_error.
        RAISE arith_error.
    ENDTRY.
  ENDMETHOD.
ENDCLASS.

...

DATA result TYPE f.
DATA text TYPE string.

DATA oref TYPE REF TO math.
DATA exc  TYPE REF TO cx_sy_arithmetic_error.

CREATE OBJECT oref.
oref->divide_1_by( EXPORTING  operand  = ...
                   IMPORTING  result = result
                   EXCEPTIONS arith_error = 4 ).

IF sy-subrc = 0.
  WRITE result.
ELSE.
  WRITE 'Arithmetic error!'.
ENDIF.
```

Abstrakte Methoden

Syntax

```
... ABSTRACT ...
```

Der Zusatz `ABSTRACT` ist nur in Klassen möglich, nicht in Interfaces. Durch den Zusatz `ABSTRACT` wird eine *abstrakte Methode* `meth` definiert. Eine abstrakte Methode wird nicht im Implementierungsteil ihrer Klasse

implementiert und kann nur in abstrakten Klassen deklariert werden. Um eine abstrakte Methode zu implementieren, muss sie in einer nicht-abstrakten Unterklasse mit dem Zusatz REDEFINITION redefiniert werden.

Hinweis
Abstrakte Methoden sind zwar auch in Klassen definierbar, die sowohl abstrakt als auch final sind, können aber nie implementiert werden und sind daher nicht verwendbar.

Finale Methoden

Syntax

```
METHODS ... FINAL ...
```

Der Zusatz FINAL ist nur in Klassen möglich, nicht in Interfaces. Durch den Zusatz FINAL wird eine *finale Methode* meth definiert. Eine finale Methode kann in einer Unterklasse nicht redefiniert werden. In finalen Klassen sind alle Methoden automatisch final und der Zusatz FINAL ist nicht erlaubt.

7.4.1.3 Funktionale Instanzmethoden

Funktionale Methoden haben beliebig viele *Eingabeparameter* und genau einen *Rückgabewert* als *Ausgabeparameter*.

Syntax

```
METHODS meth [ABSTRACT|FINAL]
  [IMPORTING {parameter_i}]
   RETURNING VALUE(r) {TYPE type_spec}|{LIKE dobj_spec}
  [{RAISING|EXCEPTIONS} {exc_i}].
```

Diese Anweisung deklariert eine funktionale Instanzmethode meth. Für die Zusätze ABSTRACT, FINAL, IMPORTING, RAISING und EXCEPTIONS gilt das Gleiche wie für allgemeine Instanzmethoden.

Statt der Zusätze EXPORTING und CHANGING hat eine funktionale Methode einen Zusatz RETURNING, der genau einen Formalparameter r als *Rückgabewert* definiert. Der Rückgabewert muss mit VALUE per Wert übergeben werden.

Für die Typisierung des Rückgabewerts r mit typing gelten die gleichen Regeln wie für die Typisierung von IMPORTING-, EXPORTING- und CHANGING-Parametern mit der Besonderheit, dass ein Rückgabewert immer

vollständig typisiert sein muss: In `typing` kann keiner der generischen Typen aus Tabelle 5.3 angegeben werden (siehe Abschnitt 8.2).

Wenn funktionale Methoden wie in Abschnitt 2.2.5 beschrieben an den Operandenpositionen von Anweisungen eingesetzt werden, an denen ein Datenobjekt erwartet wird, bestimmt die vollständige Typisierung des Rückgabewerts den Datentyp des Operanden. Bei der Ausführung einer solchen Anweisung wird die funktionale Methode aufgerufen und der Inhalt des Rückgabewerts ersetzt den Inhalt des Operanden.

Hinweise

- Wenn eine funktionale Methode in einer Operandenposition verwendet wird, können klassenbasierte Ausnahmen, die die Methode mit `RAISING` propagiert, wie bei allgemeinen Methoden in einem `TRY`-Block behandelt bzw. weiter propagiert werden. Die mit `EXCEPTIONS` definierten Ausnahmen einer funktionalen Methode können dagegen an Operandenpositionen nicht behandelt werden und führen immer zu einem Laufzeitfehler.
- Wenn eine funktionale Methode den gleichen Namen wie eine eingebaute Methode hat, wird mit dem Ausdruck `meth( a )` an einer Operandenposition immer die funktionale Methode aufgerufen. Dieser Fall kann nur in den Methoden der Klasse der funktionalen Methode auftreten.

Beispiel

Die funktionale Methode `factorial` des folgenden Beispiels hat einen Rückgabewert `fact` vom Typ `i`. Die `COMPUTE`-Anweisung zeigt den Aufruf der Methode an einer Operandenposition.

```
CLASS math DEFINITION.
  PUBLIC SECTION.
    METHODS factorial
       IMPORTING n TYPE i
       RETURNING value(fact) TYPE i.
ENDCLASS.

CLASS math IMPLEMENTATION.
  METHOD factorial.
    fact = 1.
    IF n = 0.
      RETURN.
    ELSE.
      DO n TIMES.
```

```
        fact = fact * sy-index.
      ENDDO.
    ENDIF.
  ENDMETHOD.
ENDCLASS.

...

DATA oref TYPE REF TO math.
DATA result TYPE i.

CREATE OBJECT oref.
COMPUTE result = oref->factorial( ... ).
```

7.4.1.4 Instanzkonstruktoren

Instanzkonstruktoren sind Methoden mit dem vorgegebenen Namen `constructor`, die bei der *Instanzierung* ihrer Klasse automatisch aufgerufen werden. *Konstruktoren* haben beliebig viele *Eingabeparameter* und keine *Ausgabeparameter*.

Syntax

```
METHODS constructor [FINAL]
  [IMPORTING {parameter_i}]
  [{RAISING|EXCEPTIONS} {exc_i}].
```

Diese Anweisung ist nur im *öffentlichen Sichtbarkeitsbereich* des Deklarationsteils einer Klasse möglich. Sie deklariert den Instanzkonstruktor `constructor` der Klasse.

Jede Klasse hat in ihrem öffentlichen Sichtbarkeitsbereich eine vordefinierte Methode namens `constructor`. Durch die explizite Deklaration kann die Schnittstelle der Methode `constructor` klassenspezifisch definiert und ihre Funktionalität implementiert werden. Ohne explizite Deklaration übernimmt der Instanzkonstruktor die Signatur des Instanzkonstruktors der direkten Oberklasse und ruft diesen implizit auf.

Falls der Instanzkonstruktor in einer Unterklasse implementiert wird, muss der Instanzkonstruktor der Oberklasse explizit über die Pseudoreferenz `super->constructor` aufgerufen und seine Schnittstelle versorgt werden. Ausgenommen hiervon sind nur die direkten Unterklassen des *Wurzelknotens* `object`. Vor dem Aufruf des Oberklassenkonstruktors hat ein Instanzkonstruktor nur auf die *statischen Komponenten* seiner Klasse

Zugriff. Nach dem Aufruf des Oberklassenkonstruktors kann auch auf *Instanzkomponenten* zugegriffen werden.

Mit dem Zusatz `IMPORTING` können *Eingabeparameter* nach den gleichen Regeln wie für allgemeine Methoden definiert werden. Die Zusätze `RAISING` und `EXCEPTIONS` zur Propagierung bzw. Definition von Ausnahmen haben ebenfalls die gleiche Bedeutung wie bei allgemeinen Methoden.

Der Instanzkonstruktor wird für jede Instanz einer Klasse genau einmal durch die Anweisung `CREATE OBJECT` direkt nach ihrer Erzeugung aufgerufen. Dabei müssen allen nicht-optionalen *Eingabeparametern* passende *Aktualparameter* zugeordnet werden und die Ausnahmen können behandelt bzw. weitergereicht werden. Ein Aufruf über `CALL METHOD` ist außer beim Aufruf des Oberklassenkonstruktors über `super->constructor` im redefinierten Konstruktor einer Unterklasse nicht möglich.

Instanzkonstruktoren sind implizit *final*. Der Zusatz `FINAL` kann zwar angegeben werden, ist aber nicht notwendig.

Hinweise

- Instanzkonstruktoren sind eine Ausnahme von der Regel, dass alle öffentlichen Komponenten entlang eines Pfads im *Vererbungsbaum* in einem Namensraum liegen. Der Instanzkonstruktor jeder Klasse hat eine eigene Schnittstelle und eine eigene Implementierung. Ein Instanzkonstruktor kann nicht redefiniert werden.
- Instanzkonstruktoren werden nur aus technischen Gründen im öffentlichen Sichtbarkeitsbereich einer Klasse deklariert. Die tatsächliche Sichtbarkeit wird durch den Zusatz `CREATE {PUBLIC|PROTECTED|PRIVATE}` der Anweisung `CLASS DEFINITION` gesteuert.

Beispiel

Im folgenden Beispiel erbt die Klasse `c2` von der Klasse `c1`. In beiden Klassen ist der Instanzkonstruktor `constructor` explizit deklariert. Deshalb muss er in beiden Klassen implementiert werden, wobei die Implementierung in `c2` den Aufruf des Oberklassenkonstruktors enthalten muss.

```
CLASS c1 DEFINITION.
  PUBLIC SECTION.
    METHODS constructor IMPORTING p1 TYPE any.
    ...
ENDCLASS.
```

```
CLASS c2 DEFINITION INHERITING FROM c1.
  PUBLIC SECTION.
    METHODS constructor IMPORTING p2 TYPE any.
    ...
ENDCLASS.

CLASS c1 IMPLEMENTATION.
  METHOD constructor.
    ...
  ENDMETHOD.
ENDCLASS.

CLASS c2 IMPLEMENTATION.
  METHOD constructor.
    ...
    super->constructor( p2 ).
    ...
  ENDMETHOD.
ENDCLASS.
```

7.4.1.5 Ereignisbehandler

Ereignisbehandler sind Methoden, die zwar auch mit `CALL METHOD`, hauptsächlich aber durch das Auslösen eines *Ereignisses* einer Klasse oder eines Interfaces aufgerufen werden können. Die einzig möglichen *Formalparameter* eines Ereignisbehandlers sind *Eingabeparameter*, die als *Ausgabeparameter* des Ereignisses definiert wurden.

Syntax

```
METHODS meth [ABSTRACT|FINAL]
   FOR EVENT evt OF {class|ifac}
  [IMPORTING {pi} [sender]].
```

Diese Anweisung deklariert die Instanzmethode `meth` als Ereignisbehandler für das Ereignis `evt` der Klasse `class` bzw. des Interfaces `ifac`. Für `class` und `ifac` können alle an dieser Stelle sichtbaren Klassen und Interfaces angegeben werden, die ein an dieser Stelle sichtbares Ereignis `evt` als Komponente enthalten.

Wenn das Ereignis `evt` ein Instanzereignis ist, kann der Ereignisbehandler `meth` es in allen Objekten behandeln, deren Klassen gleich `class` oder Unterklasse von `class` sind bzw. das Interface `ifac` direkt oder über eine Oberklasse implementieren. Wenn das Ereignis ein statisches Ereignis ist,

kann es der Ereignisbehandler meth in der Klasse class und deren Unterklassen bzw. in allen Klassen, die das Interface ifac implementieren, behandeln.

Mit den Zusätzen ABSTRACT und FINAL können Ereignisbehandler genau wie allgemeine Methoden entweder abstrakt oder final gemacht werden.

Der Zusatz IMPORTING definiert die *Eingabeparameter* des Ereignisbehandlers. Für p_i können nur diejenigen Namen von Formalparametern angegeben werden, die mit dem Zusatz EXPORTING der Anweisung [CLASS-]EVENTS bei der Deklaration des Ereignisses evt in der Klasse class bzw. dem Interface ifac als *Ausgabeparameter* des Ereignisses definiert wurden. Die Zusätze TYPE bzw. LIKE und OPTIONAL bzw. DEFAULT sind nicht möglich. Die Typisierung der Eingabeparameter, die Eigenschaft, ob sie optional sind, und eventuelle Ersatzparameter werden von der Deklaration des Ereignisses übernommen. Es müssen nicht alle Ausgabeparameter des Ereignisses angegeben werden.

Falls evt ein *Instanzereignis* ist, kann zusätzlich zu seinen explizit definierten Ausgabeparametern ein Formalparameter namens sender als Eingabeparameter eines Ereignisbehandlers definiert werden. Der Formalparameter sender ist ein impliziter Ausgabeparameter jedes Instanzereignisses. Er ist vollständig als *Referenzvariable* typisiert, die als *statischen Typ* die Klasse class bzw. das Interfaces ifac hat, wie es in der Deklaration des Ereignisbehandlers hinter EVENT evt OF angegeben ist. Wenn der Ereignisbehandler durch ein Instanzereignis aufgerufen wird, bekommt er in sender eine Referenz auf das auslösende Objekt übergeben.

AB RELEASE 6.10

Vor Release 6.10 wurde der statische Typ des Formalparameters sender durch die Klasse bzw. das Interface bestimmt, in der bzw. dem das Ereignis mit der Anweisung EVENTS deklariert wurde. Seit Release 6.10 bestimmt der Ereignisbehandler den Typ seines Formalparameters selbst.

Beispiel
Die folgende Klasse picture enthält einen Ereignisbehandler handle_double_click für das Instanzereignis picture_dblclick der globalen Klasse cl_gui_picture. Der Ereignisbehandler übernimmt als Eingabeparameter zwei explizite Ausgabeparameter des Ereignisses sowie den impliziten Parameter sender.

```
CLASS picture DEFINITION.
  PUBLIC SECTION.
    METHODS handle_double_click
```

```
            FOR EVENT picture_dblclick OF cl_gui_picture
            IMPORTING mouse_pos_x mouse_pos_y sender.
ENDCLASS.

CLASS picture IMPLEMENTATION.
  METHOD handle_double_click.
    ...
  ENDMETHOD.
ENDCLASS.
```

7.4.1.6 Instanzmethoden redefinieren

Eine in einer Oberklasse deklarierte Methode kann in einer Unterklasse redefiniert werden, solange sie in der Oberklasse nicht als *final* gekennzeichnet ist. Bei einer Redefinition wird die Schnittstelle der Methode nicht geändert.

Syntax

```
METHODS meth [FINAL] REDEFINITION.
```

Diese Anweisung ist nur in *Unterklassen* möglich und redefiniert eine geerbte Instanzmethode `meth`. Sie bewirkt, dass die Methode `meth` im Implementierungsteil der Unterklasse neu implementiert werden muss. Die neue Implementierung verschattet in der aktuellen Klasse die Implementierung der Oberklasse. Die redefinierte Methode greift auf die privaten Komponenten der redefinierenden Klasse zu und nicht auf eventuelle gleichnamige private Komponenten der Oberklasse. In der redefinierten Methode kann die Implementierung der direkten Oberklasse über die Pseudoreferenz `super->meth` aufgerufen werden.

Für `meth` kann außer dem Instanzkonstruktor jede nicht-*finale* Instanzmethode angegeben werden, die im *öffentlichen* oder *geschützten Sichtbarkeitsbereich* einer *Oberklasse* der aktuellen Klasse deklariert ist. Insbesondere kann `meth` eine *abstrakte* Methode einer abstrakten Oberklasse sein. Die Redefinition muss im gleichen Sichtbarkeitsbereich wie die Deklaration der Methode erfolgen. Die Schnittstelle und Art der Methode (allgemeine oder funktionale Instanzmethode, Ereignisbehandler) wird bei einer Redefinition nicht verändert.

Die Redefinition ist für die Unterklassen der redefinierenden Klasse gültig, bis sie erneut redefiniert wird. Eine Methode ist entlang eines Pfads im Vererbungsbaum so lange redefinierbar, bis sie *final* gemacht wird.

Wenn der Zusatz FINAL bei der Redefinition verwendet wird, ist die Methode ab der aktuellen Klasse final und in ihren Unterklassen nicht mehr redefinierbar.

Hinweis

Jede Objektreferenz, die auf ein Objekt einer Unterklasse zeigt, adressiert unabhängig von ihrem *statischen Typ* die redefinierten Methoden. Dies gilt insbesondere auch für die Selbstreferenz me->.

Beispiel

Im folgenden Beispiel wird die Methode m1 der Oberklasse c1 in der Unterklasse c2 redefiniert, wobei die ursprüngliche Implementierung mit super->m1 aufgerufen wird. Beide Methoden arbeiten mit dem privaten Attribut a1 der jeweiligen Klasse. Beim Aufruf über die Referenzvariable oref, die den statischen Typ c1 und den dynamischen Typ c2 hat, wird die redefinierte Methode ausgeführt.

```
CLASS c1 DEFINITION.
  PUBLIC SECTION.
    METHODS m1 IMPORTING p1 TYPE string.
  PRIVATE SECTION.
    DATA a1 TYPE string VALUE `c1: `.
ENDCLASS.

CLASS c2 DEFINITION INHERITING FROM c1.
  PUBLIC SECTION.
    METHODS m1 REDEFINITION.
  PRIVATE SECTION.
    DATA a1 TYPE string VALUE `c2: `.
ENDCLASS.

CLASS c1 IMPLEMENTATION.
  METHOD m1.
    CONCATENATE a1 p1 INTO a1.
    WRITE / a1.
  ENDMETHOD.
ENDCLASS.

CLASS c2 IMPLEMENTATION.
  METHOD m1.
    super->m1( p1 ).
    CONCATENATE a1 p1 INTO a1.
    WRITE / a1.
  ENDMETHOD.
ENDCLASS.
```

```
...

DATA oref TYPE REF TO c1.

CREATE OBJECT oref TYPE c2.
oref->m1( `...` ).
```

7.4.1.7 Statische Methoden

```
CLASS-METHODS
```

Die Anweisung `CLASS-METHODS` deklariert *statische Methoden*. Statische Methoden können über den *Klassenkomponenten-Selektor* (=>) unabhängig von Objekten verwendet werden. Statische Methoden können ohne *Komponentenselektor* nur auf die *statischen Komponenten* der eigenen Klasse oder ihrer Oberklassen zugreifen.

7.4.1.8 Allgemeine statische Methoden

Die allgemeinste Form der `CLASS-METHODS`-Anweisung erlaubt die Definition von statischen Methoden mit beliebig vielen *Ein-* und *Ausgabeparametern*.

Syntax

```
CLASS-METHODS meth
  [IMPORTING {parameter_i} [PREFERRED PARAMETER p]]
  [EXPORTING {parameter_i}]
  [CHANGING  {parameter_i}]
  [{RAISING|EXCEPTIONS} {exc_i}].
```

Diese Anweisung deklariert eine allgemeine statische Methode `meth`. Mit den Zusätzen wird die Parameterschnittstelle der Methode definiert und festgelegt, welche *Ausnahmen* die Methode propagieren bzw. auslösen kann. Die Syntax und Bedeutung der Zusätze ist dieselbe wie bei allgemeinen Instanzmethoden.

Hinweis
Statische Methoden sind nicht redefinierbar und können deshalb auch nicht *abstrakt* oder *final* gemacht werden.

7.4.1.9 Funktionale statische Methoden

Funktionale Methoden haben beliebig viele *Eingabeparameter* und genau einen *Rückgabewert* als *Ausgabeparameter*.

Syntax

```
CLASS-METHODS meth
  [IMPORTING {parameter_i}]
   RETURNING VALUE(r) {TYPE type_spec}|{LIKE dobj_spec}
  [{RAISING|EXCEPTIONS} {exc_i}].
```

Diese Anweisung deklariert eine funktionale statische Methode `meth`. Die Zusätze haben genau dieselbe Syntax und Bedeutung wie bei funktionalen Instanzmethoden.

Beispiel

Die folgende Klasse `circle` enthält zwei funktionale statische Methoden `circumference` und `area`, die mit der Konstanten p_i arbeiten.

```
CLASS circle DEFINITION.
  PUBLIC SECTION.
    CONSTANTS p_i TYPE f
                VALUE '3.14159265358979324'.
    CLASS-METHODS: circumference IMPORTING r TYPE f
                                 RETURNING value(c)
                                           TYPE f,
                   area          IMPORTING r TYPE f
                                 RETURNING value(a)
                                           TYPE f.
ENDCLASS.

CLASS circle IMPLEMENTATION.
  METHOD circumference.
    c = 2 * p_i * r.
  ENDMETHOD.
  METHOD area.
    a = p_i * r ** 2.
  ENDMETHOD.
ENDCLASS.

...

DATA: circ TYPE f,
      area TYPE f,
      radius TYPE f.

radius = ...
circ = circle=>circumference( radius ).
area = circle=>area( radius ).
```

7.4.1.10 Statische Konstruktoren

Statische Konstruktoren sind Methoden mit dem vorgegebenen Namen `class_constructor`, die vor der ersten Verwendung ihrer Klasse automatisch aufgerufen werden. *Statische Konstruktoren* haben keine Parameterschnittstelle.

Syntax

```
CLASS-METHODS class_constructor.
```

Diese Anweisung ist nur im öffentlichen Sichtbarkeitsbereich des Deklarationsteils einer Klasse möglich. Sie deklariert den statischen Konstruktor `class_constructor` der Klasse.

Jede Klasse hat in ihrem öffentlichen Sichtbarkeitsbereich eine vordefinierte Methode namens `class_constructor`. Durch die explizite Deklaration kann ihre Funktionalität klassenspezifisch implementiert werden. Ohne explizite Deklaration ist der statische Konstruktor leer.

Der statische Konstruktor wird genau einmal pro Klasse und *internem Modus* vor dem ersten Zugriff auf die Klasse automatisch ausgeführt. Ein Zugriff auf die Klasse ist die Erzeugung einer Instanz der Klasse oder die Adressierung einer statischen Komponente über den Klassenkomponenten-Selektor.

Beim ersten Zugriff auf eine Unterklasse wird im *Vererbungsbaum* nach der nächsthöheren Oberklasse gesucht, deren statischer Konstruktor noch nicht ausgeführt wurde. Erst im Anschluss daran wird der statische Konstruktor dieser Oberklasse ausgeführt und danach sukzessive der aller folgenden Unterklassen bis zur angesprochenen Unterklasse.

Wie alle statischen Methoden kann auch der statische Konstruktor nur auf die *statischen Komponenten* seiner Klasse zugreifen. Weiterhin darf der statische Konstruktor seine eigene Klasse nicht explizit ansprechen.

Hinweise

- Wie *Instanzkonstruktoren* sind auch statische Konstruktoren eine Ausnahme von der Regel, dass alle öffentlichen Komponenten entlang eines Pfads im *Vererbungsbaum* in einem Namensraum liegen.
- Der Zeitpunkt der Ausführung des statischen Konstruktors liegt nicht definitiv fest. Es wird allein seine Ausführung vor dem ersten Zugriff auf die Klasse garantiert. In der Regel wird der statische Konstruktor unmittelbar vor dem Zugriff auf die Klasse ausgeführt. Nur bei einem

Zugriff auf ein statisches Attribut der Klasse wird der statische Konstruktor schon zu Beginn des jeweiligen *Verarbeitungsblocks* aufgerufen. Eine Programmierung, die sich auf die Ausführung des statischen Konstruktors zu einem bestimmten Zeitpunkt verlässt, wird nicht empfohlen.

Beispiel
Der statische Konstruktor der folgenden Klasse setzt beim ersten Zugriff auf die Klasse das statische Attribut `access_program` mithilfe des Systemfeldes `sy-repid` auf den Namen des Programms eines *internen Modus*, das die Klasse als Erstes verwendet.

```
CLASS some_class DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS class_constructor.
  PRIVATE SECTION.
    CLASS-DATA access_program TYPE sy-repid.
ENDCLASS.

CLASS some_class IMPLEMENTATION.
  METHOD class_constructor.
    access_program = sy-repid.
  ENDMETHOD.
ENDCLASS.
```

7.4.1.11 Statische Ereignisbehandler

Statische Ereignisbehandler sind statische Methoden, die durch ein Ereignis einer Klasse oder eines Interfaces aufgerufen werden. Die einzig möglichen *Formalparameter* eines Ereignisbehandlers sind *Eingabeparameter*, die als *Ausgabeparameter* des Ereignisses definiert wurden.

```
CLASS-METHODS meth
   FOR EVENT evt OF {class|ifac}
  [IMPORTING {pi} [sender]].
```

Diese Anweisung deklariert die statische Methode `meth` als Ereignisbehandler für das Ereignis `evt` der Klasse `class` bzw. des Interfaces `ifac`. Syntax und Bedeutung der Zusätze sind identisch mit der Deklaration von Instanzmethoden als Ereignisbehandler.

Statische Ereignisbehandler können unabhängig von einer Instanz der Klasse von dem Ereignis `evt` aufgerufen werden.

Beispiel

In der folgenden Klasse `dialog_box` wird ein statischer Ereignisbehandler `close_box` für das Ereignis definiert, das ausgelöst wird, wenn ein Benutzer eine Dialogbox des GUI Control Framework (CFW) schließen will.

```
CLASS dialog_box DEFINITION.
  PUBLIC SECTION.
    METHODS constructor.
    ...
  PRIVATE SECTION.
    CLASS-DATA open_boxes TYPE i.
    CLASS-METHODS close_box
      FOR EVENT close OF cl_gui_dialogbox_container
      IMPORTING sender.
    ...
ENDCLASS.

CLASS dialog_box IMPLEMENTATION.
  METHOD constructor.
    ... " create a dialogbox
    open_boxes = open_boxes + 1.
  ENDMETHOD.
  METHOD close_box
    ... " close the dialogbox referred by sender
    open_boxes = open_boxes - 1.
  ENDMETHOD.
ENDCLASS.
```

7.4.2 Ereignisse

Ereignisse werden mit den Anweisungen `EVENTS` und `CLASS-EVENTS` deklariert. Die Deklaration eines Ereignisses in einer Klasse bewirkt, dass die Methoden der Klasse das Ereignis auslösen und dadurch die Ausführung von *Ereignisbehandlern* bewirken können. Bei der Deklaration können für ein Ereignis Ausgabeparameter definiert werden, für die beim Auslösen *Aktualparameter* an die Ereignisbehandler übergeben werden.

7.4.2.1 Instanzereignisse

```
EVENTS
```

Die Anweisung `EVENTS` deklariert *Instanzereignisse*. Instanzereignisse sind an Objekte gebunden. Sie können nur in *Instanzmethoden* der gleichen Klasse ausgelöst werden.

Syntax

```
EVENTS evt [EXPORTING { VALUE(pi) typing
                       [OPTIONAL|{DEFAULT defi}]  } ].
```

Diese Anweisung deklariert ein Instanzereignis `evt` in einer Klasse oder einem Interface. Das Instanzereignis `evt` kann in allen Instanzmethoden der gleichen Klasse bzw. einer Klasse, die das Interface implementiert, und – falls dort sichtbar – in den Instanzmethoden von Unterklassen mit der Anweisung `RAISE EVENT` ausgelöst werden.

Der Zusatz `EXPORTING` definiert die Parameterschnittstelle des Ereignisses `evt`. Ein Ereignis hat ausschließlich Ausgabeparameter p_i, die per Wert übergeben werden. Die Syntax der Zusätze `VALUE`, `OPTIONAL`, `DEFAULT` und die Typisierung mit `typing` entspricht der Definition der Formalparameter in der Schnittstelle von Methoden mit der Anweisung `METHODS`.

Für alle nicht-optionalen Ausgabeparameter müssen, für alle optionalen Ausgabeparameter können beim Auslösen des Ereignisses mit der Anweisung `RAISE EVENT` passende Aktualparameter angegeben werden. Optionale Parameter, für die kein Aktualparameter angegeben wird, werden auf ihren typgerechten Initialwert bzw. auf den Ersatzparameter def_i gesetzt.

Bei der Deklaration eines Ereignisbehandlers mit dem Zusatz `FOR EVENT OF` der Anweisung `[CLASS-]METHODS` können die Ausgabeparameter des Ereignisses als Eingabeparameter des Ereignisbehandlers definiert werden, wobei die Eigenschaften der Eingabeparameter von den in `EVENTS` definierten Ausgabeparametern übernommen werden.

Neben den explizit mit `EXPORTING` definierten Ausgabeparametern hat jedes Instanzereignis implizit einen Ausgabeparameter `sender`. Dieser Ausgabeparameter ist vom Typ einer *Referenzvariablen*. Beim Auslösen des Ereignisses mit der Anweisung `RAISE EVENT` bekommt `sender` implizit die Referenz auf das auslösende Objekt zugewiesen.

Bis Release 6.10 wurde der statische Typ des impliziten Parameters `sender` durch die Klasse bzw. das Interface bestimmt, in der die Anweisung `EVENTS` aufgeführt ist. Seit Release 6.10 wird für jeden Ereignisbehandler der statische Typ des Eingabeparameters `sender` durch den Zusatz `FOR EVENT OF` der Anweisung `[CLASS-]METHODS` bestimmt.

Hinweis
Der dynamische Typ des impliziten Formalparameters `sender` ist immer die Klasse des Objekts, in dem das Ereignis ausgelöst wird.

Beispiel
Im folgenden Interface `window` werden drei Ereignisse mit jeweils einem expliziten nicht-optionalen Ausgabeparameter `status` deklariert. Die Klasse `dialog_window` implementiert das Interface `window`. Das Interface `window_handler` enthält Ereignisbehandler, die sowohl den expliziten Parameter `status` als auch den impliziten Parameter `sender` importieren. Der statische Typ des Eingabeparameters `sender` ist die Klasse `dialog_window`. Vor Release 6.10 war es das Interface `window`.

```
INTERFACE window.
  EVENTS: minimize EXPORTING VALUE(status) TYPE i,
          maximize EXPORTING VALUE(status) TYPE i,
          restore  EXPORTING VALUE(status) TYPE i.
ENDINTERFACE.

CLASS dialog_window DEFINITION.
  PUBLIC SECTION.
    INTERFACES window.
ENDCLASS.

INTERFACE window_handler.
  METHODS: minimize_window
             FOR EVENT window~minimize OF dialog_window
             IMPORTING status sender,
           maximize_window
             FOR EVENT window~maximize OF dialog_window
             IMPORTING status sender,
           restore
             FOR EVENT window~restore OF dialog_window
             IMPORTING status sender.
ENDINTERFACE.
```

7.4.2.2 Statische Ereignisse

```
CLASS-EVENTS
```

Die Anweisung `CLASS-EVENTS` deklariert statische Ereignisse. Statische Ereignisse sind nicht an Objekte gebunden. Sie können in allen *Methoden* der gleichen Klasse ausgelöst werden.

Syntax

```
CLASS-EVENTS evt [EXPORTING { VALUE(pi) typing
                        [OPTIONAL|{DEFAULT defi}]  } ].
```

Diese Anweisung deklariert ein statisches Ereignis `evt` in einer Klasse oder einem Interface. Das statische Ereignis `evt` kann in allen Methoden der gleichen Klasse bzw. einer Klasse, die das Interface implementiert, und – falls dort sichtbar – in den Methoden von Unterklassen mit der Anweisung `RAISE EVENT` ausgelöst werden.

Der Zusatz `EXPORTING` definiert die Parameterschnittstelle des Ereignisses `evt`. Syntax und Bedeutung der Zusätze `VALUE`, `OPTIONAL`, `DEFAULT` und die Typisierung mit `typing` entsprechen der Definition von Instanzereignissen mit der Anweisung `EVENTS`.

Hinweis
Statische Ereignisse haben keinen impliziten Formalparameter `sender`.

7.4.3 Interfaces implementieren bzw. einbinden

Interfaces können von Klassen implementiert oder von anderen Interfaces eingebunden werden. Für Interfacekomponenten können Aliasnamen definiert werden.

```
INTERFACES
```

Die Anweisung `INTERFACES` ist im *öffentlichen Sichtbarkeitsbereich* des Deklarationsteils von Klassen und in Interface-Deklarationen möglich.

7.4.3.1 Interfaces in Klassen implementieren

Syntax

```
INTERFACES ifac
  { { [ABSTRACT METHODS {methi}]
      [FINAL METHODS {methj}] }
    | [ALL METHODS {ABSTRACT|FINAL}] }
  [DATA VALUES {attri = vali}].
```

Im *öffentlichen Sichtbarkeitsbereich* einer Klasse implementiert die Anweisung `INTERFACES` das Interface `ifac` in der Klasse. Zudem ist die Angabe von Zusätzen möglich, die die Eigenschaften von Interfacekomponenten in der Klasse bestimmen.

Für ifac können alle an dieser Stelle sichtbaren lokalen oder globalen Interfaces angegeben werden. Durch die Implementierung werden die Komponenten des Interfaces zu öffentlichen Komponenten der Klasse. Eine *Interfacekomponente* namens comp hat in der Klasse den Namen ifac~comp, wobei ifac der Name des Interfaces und das Zeichen ~ der *Interfacekomponenten-Selektor* ist. Eine Klasse muss alle Methoden des Interfaces in ihrem Implementierungsteil implementieren, solange sie sie nicht als abstrakt erklärt.

Mit den Zusätzen ABSTRACT METHODS und FINAL METHODS können einzelne Instanzmethoden $meth_i$ des Interfaces in der implementierenden Klasse *abstrakt* oder *final* gemacht werden. Dabei gelten die gleichen Regeln wie für die Zusätze ABSTRACT und FINAL der Anweisung METHODS. Insbesondere muss die gesamte Klasse abstrakt sein, wenn eine Interface-Methode abstrakt gemacht wird und es darf keine Interface-Methode gleichzeitig hinter ABSTRACT METHODS und FINAL METHODS aufgeführt werden. Statt in der Klasse einzelne Interface-Methoden abstrakt oder final zu machen, können mit dem Zusatz ALL METHODS {ABSTRACT|FINAL} alle Interface-Methoden entweder abstrakt oder final gemacht werden.

Mit dem Zusatz DATA VALUES können einzelnen Attributen $attr_i$ Anfangswerte zugeordnet werden. Der Zusatz erfüllt für Instanzattribute die gleiche Funktion wie der Zusatz VALUE der Anweisung DATA für klasseneigene Attribute. Konstanten, die im Interface mit der Anweisung CONSTANTS deklariert sind, können nicht hinter dem Zusatz DATA VALUES aufgeführt werden.

Hinweise

Eine Klasse kann beliebig viele verschiedene Interfaces implementieren. Alle von einer Klasse implementierten Interfaces liegen gleichberechtigt auf einer Ebene. Wenn ein in einer Klasse implementiertes Interface ifac zusammengesetzt ist, d.h. *Komponenteninterfaces* enthält, werden diese unabhängig von ihrer Schachtelungshierarchie wie einzelne Interfaces in der Klasse implementiert und ihre Komponenten werden nicht über den Namen ifac, sondern den ihres Komponenteninterfaces angesprochen. Die mehrfache Verwendung des Interfacekomponenten-Selektors in einem Bezeichner ($ifac_1$~$ifac_2$~comp) ist prinzipiell nicht möglich.

Jedes Interface kommt in einer Klasse genau einmal vor und jede Interfacekomponente comp ist immer eindeutig über ifac~comp ansprechbar. Auch die Komponenten eines Interfaces, das dadurch, dass es Interface-

komponente eines oder mehrerer anderer Interfaces ist, scheinbar mehrmals in einer Klasse implementiert werden kann, gibt es nur einmal.

7.4.3.2 Interfaces in Interfaces einbinden

Syntax

```
INTERFACES ifac.
```

In der Deklaration eines Interfaces bindet die Anweisung `INTERFACES` das Interface `ifac` in dem deklarierten Interface ein. Die Angabe von Zusätzen ist nicht möglich. Das Interface `ifac` wird dadurch zum *Komponenteninterface* eines *zusammengesetzten Interface*.

Ein Interface kann aus beliebig vielen verschiedenen Interfaces zusammengesetzt werden. Alle Komponenteninterfaces liegen gleichberechtigt auf einer Ebene. Wenn ein Komponenteninterface selbst zusammengesetzt ist, d.h. Komponenteninterfaces enthält, spielt die Schachtelungshierarchie keine Rolle für die Zusammensetzung des Interfaces, aber für die Zugriffsmöglichkeiten auf Interfacekomponenten.

Um innerhalb eines zusammengesetzten Interfaces auf die Komponente `comp` eines Komponenteninterfaces `ifac` zuzugreifen, kann der Ausdruck `ifac~comp` mit dem Interfacekomponenten-Selektor (~) verwendet werden. Die mehrfache Verwendung des Interfacekomponenten-Selektors in einem Bezeichner ($ifac_1$~$ifac_2$~`comp`) ist prinzipiell nicht möglich. In einem zusammengesetzten Interface kann mit dem Interfacekomponenten-Selektor nur auf die Interfacekomponenten der Komponenteninterfaces zugegriffen werden, die in diesem Interface mit der Anweisung `INTERFACES` eingebunden werden. Um in einem zusammengesetzten Interface auf Komponenten zuzugreifen, die nicht über die einmalige Verwendung des Interfacekomponenten-Selektors adressierbar sind, müssen mit der Anweisung `ALIASES` *Aliasnamen* für die Interfacekomponenten deklariert werden.

Hinweis
Jedes Interface kommt mit seinen Komponenten in einem zusammengesetzten Interface genau einmal vor. Auch ein Interface, das dadurch, dass es Interfacekomponente eines oder mehrerer anderer Interfaces ist, scheinbar mehrmals in ein Interface eingebunden werden kann, gibt es nur einmal.

Da es keine getrennten Namensräume für globale und lokale Interfaces gibt, ist bei der Zusammensetzung lokaler Interfaces darauf zu achten, dass es nicht zu Kombinationen gleichnamiger globaler und lokaler Inter-

faces kommt, da diese bei ihrer Implementierung nicht auf einer Ebene liegen könnten.

Beispiel

Das folgende Beispiel zeigt, wie mit der Anweisung `INTERFACES` Interfaces zusammengesetzt und implementiert werden. Die Klasse `c1` implementiert die zusammengesetzten Interfaces `i2` und `i3`. Obwohl `i1` Komponenteninterface von `i2` und `i3` ist, ist es in der Klasse `c1` nur einmal vorhanden. Mit einer Referenzvariablen `iref1` vom statischen Typ `i1` wird ein Objekt der Klasse `c1` erzeugt und die dort implementierte Methode `i1~m1` aufgerufen.

```
INTERFACE i1.
  METHODS m1.
ENDINTERFACE.

INTERFACE i2.
  INTERFACES i1.
  METHODS m2.
ENDINTERFACE.

INTERFACE i3.
  INTERFACES i1.
  METHODS m3.
ENDINTERFACE.

CLASS c1 DEFINITION.
  PUBLIC SECTION.
    INTERFACES: i2, i3.
ENDCLASS.

CLASS c1 IMPLEMENTATION.
  METHOD i1~m1.
    ...
  ENDMETHOD.
  METHOD i2~m2.
    ...
  ENDMETHOD.
  METHOD i3~m3.
    ...
  ENDMETHOD.
ENDCLASS.
```

```
...
DATA iref1 TYPE REF TO i1.
CREATE OBJECT iref1 TYPE c1.
iref1->m1( ).
```

7.4.3.3 Aliasnamen

```
ALIASES
```

Deklaration von *Aliasnamen* für Interfacekomponenten.

Syntax

```
ALIASES alias FOR ifac~comp.
```

Diese Anweisung deklariert im Deklarationsteil einer Klasse oder eines Interfaces einen *Aliasnamen* `alias` für eine Komponente `comp` des Interfaces `ifac`. Das Interface `ifac` muss in der gleichen Klasse implementiert bzw. im gleichen Interface eingebunden sein. Der Aliasname kann überall, wo er sichtbar ist, statt `ifac~comp` verwendet werden, um auf die Interfacekomponente `comp` zuzugreifen.

Ein Aliasname gehört zu den Komponenten der Klasse und des Interfaces. Er liegt im gleichen Namensraum wie die übrigen Komponenten und wird an Unterklassen vererbt. In Klassen kann ein Aliasname in jedem *Sichtbarkeitsbereich* deklariert werden.

Hinweis

Bei der Implementierung von Interface-Methoden im *Implementierungsteil* von Klassen darf der Aliasname nicht verwendet werden. In der Anweisung `METHOD` muss immer der vollständige Bezeichner `ifac~meth` angegeben werden.

Beispiel

In den folgenden Interfaces `i2`, `i3` und der Klasse `c1` werden *Aliasnamen* für die Methoden der eingebundenen bzw. implementierten Interfaces deklariert. Im Implementierungsteil der Klasse werden die Interface-Methoden in den `METHODS`-Anweisungen über den Interfacekomponenten-Selektor implementiert. Innerhalb der Methoden können die Aliasnamen der Klasse verwendet werden.

```
INTERFACE i1.
  METHODS meth.
ENDINTERFACE.

INTERFACE i2.
  INTERFACES i1.
  ALIASES m1 FOR i1~meth.
  METHODS meth.
ENDINTERFACE.

INTERFACE i3.
  INTERFACES i2.
  ALIASES: m1 FOR i2~m1,
           m2 FOR i2~meth.
  METHODS meth.
ENDINTERFACE.

CLASS c1 DEFINITION.
  PUBLIC SECTION.
    INTERFACES i3.
    ALIASES: m1 FOR i3~m1,
             m2 FOR i3~m2,
             m3 FOR i3~meth.
ENDCLASS.

CLASS c1 IMPLEMENTATION.
  METHOD i1~meth.
    ... m2( ) ...
  ENDMETHOD.
  METHOD i2~meth.
    ... m3( ) ...
  ENDMETHOD.
  METHOD i3~meth.
    ... m1( ) ....
  ENDMETHOD.
ENDCLASS.
```