11. Erweiterbares Typsystem

- **Benutzerdefinierte Typen: Überblick**[1]
  - Typdefinition
  - Methodenspezifikation
- **Typklassifikation**
- **Umbenannte Typen (UDTs)**
  - Erhöhung der Typsicherheit
  - Casting-Funktionen und neue Operationen/Methoden
- **Strukturierte Typen (UDTs)**
  - Spalten basierend auf UDTs
  - Tabellen mit Typbindung
  - Typ- und Tabellenhierarchien
  - Einsatzbeispiele
- **Konstruierte Typen**
  - Tupeltypen
  - Referenztypen
  - Kollektionstypen
- **Von SQL aufrufbare Routinen**
  - SQL-Routinen und externe Routinen
  - Funktionsresolution

1. Benutzung als Synonyme: Relation - Tabelle, Tupel - Zeile, Attribut - Spalte

Benutzerdefinierte Typen

- **Benutzerdefinierte Datentypen**
  - benutzerdefinierte, benannte Typen zur Modellierung von Entities
  - Beispiele: Angestellter, Projekt, Geld, Polygon, Text, ...
- **Benutzerdefinierte Methoden und Funktionen (Operatoren)**
  - benutzerdefinierte Operationen zur Modellierung des Verhaltens von Entities in speziellen Anwendungsbereichen
  - lokale und globale Definition in einem DB-Schema möglich
  - Beispiel: Einstellung, Beurteilung, Konversion, Längenbestimmung, Enthaltensein, Ranking, ...
- **Was ist im Vergleich zu vordefinierten Typen (vom Entwerfer) zu tun?**
  - Instanzierung:
    Wie werden die Daten zugewiesen und physisch in einer Tabelle gespeichert?
  - Ordnung:
    Wie vergleicht man Werte des Typs?
  - Verhalten:
    Wie werden Werte des Typs manipuliert (z.B. +, –, ...)?
  - Casting:
    Wie können Werte des Typs konvertiert werden in Werte eines anderen Types (z.B. Wirtssprachenanbindung)

Benutzerdefinierte Typen (2)
- Schlüsseleigenschaften -

- **Neue Funktionalität**
  - **beliebige** Erweiterung der Menge der verfügbaren Typen
  - **beliebige** Erweiterung der Menge der Operationen auf Typen
    - ➥ Erhöhung der Modellierungsmächtigkeit von SQL, um komplexe Operationen/Berechnungen ins DBMS zu verlagern

- **Flexibilität**
  - Integration von relationalen und objektorientierten Konzepten in einer einzigen Sprache
  - Definition von typspezifischem Verhalten für neue Typen (anwendungsspezifische Semantik)

- **Konsistenz**

  Typsicherheit (strong typing) gewährleistet, daß Funktionen auf korrekte Werte von Typen angewendet werden

- **Kapselung**

  Anwendungen beziehen sich auf die „Außenansicht“ des Typs; die interne Repräsentation bleibt verborgen

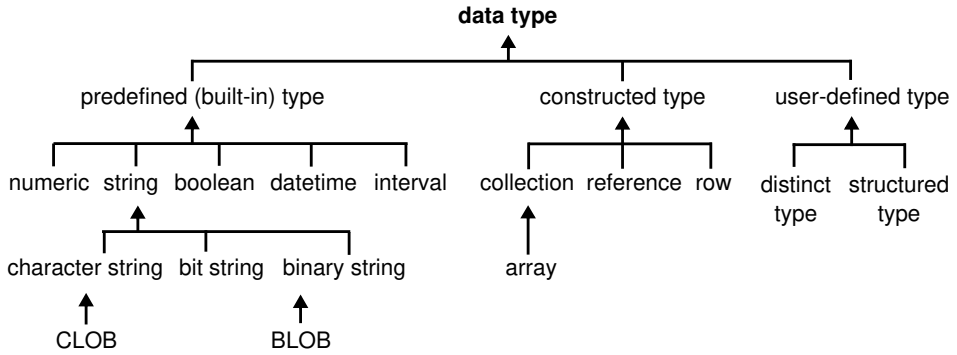
- **Leistungsverhalten**

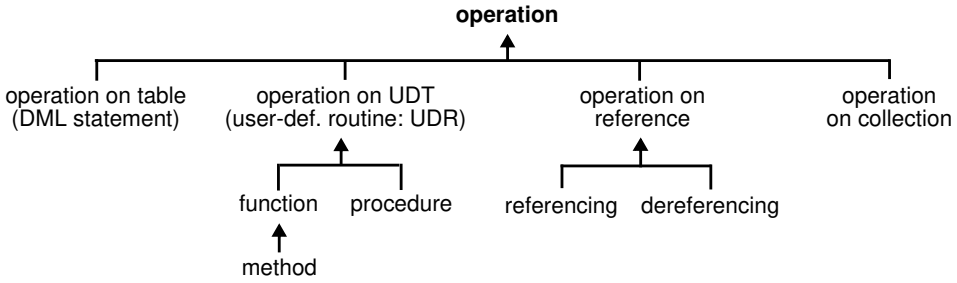
  Möglichkeit zur DBMS-Integration von Typen und Funktionen als „first class citizen“ (volle Verarbeitungsmächtigkeit)

Benutzerdefinierte Typen (3)

- **Vereinfachte Anwendungsentwicklung**
  - Wiederverwendung von Code (Klassenbibliotheken)
  - Überladen und Überschreiben
    (ein einziger Funktionsname für eine Menge von Operationen auf verschiedenen Typen)
  - konsistenter Einsatz von Funktionen/Typen in allen Anwendungen durch ihre „Standardisierung“
  - Änderungen bei Funktionen/Typen sind gekapselt, was ihre Wartung vereinfacht

- **Überblick:** Was ist bereits aus SQL-92 bekannt?



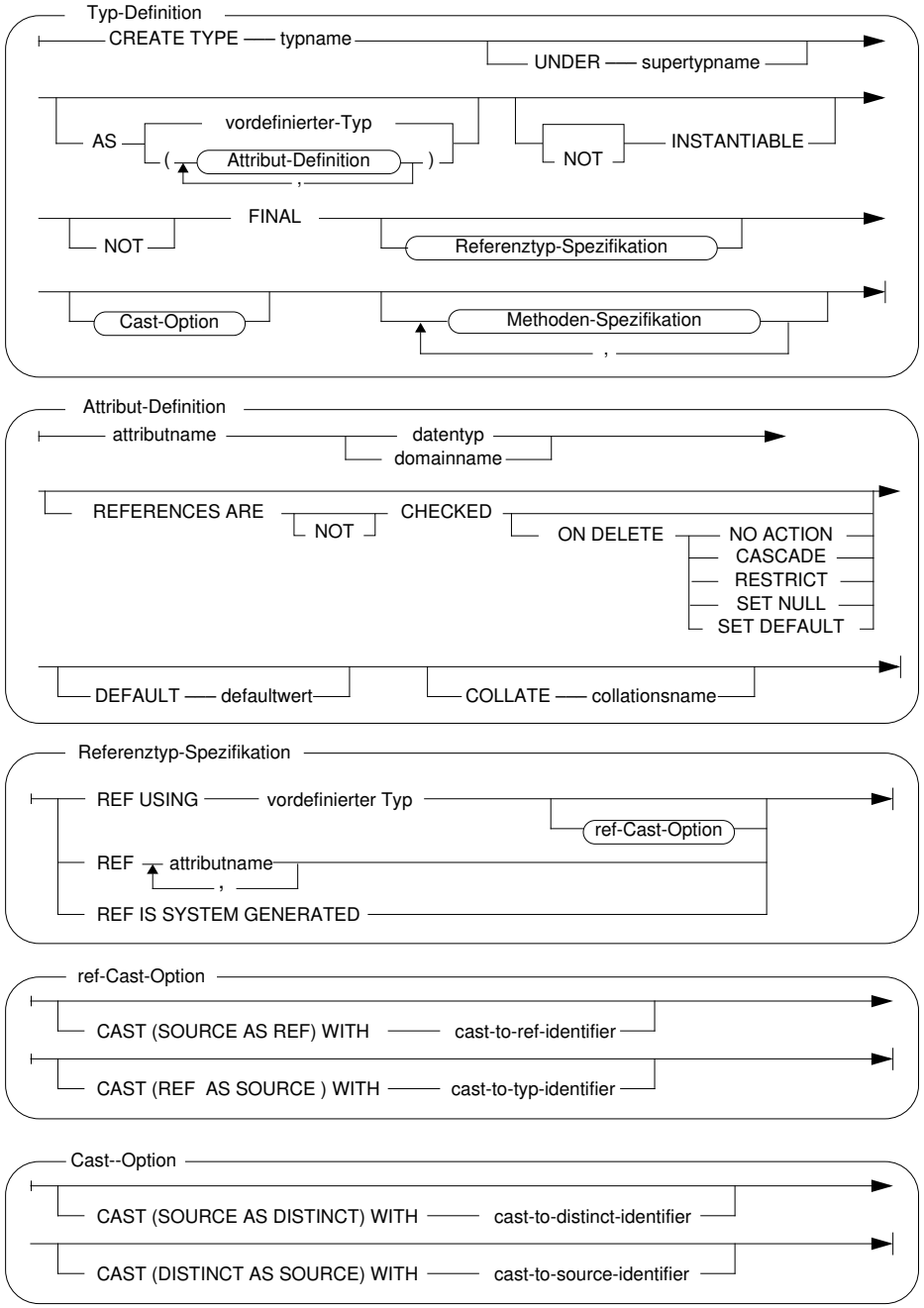


Benutzerdefinierte Typen (4)

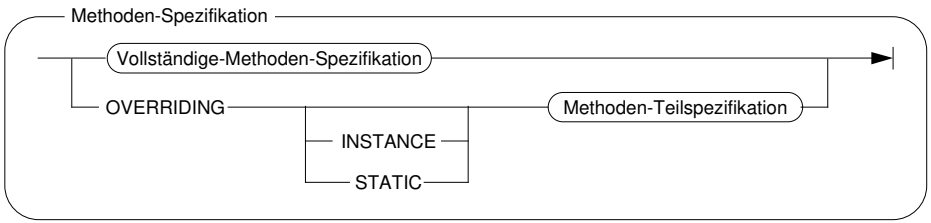
- **Definition des Datentyps**
  - CREATE TYPE definiert einen neuen strukturierten Typ (ADT, UDT): eine benamte Menge von gültigen Datenwerten
  - Festlegung von bis zu 12 Aspekten (ggf. über Defaultwerte) erforderlich
    - Name, Beziehung zu anderen Typen
    - Basistyp bei umbenannten Typen, Attributdeskriptoren bei strukturierten Typen; Grad (Anzahl der Attribute) des Typs
    - Angabe, ob instanzierbar (INSTANTIABLE)
    - Angabe, ob Vererbung und Subtypbildung (bei strukturierten Typen: NOT FINAL, bei umbenannten Typen: FINAL)
    - Casting
    - Deskriptoren für die Methodensignaturen
    - verschiedene Spezifikationen zur Ordnung des Typs (nicht behandelt)

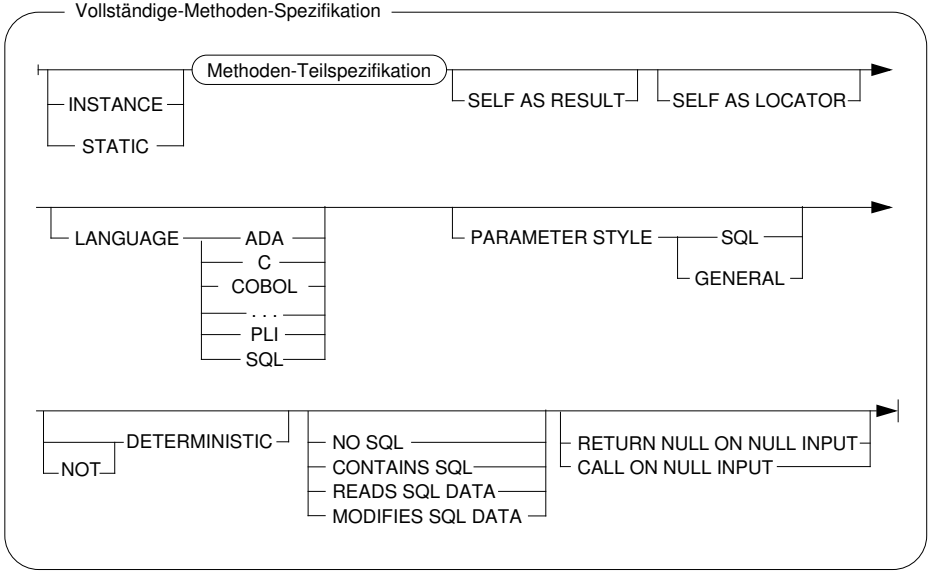
- **Definition von Methoden (und Funktionen)**
  - Name
  - Signatur (d.h. Parameterliste)
  - Ergebnis
  - Implementierung

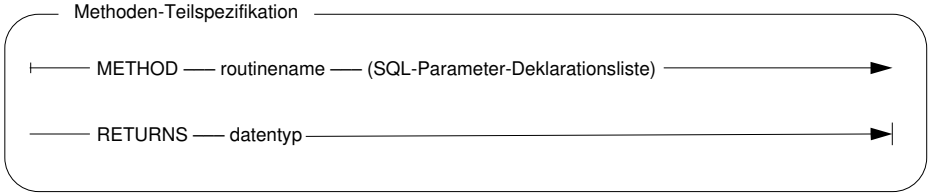
Typdefinition



Methodenspezifikation

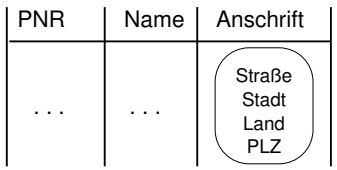




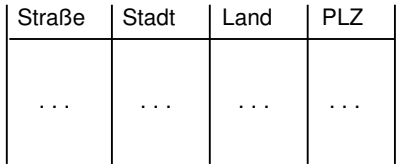


Typklassifikation

- **Benutzerdefinierte Typen (UDT) und Routinen (UDR)**[1]
  - **Umbenannte Typen**
    - fördern *strong typing*, erlauben die Spezifikation von Verhalten
    - auch einzigartige Typen (*distinct types*) genannt
  - **Strukturierte Typen** (ADT-ähnlich, grob: Klassen in OO-Terminologie)
    - definieren bestimmte Objekteigenschaften
    - Kapselung:
      Zustand (interne Datenstrukturen) **+** Verhalten (zugehörige Routinen)
    - Bildung von Subtypen und dabei Nutzung von Vererbung:
      Ohne Typhierarchien keine Tabellenhierarchien!
    - Überladen, Überschreiben, spätes Binden
    - sind verwendbar als **Parametertypen** oder **Attributtypen** oder dienen zur **Definition von Tabellen**
      (die Zeilen mit Objekteigenschaften aufnehmen können)



benamter Spaltentyp

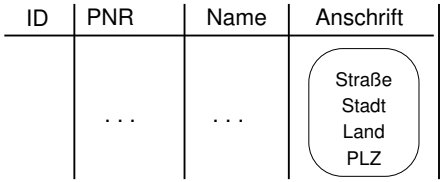


benamter Zeilentyp

1. The features of SQL:1999 can be crudely partitioned into its „relational features“ and its „object-oriented features“. „Relational“ is more appropriately categorized as „features that relate to SQL‘s traditional role and data model“. “Object-oriented features“ are focussed on adding support for object-oriented concepts to the SQL language.

Typklassifikation (2)

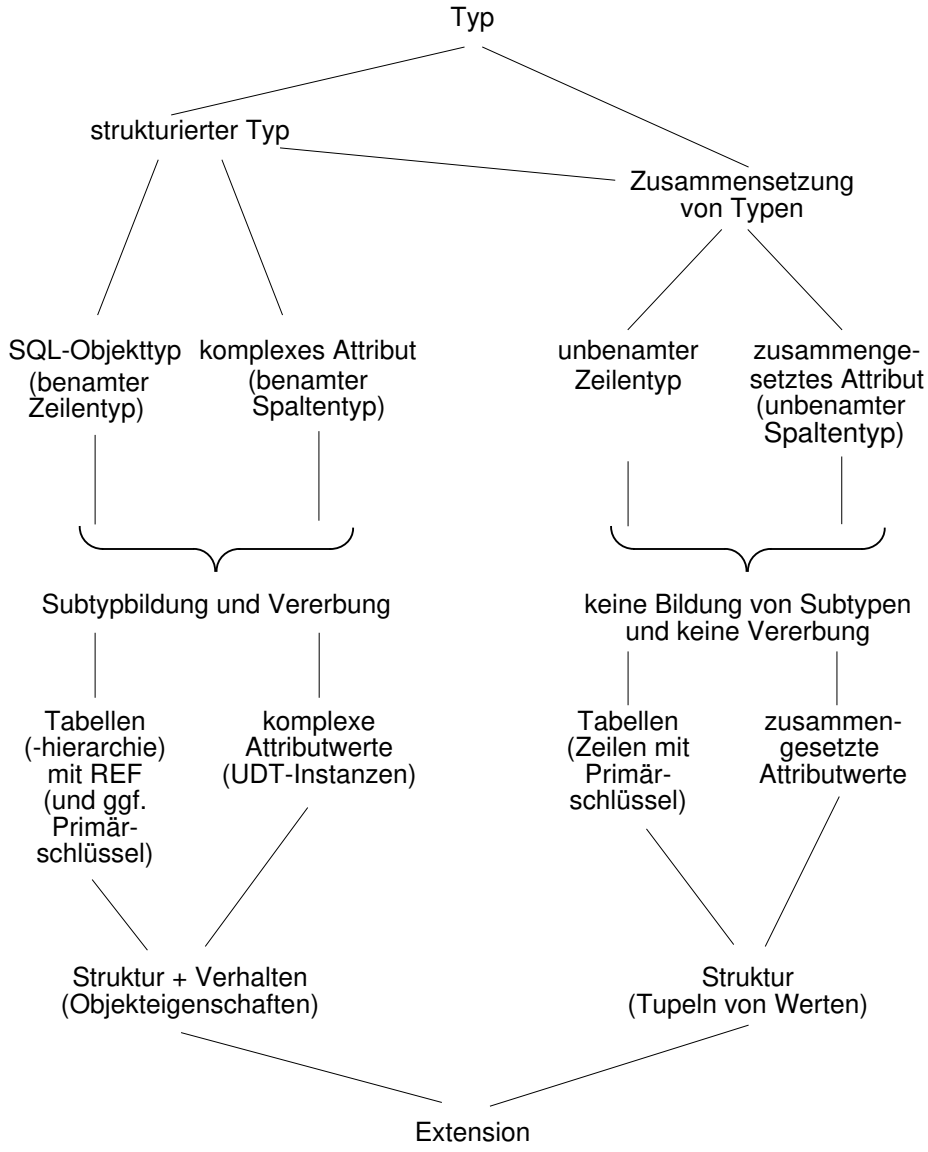
- **Konstruierte Typen (Typkonstruktoren)**
  - **Unbenamte Zeilen- und Spaltentypen**
    - sind aus vordefinierten, benutzerdefinierten und konstruierten Typen zusammengesetzt (haben keine Typnamen: „unnamed row types“)
    - verwendbar als **Datentypen von Zeilen und Spalten**
    - Schachtelung von Zeilen (*nested rows*)
    - aber: keine Objekteigenschaften!
  - **Referenztypen**
    - definiert auf strukturierten Typen
    - eindeutige Identifikation der Zeilen (neben Primärschlüssel)
    - Pfadausdrücke, Navigation
    - erforderlich zur Referenzierung von Zeilen in Tabellen



  - **Kollektionstypen (ARRAY, SET, LIST, BAG)**
    - erlauben neben individuellen Werten auch die Speicherung von zusammengesetzten Werten (*composite values*) als Attributwerte
    - z. Z. nur ARRAY

Typklassifikation (3)

- **Vereinfachte Zusammenfassung:**
  Strukturierter Typ vs. Zusammensetzung von Typen



Umbenannte Typen

- **Einsatz von vordefinierten Typen**

```
CREATE TABLE Verkäufe
    ( Kunden_Nr          INTEGER,
      . . .
      Verkaufspreis      DECIMAL (9,2),
      Einkaufspreis      DECIMAL (9,2));

SELECT    . . ., Verkaufspreis – Einkaufspreis AS Gewinn
FROM      Verkäufe
WHERE     Verkaufspreis > 17,50
```

- **Welche Operationen sind für DECIMAL bereits „eingebaut“?**
  - Zuweisung/Speicherung
  - Ordnung
  - Methoden
  - Casting

- **Umbenannte Typen**
  - verbesserte Modellierung, erhöhte Gewährleistung von Typsicherheit
  - jedoch keine Vererbung und keine Bildung von Subtypen (FINAL), immer INSTANTIABLE
  - Umbenennung des Typs gewöhnlich damit verbunden, ein zu seinem Basistyp unterschiedliches Verhalten zu erreichen

- **Beispiele**

```
CREATE TYPE    GELD     AS    DECIMAL (9,2) FINAL;
CREATE TYPE    ALTER    AS    INTEGER FINAL;
CREATE TYPE    IQ       AS    INTEGER FINAL;
CREATE TYPE    VIDEO    AS    BLOB (100 M) FINAL;
```


Umbenannte Typen (2)

- **Alter1, Alter2 und Iq1 seien Attribute umbenannter Typen:**

```
Alter1   +   20
Iq1      –   Alter1
```

- **Casting-Funktionen**
  - Bei Erzeugung eines umbenannten Typs werden automatisch zwei Casting-Funktionen zum Konvertieren von Werten zwischen dem **umbenannten Typ** und seinem **Basistyp** generiert
  - Systemgenerierte Funktionen

Basistyp	Casting-Funktionen
SMALLINT	smallint
INTEGER	integer
DECIMAL (p,s)	decimal
REAL	real
DOUBLE	double
CHAR (n)	char
VARCHAR (n)	varchar
. . .	

  - Casting-Funktionen für ALTER

```
alter (INTEGER)   returns   ALTER
integer (ALTER)   returns   INTEGER
```

  - Explizites Casting (muß programmiert werden)

Umbenannte Typen (3)

- **Einfaches Arbeiten mit umbenannten Typen**
  verlangt neben Casting-Funktionen
  - Übernahme von Operationen vom Basistyp (Quellenfunktion) und/oder
  - Definition neuer, eigener Operationen

- **Quellenbasierte Funktionen**
  - Eine quellenbasierte Funktion ist eine neue Funktion, die auf einer bereits existierenden Quellenfunktion basiert
  - Quellenfunktionen von INTEGER wie +, –, SUM, AVG können vom Typ ALTER übernommen werden, um entsprechende Operationen direkt auszuführen:

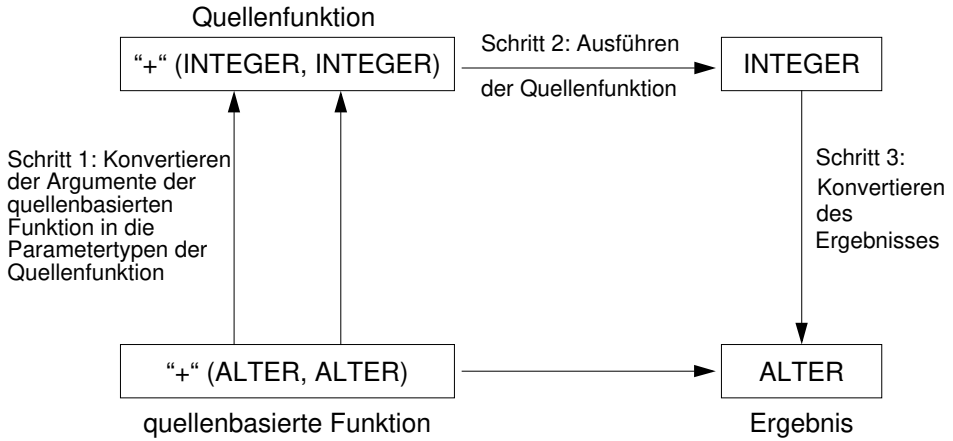
    SELECT Alter1 + alter (lq1) AS Kennziffer

  - Beispiel:

    “+“ (ALTER, ALTER) sei vom Erzeuger des Typs ALTER als quellenbasierte Funktion angelegt:

    CREATE FUNCTION “+“ (ALTER, ALTER)
    RETURNS ALTER
    SOURCE “+“ (INTEGER, INTEGER)

- **Ausführung einer quellenbasierten Funktion**



Umbenannte Typen (4)

- **Benutzung umbenannter Typen**
  - WHERE lq1 > 150 ?
  - SET lq1 = lq1 · 2 ?
  - WHERE Alter1 + lq1 > Alter2 ?

Umbenannte Typen (5)

- **Anwendungsbeispiel**

```
CREATE TYPE EURO AS DECIMAL (9,2) FINAL;
CREATE TYPE US_DOLLAR AS DECIMAL (9,2) FINAL;

CREATE TABLE E_Verkäufe                CREATE TABLE US_Verkäufe
    (  Kunden_Nr      INTEGER,             (  Kunden_Nr      INTEGER,
       Vertrags_Nr    INTEGER,                Vertrags_Nr    INTEGER,
       Gesamt         EURO);                  Gesamt         US_DOLLAR);

ALTER TABLE E_Verkäufe
ADD Bonus EURO
```

- **Hauptaspekt ist Typsicherheit:**
  Es läßt sich Typkorrektheit und Typverhalten garantieren!

```
SELECT     E.Kunden_Nr, E.Gesamt + US.Gesamt AS Gesamt
    FROM   E_Verkäufe E, US_Verkäufe US
    WHERE  E.Vertrags_Nr = US.Vertrags_Nr
    AND    E.Gesamt > US.Gesamt                    Fehler!!!
```

➥ **Euro und US_Dollar können nicht miteinander addiert und verglichen werden!**

- **Ist damit das Problem gelöst?**
  E.Gesamt + US.Gesamt

  - Hier sind offensichtlich spezielle Konversionsroutinen, die Semantik berücksichtigen, erforderlich!
    US_Dollar_To_Euro (US.Gesamt)

➥ **CAST-Anweisung: explizite Programmierung der Casting-Funktion**

Benutzerdefinierte Typen und Routinen

- **Ziel: „Objektorientierung[1] für die Spalten und Zeilen von Tabellen“**
  - komplexe Strukturen für Objekte definieren
  - Verhalten (Operatoren, Funktionalität) für Objekte definieren
  - Mächtige SQL-Anfragen, die die Semantik von Objekten „verstehen“

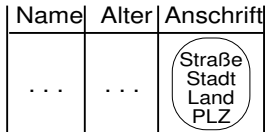
- **Definition von neuen Typen**
  - benannte, benutzerdefinierte Typen
    mit Verhalten (Routinen) und gekapselter interner Struktur (Attribute)
  - keine Unterscheidung zwischen ADTs und ROW-Types
    (wie bei früheren SQL-Versionen)

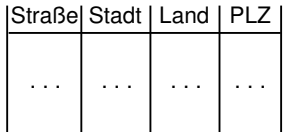
  ➥ SQL:1999 vereinigt zwei Typkonzepte in ein einheitliches Typkonzept:
  strukturierte Typen!

- Strukturierte Typen als „**white box**“-Definition:
  Interne Struktur ist in SQL-Termen definiert

```
CREATE TYPE Adresse AS
    (  Straße      CHAR (30),
       Stadt       CHAR (20),
       Land        CHAR (2),
       PLZ         INTEGER) NOT FINAL;
```

➥ **als Datentyp für Attribute oder als Zeilentyp verwendbar**





1. In spite of certain characteristics like type hierarchies, encapsulation, etc., instances of SQL:1999 structured types are simply values. Such values may be more complex than an instance of INTEGER, but it is still a value without any identity other than that provided by its value.

Benutzerdefinierte Typen und Routinen (2)

- Strukturierte Typen als „**black box**“-Definition:
  interne Struktur ist außerhalb von SQL definiert

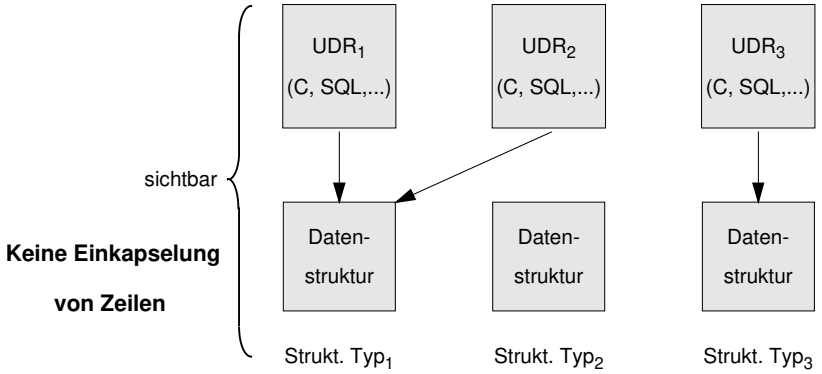
  CREATE TYPE Adresse **AS**
  (**internal length** = 56 **VARYING**,
  input = Adreßeingabe,
  output = Adreßausgabe)

  ➥ „black box“-ADTs sind kein Teil von SQL:1999

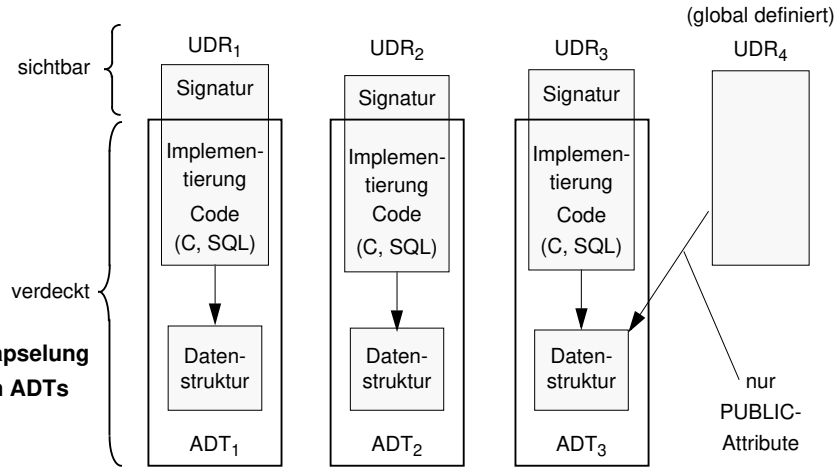
- **Eigenschaften**
  - Plattformabhängigkeit
  - DBS „versteht“ nicht die Objektsemantik
    - keine Unterstützung bei der Suche
    - keine Maßnahmen zur Integritätssicherung
  - oft mit Hilfe von LOBs implementiert

Benutzerdefinierte Typen und Routinen (3)

- **UDRs für strukturierte Typen („white box“-ADTs)**



- **UDRs für „black box“-ADTs** (nicht in SQL:1999, aber in spez. Systemen)



Strukturierte Typen

- **Beispiel:**

```
CREATE TYPE Adresse AS
  ( Straße      CHAR (30),
    Stadt       CHAR (20),
    Land        CHAR (2),
    PLZ         INTEGER) NOT FINAL;

CREATE TYPE Bitmap AS BLOB (10M) FINAL;

CREATE TYPE Immob_t AS
    ( Besitzer    REF (Person),
      E-Preis     GELD,
      V-Preis     GELD,
      . . .
      Ort         Adresse,
      Ansicht     Bitmap) INSTANTIABLE NOT FINAL
      REF IS SYSTEM GENERATED
      METHOD Gewinn RETURNS GELD;
```

- **Strukturierte Typen sind überall in SQL einsetzbar, wo vordefinierte Typen benutzt werden können**
  - Typen von Attributen anderer strukturierter Typen
  - Typen von Parametern von Funktionen, Methoden und Prozeduren
  - Typen von SQL-Variablen
  - Typen von Wertebereichen oder Spalten in Tabellen

Strukturierte Typen (2)

- **Automatisch generierte Funktionen (Methoden) bei CREATE TYPE**
  - **Konstruktor-Methode** erzeugt Instanzen vom strukturierten Typ, initialisiert mit Defaultwerten:
    Adresse () → Adresse
  - **Observer- und Mutator-Methoden** (O, M) manipulieren einzelne Attribute. Sie können
    - als Grundfunktion zur Implementierung von zusätzlichem Verhalten eingesetzt oder
    - an der Objektschnittstelle (Signatur) zur Verfügung gestellt werden
    - nicht überladen werden.

    (a sei Adressen-Variable/Ausdruck)

a.Straße ()	→	CHAR (30)
a.Stadt ()	→	CHAR (20)
a.Land ()	→	CHAR (2)
a.PLZ ()	→	INTEGER
a.Straße (CHAR (30))	→	Adresse
a.Stadt (CHAR (20))	→	Adresse
a.Land (CHAR (2))	→	Adresse
a.PLZ (INTEGER)	→	Adresse

- **Schachtelung von strukturierten Typen ist möglich**

Strukturierte Typen (3)

- **Alle Methoden** haben denselben Namen wie die Attribute, denen sie zugeordnet sind
  - Stadt
  - Stadt ()
  - Stadt ('KL')

- **Definition von weiteren Methoden/Funktionen (z.B. Gewinn)**
  - in SQL oder
  - in einer externen Programmiersprache (wie C)

- **Allgemeine Unterscheidungsregeln bei Signaturen**
  - zwei Routinen mit demselben Namen können in derselben Namensklasse (Schema) sein
  - Routinen sind durch ihre Kategorie unterscheidbar: Prozedur, Funktion, O-Methode, M-Methode, Instanz-Methode, statische Methode
  - Routinen in derselben Kategorie sind unterscheidbar durch Parameteranzahl und deklarierte Typen in der Parameterliste

Strukturierte Typen (4)

- **Wie werden Instanzen von UDTs verarbeitet?**

```
CREATE TABLE Adressenliste
    ( LfdNr      INTEGER PRIMARY KEY,
      Ort        Adresse);
```

- **Einfügen einer Zeile**
  - Vorbereitung im Wirtsprogramm

```
BEGIN
  DECLARE a Adresse;
  SET     a = Adresse ();
  SET     a = a.Straße ('A-Straße');
  . . .
  SET     a = a.PLZ (67653);
  INSERT INTO Adressenliste VALUES (123, a);
END;
```

  - alternativ: direktes Einfügen

```
INSERT INTO Adressenliste
  VALUES  ( :lfdnr, NEW Adresse ('A-Straße, ..., 67653');
```

- **Aktualisieren von Straße**

```
UPDATE    Adressenliste
  SET     Ort = Ort.Straße ('B-Straße')
  WHERE   LfdNr = :x;
```

Strukturierte Typen (5)

- **Zugriff mit O-Methoden**

```
SELECT   a.Ort.Stadt (), a.Ort.PLZ ()
INTO     :x, :y
FROM     Adressenliste a
WHERE    a.Ort.PLZ > 80000;
```

- **Allgemeine Regeln**
  - Punktnotation ist zum Aufruf von Methoden erforderlich
  - Methoden ohne Parameter benötigen keine „()“
  - Punktnotation unterstützt den „navigierenden“ Zugriff bei geschachtelten strukturierten Typen über mehrere Ebenen (a.b.c.d.e)

Strukturierter Typ als SQL-Objekttyp

- **Ziel:**
  Objektorientierung für die Zeilen von Tabellen

- **Strukturierter Typ**
  dient zur Typisierung der in einer Tabelle gespeicherten Objekte
  - Aus Attributen des strukturierten Typs werden „Spalten“ der darauf definierten Tabelle (*typed table*)
  - Eine **Extra-Spalte** definiert (eindeutige) REF-Werte für die Zeilen
    - systemgeneriert: REF IS SYSTEM GENERATED
    - benutzergeneriert: REF USING <vordefinierter Typ>
    - abgeleitet: REF <Attributliste> (UNIQUE NOT NULL)
  - Explizite Definition eines strukturierten Typs erlaubt seine Verwendung bei mehreren TABLE-Definitionen
  - Konzept ist wichtig für Tabellenhierarchien

- **Beispiel:**

```
CREATE TYPE Immob_t AS (                          // Strukturierter Typ

    ( Besitzer    REF (Person),
      E-Preis     GELD,
      V-Preis     GELD,
      . . .
      Ort         Adresse,
      Ansicht     Bitmap) INSTANTIABLE NOT FINAL
      REF IS SYSTEM GENERATED
      METHOD Gewinn RETURNS GELD;


CREATE TABLE Immobilien OF Immob_t          // Tabelle mit Typbindung
    (REF IS OID SYSTEM GENERATED)   // selbst-referenzierendes Attribut
```

Verarbeitung von strukturierten Typen

- **Durch Typschachtelung können „Mischformen“ auftreten**

```
CREATE TABLE Immobilien OF Immob_t
    (REF IS OID SYSTEM GENERATED);
```

OID	Besitzer	E-Preis	V-Preis	...	Ort	Ansicht

- **Erzeugen von Zeilen**

```
DECLARE im Immob_t
    SET im = Immob_t();                    /*  Aufruf des Konstruktors  */
    . . .
    SET im.E-Preis = geld (500.000,00);
    /* zulässig für im = im.E-Preis (geld (500.000,00))                 */

    SET im.Ort.Stadt = 'KL';
    /* zulässig für im = im.Ort.Stadt('KL')                             */

    SET im.V-Preis = im.E-Preis() + geld (100.000,00);      /* „+“ ? */
    . . .
    INSERT INTO Immobilien
        VALUES (im);
```

- **Direktes Einfügen**

```
INSERT INTO Immobilien
    VALUES (:x, geld(500.000,00), geld(500.000,00) + geld(100.000,00),
    . . ., NEW Adresse ('A-Straße, KL, RLP, 67653'), :y);
```


Verarbeitung von strukturierten Typen (2)

- **Einsatz von Methoden und Funktionen**
  - Sie können überall aufgerufen werden, wo in SQL Skalarwerte erlaubt sind
  - Benutzerdefinierte Methoden/Funktionen sind (in SQL oder C) zu programmieren. DBMS verwaltet sie in speziellen Bibliotheken.

- **Beispiel für getypte Tabelle**
  Gen_Adr, Euro und Contains sind als benutzerdefinierte Funktionen verfügbar

```
UPDATE Immobilien
SET  V-Preis = geld (1.2 * decimal (E-Preis))
WHERE Ort.Land() = 'RLP';

SELECT E-Preis
FROM Immobilien
WHERE Ort = Gen_Adr (Adresse(), 'A-Straße, KL, RLP, 67653');

SELECT    Euro (V-Preis), Ort.Stadt ()
FROM      Immobilien
WHERE     Ort.PLZ() = 67653
  AND     Contains (Ansicht, 'Seeufer');
```

- **Bleibt der Typ der Tabelle bei Anfragen/Sichten erhalten?**
  - Anfragen auf getypen Tabellen greifen auf **Attribute** (Spalten) zu
  - Änderungsanweisungen auf getypten Tabellen modifizieren Attribute
  - Ergebnistyp von Anfragen?

Verarbeitung von strukturierten Typen (3)

- **Beispiele für strukturierten Spaltentyp**

```
CREATE TABLE ImmoListe AS
      ( LfdNr      INTEGER PRIMARY KEY,
        Immobilie Immob_t);
```

- **Beispiele**

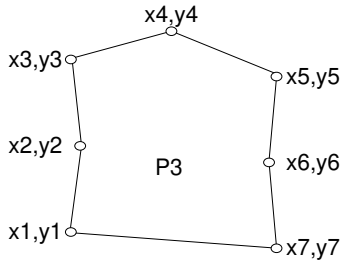
```
DECLARE im Immob _t;
. . .
/* Erzeugen eines im-Wertes */
INSERT INTO ImmoListe
      VALUES (123, im);

UPDATE ImmoListe
      SET Immobilie().V-Preis() = geld (1.2 * decimal(Immobilie(). E-Preis()))
      WHERE LfdNr = 123;


SELECT     Immobilie().E-Preis()
FROM       ImmoListe
WHERE      Immobilie().Ort() = Gen_Adr(Adresse(), 'A-Straße, ...,67653');


SELECT     Euro(Immobilie().V-Preis()),
           Immobilie().Ort().Stadt()
FROM       ImmoListe
WHERE      Immobilie().Ort().PLZ() = 67653
   AND     Contains (Immobilie().Ansicht(), 'Seeufer');
```


Strukturierte Typen: Beispiele

- **Erinnerung:**
  Darstellung eines Polygons im Relationenmodell

```
CREATE TABLE Vieleck
   (Id      CHAR(5)      NOT NULL,
   PktNr    INTEGER      NOT NULL,
   X        DECIMAL      NOT NULL,
   Y        DECIMAL      NOT NULL,
      PRIMARY KEY (Id, PktNr));
```

Polygon P3 und seine relationale Darstellung



Vieleck			
Id	PktNr	X	Y
• • •			
P3	1	x1	y1
P3	2	x2	y2
P3	3	x3	y3
P3	4	x4	y4
P3	5	x5	y5
P3	6	x6	y6
P3	7	x7	y7
• • •			

➥ Wie werden Operationen und Integritätsbedingungen auf Polygonen realisiert?

- **Jetzt als strukturierte Typen definierbar:**
  Punkte, Linien, Flächen, Graphen, Rasterdaten usw. inklusive Verhalten

➥ **Objekt-relationales Datenmodell** bietet erhebliche Vorteile!

Strukturierte Typen: Beispiele (2)

- **Definition des strukturierten Typs Punkt:**

  CREATE TYPE Punkt **AS** (
  X_Koord **DECIMAL** (7,3),
  Y_Koord **DECIMAL** (7,3)) **NOT FINAL**;

  - Alle Attribute sind nur über Methoden zugreifbar:
    mehr physische Datenunabhängigkeit
  - Automatisch bereitgestellte Methoden zum Lesen und Ändern (O, M)

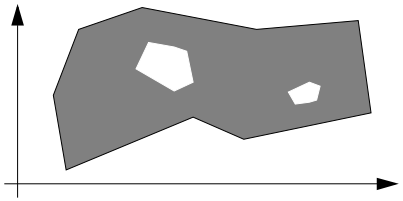
  METHOD X_Koord () **RETURNS** (**DECIMAL** (7,3))

  METHOD X_Koord (**DECIMAL** (7,3)) **RETURNS** Punkt

  METHOD Y_Koord () **RETURNS** (**DECIMAL** (7,3))

  METHOD Y_Koord (**DECIMAL** (7,3)) **RETURNS** Punkt

- **Nutzung des Typs Punkt[1]:**



  CREATE TYPE Polygon **AS** (
  Nummer **INTEGER**,
  Rand **LIST** (Punkt),
  Löcher **SET** (**LIST** (Punkt)))
  INSTANTIABLE NOT FINAL

  METHOD Umfang () **RETURNS DECIMAL**,

  METHOD FLäche () **RETURNS DECIMAL**,

  METHOD Enthält (Punkt) **RETURNS BOOLEAN**,

  METHOD Schneidet (Polygon) **RETURNS BOOLEAN**,

  . . .

- **Separierung von Signatur und Implementierung**
  CREATE METHOD ... BEGIN <Programmcode> **END**;

1. In SQL:1999 müßte der einzige verfügbare Kollektionstyp ARRAY gewählt werden

Strukturierte Typen: Beispiele (3)

- **Ziel**
  - Direkte Modellierung und Speicherung komplexer Daten in Tabellen
  - Erweiterte Infrastruktur für SQL/MM (Multimedia)

- **Tabellen mit strukturierten Typen als Datentyp für Attribute**

  CREATE TABLE Flurstücke (
  Nummer **INTEGER** PRIMARY KEY,
  Geometrie **Polygon**,
  Eigentümer **CHAR** (50));

  ➥ Flurstücke ist Tabelle ohne benannten Typ!

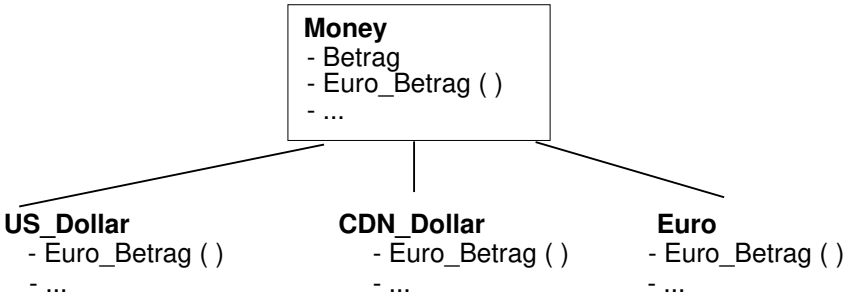
- **Verwendung von Operationen auf strukturierten Typen in SQL-Anfragen**

```
SELECT   F.Nummer                          // Welche Flurstücke > 20000 qm
FROM     Flurstücke F                      // gehören der LH München?
WHERE    F.Eigentümer = 'LH München'
         AND F.Geometrie.Fläche () > 20 000,00;
```

```
SELECT   F1.Nummer, F2.Nummer              // Gibt es überlappende Flurstücke?
FROM     Flurstücke F1, Flurstücke F2
WHERE    F1.Geometrie.Schneidet (F2.Geometrie);
```

Strukturierte Typen und Vererbung

- **Strukturierter Typ kann Subtyp eines anderen strukturierten Typs sein**
  - Vererbung von Attributen und Verhalten (Methoden) vom Supertyp
  - Überschreibung von Verhalten
  - CREATE TYPE **Money** AS (Betrag ...) NOT INSTANTIABLE NOT FINAL;
- Bisher keine Unterstützung für **Mehrfachvererbung!** (➥ SQL4)



- **Substituierbarkeit:**
  Jede Zeile kann eine Instanz eines verschiedenen Subtyps haben

```
CREATE TABLE  Immobilien_Info
  (Preis      Money,
   Besitzer   CHAR (40),
   Grundstück Adresse)
```

Preis	Besitzer	Grundstück
<US_Dollar> Betrag: 100.000	'S.Weiss'	<Adresse>
<CDN_Dollar> Betrag: 400.000	'Dr.W.Gruen'	<Adresse>
<Euro> Betrag: 150.000	'D.Schwarz'	<Adresse>

```
SELECT  Besitzer, Euro_Betrag(Preis)
FROM    Immobilien_Info
WHERE   Euro_Betrag(Preis) <
        Euro (500.000)
```

- **Dynamische Auswahl** (dynamic *dispatch* nur bei Methoden!) und **spätes Binden** der Werte von **Preis** auf der Basis des Typs **Money**

Typ- und Tabellenhierarchie - Motivation

- **Tabelle Ortsstraßen**
  CREATE TABLE Ortsstraßen (
  Name **CHAR** (40) PRIMARY KEY,
  Länge **DECIMAL** (9,2),
  Breite **DECIMAL** (5,2),
  O_Bez **Orte**);
- **Einfügen von Zeilen**
  INSERT INTO Ortsstraßen **VALUES** ('Mozartstr', 3,25, 8,75, 'München');
- Ortsstraßen

Name	Länge	Breite	O_Bez
Schillerstr	2,50	7,50	Köln
Mozartstr	3,25	8,75	München

- **Tabelle Autobahnen**
  CREATE TABLE Autobahnen (
  Name **CHAR**(40) PRIMARY KEY,
  Länge **DECIMAL** (9,2),
  Breite **DECIMAL** (5,2),
  Gebühr **Money**);
- **Einfügen von Zeilen**
  INSERT INTO Autobahnen **VALUES** (A8, 564,50, 20,10, US_Dollar(10)
- Autobahnen

Name	Länge	Breite	Gebühr
A6	324,00	18,20	<Euro> 20
A8	564,50	20,10	<US_Dollar> 10

- **Anfrage:** Suche alle Straßen mit einer Breite größer als 5,50m.

Typ- und Tabellenhierarchie

- **Ziel:**
  Modellierung von Hierarchien von Objektmengen

- **Ohne Typhierarchien keine Tabellenhierarchien!**
  - Strukturierter Typ kann andere strukturierte Typen als Subtypen haben
  - Tabelle mit Typbindung (typed table) kann Subtabellen haben

- **Eigenschaften**
  - Subtabellen erben Attribute, Constraints, Trigger usw. von der Supertabelle
  - Anfragen erhalten eine höhere Ausdrucksmächtigkeit
  - Anfragen auf der Supertabelle arbeiten auch auf den Subtabellen

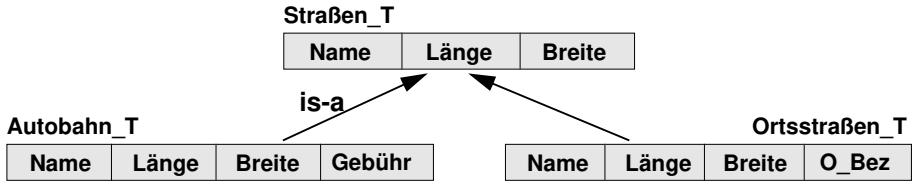
- **Aspekte der Zugriffskontrolle**
  - Anfragen, die Subtabellen referenzieren, benötigen SELECT-Privileg
  - SELECT-Privileg mit WITH HIERARCHY OPTION auf Supertabelle gewährt Zugriff auf alle zugehörigen Subtabellen

Typ- und Tabellenhierarchie - Beispiele (1)

- **Strukturierter Typ Straßen_T**
  CREATE TYPE Straßen_T **AS** (
  Name **CHAR** (40),
  Länge **DECIMAL** (9,2),
  Breite **DECIMAL** (5,2)) **NOT FINAL . . .**;

- **Subtypen**
  CREATE TYPE Autobahn_T **UNDER** Straßen_T(Gebühr Money) **NOT FINAL ...**;
  CREATE TYPE Ortsstraßen_T **UNDER** Straßen_T(O_Bez Orte) **NOT FINAL ...**;

- **Typhierarchie**



- **Tabellenhierarchie**
  - **CREATE TABLE** Straßen *// Supertabelle Straßen*
    OF Straßen_T (**PRIMARY KEY** Name, . . .);
  - **CREATE TABLE** Autobahnen *// Subtabelle Autobahnen*
    OF Autobahn_T **UNDER** Straßen;
  - **CREATE TABLE** Ortsstraßen *// Subtabelle Ortsstraßen*
    OF Ortsstraßen_T **UNDER** Straßen;

- **Erzeugen von Privatstraßen (vom Typ Ortsstraßen_T)?**

Wie sieht die Tabellenhierarchie aus?

Typ- und Tabellenhierarchie - Beispiele (2)

- **Einfügen von Zeilen in Straßen, Autobahnen, Ortsstraßen**

  INSERT INTO Ortsstraßen **VALUES** (‘Mozartstr’, 3,25, 8,75, ‘München’)
  INSERT INTO Autobahnen **VALUES** (‘A8’, 564,50, 20,10, US_Dollar(10))
  INSERT INTO Straßen **VALUES** (‘Schillerweg’, 7,75, 5,00)

- **Wie werden die Zeilen gespeichert?**

  Hausklassenmethode

  Straßen (**INSTANTIABLE**))

OID	Name	Länge	Breite
O21	Schillerweg	7,75	5,00

  Autobahnen

OID	Name	Länge	Breite	Gebühr
O08	A6	324,00	18,20	<Euro> 20
O71	A8	564,50	20,10	<US_Dollar> 10

  Ortsstraßen

OID	Name	Länge	Breite	O_Bez
O12	Schillerstr	2,50	7,50	Köln
O12	Mozartstr	3,25	8,75	München

- **Anfrage:** Automatische Berücksichtigung der ganzen Tabellenhierarchie

  Suche alle Straßen mit einer Breite größer als 5,50m.

  SELECT * **FROM** Straßen **WHERE** Breite > 5,50

- **Änderung:** Automatische Änderung in den korrespondierenden Super- und Subtabellen

  UPDATE Straßen **SET** Breite = 10,00 **WHERE** Name **LIKE** ‘Schiller%‘

  oder

  UPDATE Ortsstraßen **SET** Breite = 10,00 **WHERE** O_Bez = München

Konstruierte Typen: Tupeltyp

- **Ziel: Einfacher Umgang mit zusammengesetzten Werten**

- **Tupeltyp (*ROW Type*) als Datentyp von zusammengesetzten Attributen**
  - Zusammengesetztes Attribut kann in **einer** Spalte gespeichert werden
  - Tupel kann als **ein** Argument an Routinen und als Rückgabewert von Funktionen dienen
  - entspricht Record-Typen in Programmiersprachen

- **Tupeltyp** (früher: *unnamed row type*)
  - Definition von geschachtelten Tabellenstrukturen
  - spezielle Operationen:
    Konstruktor, Zuweisung, Vergleich von zusammengesetzten Werten

- **Beispiel:**

```
CREATE TYPE Straßen_T AS (                          // Strukturierter Typ
    Name          VARCHAR (40),
    Verwaltung    ROW (Bezeich VARCHAR (20),        // Tupeltyp
                       Stadt   VARCHAR (30)),
    Geometrie     Polygon,
    Referenzpunkt ROW ( Rechts  GK_Koordinate,      // Tupeltyp
                        Hoch    GK_Koordinate,
                        Höhe    NN_Höhe)) NOT FINAL . . .;
```

Referenztypen

- **Ziel: Verweise (Referenzen) auf andere Objekte**
- **Eigenschaften**
  - Referenzen repräsentieren Beziehungen zwischen Objekten
  - Ein Objekt kann von vielen anderen referenziert werden
  - Referenz verweist immer auf eine Zeile (Objekt) einer getypten Tabelle, nicht auf ein Objekt in einer Spalte
- **Verwendung**
  - Referenztypen können mit Zeilentypen kombiniert werden
  - Sie erleichtern die Modellierung von Beziehungen zwischen Typen
  - Wertebereich von Referenzen kann durch die SCOPE-Klausel eingeschränkt werden
  - Bei mehreren Tabellendefinitionen mit gleichem strukturierten Typ lassen sich für Referenzen unterschiedliche Gültigkeitsbereiche festgelegen
- **Beispiel**

```
CREATE TYPE Kunden_T AS (
    KNR          INTEGER,
    Name         CHAR (50),
    Anschrift    Adresse)
    NOT FINAL REF USING INTEGER;                    //Strukturierter Typ

CREATE TABLE Kunden OF Kunden_T (
    PRIMARY KEY KNR,
    REF IS kid USER GENERATED);     // selbst-referenzierendes Attribut

CREATE TABLE Privat_Kunden OF Kunden_T (
    PRIMARY KEY KNR,
    REF IS pid USER GENERATED);     // selbst-referenzierendes Attribut
```


Referenztypen (2)

- **Beispiel** (Fortsetzung)

```
CREATE TYPE Konto_T AS (
    Konto_Nr     INTEGER,
    Kunde        REF (Kunden_T),   // Verweis auf den zugehörigen Kunden
    Typ          CHAR (1),
    Eröffnet     DATE,
    Zinsrate     DOUBLE PRECISION,
    Kontostand   DOUBLE PRECISION
    ) NOT FINAL REF Konto_Nr;

CREATE TABLE Konto OF Konto_T (
    PRIMARY KEY Konto_Nr,
    REF IS koid DERIVED,
    Kunde WITH OPTIONS SCOPE Kunden);
```

Kunden

kid	KNR	Name	Anschrift
	. . .	. . .	Straße Stadt Land PLZ

Privatkunden

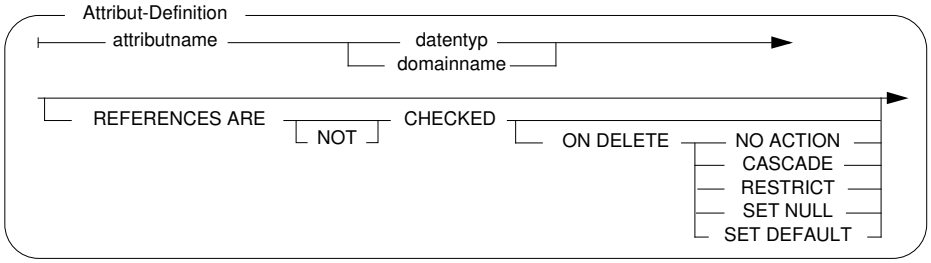
pid	KNR	Name	Anschrift
	. . .	. . .	Straße Stadt Land PLZ

Konto

koid	Konto_Nr	Kunde	Typ	. . .	Kontostand
	. . .	. . .	. . .	. . .	. . .

Referenzen und Pfadausdrücke

- **Referenzen können nur auf „top-level rows“ verweisen**
  - Referenzwert ändert sich nicht, solange die entsprechende Zeile existiert
  - Referenzwerte werden nicht wiederbenutzt
  - Referenzwerte sind innerhalb der DB eindeutig
- **Es lassen sich „referentielle Aktionen” spezifizieren**



- **Referenzen können zur Spezifikation von Pfadausdrücken verwendet werden**

  ➥ Nur „scoped”-Referenzen können (mit Operator „->“) dereferenziert werden

```
SELECT   a.Konto_Nr, a.Kunde -> Name
FROM     Konto a
WHERE    a.Kunde -> Anschrift.Stadt = 'München'
         AND a.Kontostand > 1.000.000,00;
```

- **Autorisierung des Zugriffs wird nach dem SQL-Autorisierungsmodell überprüft**
  - Benutzer muß SELECT-Rechte auf den entsprechenden Attributen der Tabelle (Kunden) besitzen
  - Autorisierung kann zur Übersetzungszeit geprüft werden, falls eine SCOPE-Klausel spezifiziert wurde

Kollektionstypen

- **Warum Nutzung von Kollektionen?**
  - Modellierung von Attributen mit Kollektionswerten (*repeating groups*, *NF^2 tables*)
  - Wichtiger Baustein zur Modellierung: Oft Einsatz von Funktionen (prozeduraler Zugriff) erforderlich
  - häufig in Standard-Typ-Bibliotheken verwendet, z. B. SQL/MM (Fulltext, Spatial)
- **Kollektionstypen**
  - Array
  - Mengen
  - Listen (Sequenzen)
  - Multimengen (Bag)
- **Beispiel**

```
CREATE TABLE Stadtviertel (
  Name        CHAR (40) PRIMARY KEY,
  Geometrie   Polygon,
  Baublöcke   LIST (INTEGER),     // Nummern der enthaltenen Baublöcke
  Fläche      DECIMAL (10, 2),
  Straßen     SET (REF (Straßen_T)));          // Straßen im Stadtviertel
```

- **Wie geht man mit (großen) Kollektionen um?**
  - Laden, Ändern, Suchen, Transportieren?
  - Transformation von Kollektionen zu Tabellen sowie Anfragen darauf möglich

Beispiele - Kollektionstypen (1)

- **SQL99 unterstützt z. Z. nur ARRAY!**
  - z. Z. nur Definition „one-dimension, fixed-max-length“ möglich
  - Elementzugriff auf Array (Positionszugriff) und deklarative Anfragen auf Array-Elemente
  - Andere Operationen: Cardinality, Catenation, Vergleich, CAST, . . .

- **Tabelle Stadt mit ARRAY-wertigem Attribut S_Viertel**

CREATE TABLE Stadt (
Name **CHAR** (40) PRIMARY KEY,
S_Viertel **CHAR** (40) **ARRAY** [20],
Land **VARCHAR** (30));

- **Einfügen von Zeilen in Tabelle Stadt**

INSERT INTO Stadt (Name, S_Viertel, Land) **VALUES**
(‘Kaiserslautern’, **ARRAY** [‘Betzenberg’, ‘Uni-Wohngebiet’], ‘Rheinland-Pfalz’);

Stadt

Name	S_Viertel	Land
Mannheim	Käfertal	Baden-Württemberg
	Neckarau	
	Quadrate	
Ludwigshafen	BASF	Rheinland-Pfalz
	Fachhochschule	

Beispiele - Kollektionstypen (2)

- **Anfrage** (mit Positionszugriff)**:**
Suche von allen Städten in Rheinland-Pfalz das erste aufgelistete Stadtviertel.

SELECT Name, S_Viertel [1] **AS** Stadtteil
FROM Stadt
WHERE Land = ‘Rheinland-Pfalz’;

Ergebnistabelle

Name	Stadtteil

- **Deklarative Anfragen auf Array-Elemente:**
  - implizite Umwandlung von Arrays in Tabellen
  - Element-Auswahl über Inhalt oder Position
  - Unnesting

- **Anfrage:**
Suche von allen Städten in Rheinland-Pfalz alle aufgelisteten Stadtviertel.

SELECT s.Name, v.Stadtteil
FROM Stadt **AS** s, **UNNEST** (s.S_Viertel) **AS** v (Stadtteil)
// Umwandlung des Arrays in eine Tabelle
WHERE Land = ‘Rheinland-Pfalz’;

Ergebnistabelle

Name	Stadtteil

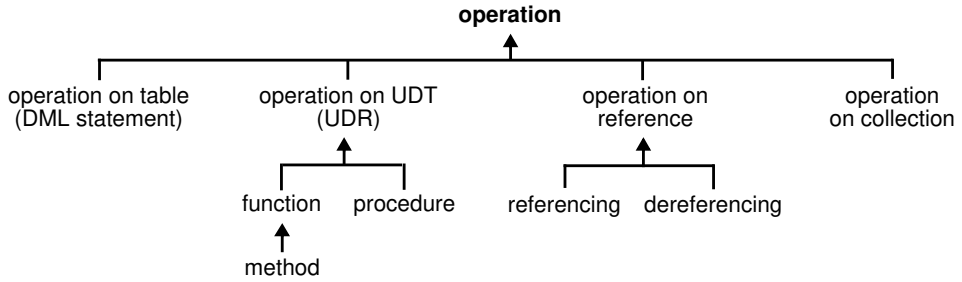
Von SQL aufrufbare Routinen

- **Überblick**
  - benannter, persistenter Code (*DB-stored*)
  - aufgerufen von SQL (*SQL-invoked*)
  - enthält einen Kopf (*Header*) und einen Rumpf (*Body*)

- **Klassifikation nach Sprache**
  - **SQL-Routine**
    - Header (Signatur) und Body spezifiziert in SQL
    - Routine-Body enthält eine einzige SQL-Anweisung inkl. einer zusammengesetzten PSM-Anweisung (*BEGIN . . . END*)
  - **Externe Routine**
    - Header (Signatur) spezifiziert in SQL
    - Body geschrieben in einer Wirtssprache
    - Body kann eingebettete SQL-Anweisungen enthalten

Von SQL aufrufbare Routinen (2)

- **Überblick**



- **Klassifikation nach Formen**
  - **Benutzerdefinierte Funktionen**
    - geben immer einzelne Werte als Ergebnis zurück
    - Überladbar, Typüberprüfung zur Übersetzungszeit
    - Auswahl über statische Parametertypen (statisches Binden)
  - **Benutzerdefinierte Prozeduren**
    - Aufgerufen durch eine CALL-Anweisung
    - Statisches Binden, kein Überladen
  - **Benutzerdefinierte Methoden**
    - sind Funktionen mit speziellen Aufrufkonventionen (p.JahresGehalt())
    - Überladbar, redefinierbar (überschreibbar)
    - „dynamic dispatch”; Laufzeit-Funktion von „SELF” bestimmt
    - Spätes Binden

SQL-Routinen

- **SQL-Prozeduren**
  - **Definitionsbeispiel**

```
CREATE PROCEDURE Kontoabfrage (IN KontoNr INT,
            OUT Betrag DECIMAL (15,2))
BEGIN
   SELECT       Kontostand INTO Betrag
      FROM      Konten
      WHERE     Kontonummer = KontoNr;
   IF Betrag < 100
      THEN      SIGNAL Niedriger_Kontostand
   END IF;
END
```

  - **Aufruf durch CALL-Anweisung**

```
CALL Kontoabfrage (4711, Betrag);
```

  - **Sind alle Arten von SQL-Anweisungen im Routine-Body erlaubt?**

- **Externe Prozeduren**

```
CREATE PROCEDURE Kontoabfrage (IN KontoNr INT,
            OUT Betrag DECIMAL (15,2))
LANGUAGE C
EXTERNAL NAME 'Konten/Abfrage_Prozedur'
```


SQL-Routinen (2)

- **SQL-Funktionen**
  - **Definitionsbeispiel**

```
CREATE FUNCTION Kontoabfrage (KontoNr INT)
RETURNS DECIMAL (15,2)
   BEGIN
      DECLARE       Betrag DECIMAL (15,2);
      SELECT        Kontostand INTO Betrag
         FROM       Konten
         WHERE      Kontonummer = KontoNr;
      IF Betrag < 100
         THEN SIGNAL Niedriger_Kontostand
      END IF;
      RETURN Betrag
   END
```

  - **Funktionsaufruf als Teil eines Ausdrucks**

```
SELECT Kontonummer, Kontoabfrage (KontoNr)
FROM Konten
```

  - **Ausnahmen?**

- **Externe Funktionen**

```
CREATE FUNCTION Kontoabfrage (KontoNr INT)
RETURNS DECIMAL (15,2)
LANGUAGE C
EXTERNAL NAME 'DBA/Konten/Abfrage'
```

Zusammenfassung

- **Objekt-Relationale Erweiterungen sind zentral**
  - Erweiterbares Typsystem
    - Benutzerdefinierte Typen (UDT) beschreiben die Anwendungsdaten
    - Benutzerdefinierte Routinen (UDR) definieren ein Verhalten für die Anwendungsdaten
  - Regeln und Trigger
  - Große Objekte (LOB) bis GByte werden unterstützt

- **Diese Erweiterungen erlauben mächtigere SQL-Anfragen**
  - einfachere und bessere Anwendungsentwicklung und -optimierung
  - einfachere, schnellere und mächtigere SQL-Anfragen
  - bessere Entscheidungsunterstützung, mächtigere Anfragegeneratoren

- **Trigger/Constraints erlauben**
  - Verbesserung der Datenintegrität
  - bessere Modellierung der Anwendungssemantik
  - Implementierung von Anwendungsregeln („Geschäftsregeln“)

- **Ziel: offene Architektur für SQL-Klassenbibliotheken (z. B. SQL/MM)**

➥ erlaubt Anwendern die Integration von Funktionalität von externen Anbietern

- **Wettbewerber**
  - IBM: DB2 Universal Database, mit Parallelität verfügbar
  - Informix Dynamic Server
  - Oracle
  - Microsoft SQL Server, Sybase Adaptive Server
  - CA Associates (OpenIngres ++?), Software AG (Adabas C ++?)

Kollektionstypen - Vorschläge

- **Weitere Kollektionstypen**
  - erlauben eine verbesserte Modellierung
  - liegen als Standardisierungssvorschläge vor

➥ komplexere Strukturen **implizieren komplexere Operationen**!

- **Beispiel**
  - soll einen Eindruck über die Modellierungsmächtigkeit von Kollektionen vermitteln
  - zeigt die Nutzung von Kollektionstypen einschließlich Schachtelung und Kombination mit strukturierten Typen

```
CREATE TABLE Pers (
   ID              INTEGER PRIMARY KEY,
   Name            ROW   (  Nachname CHAR(20),
                             Vornamen LIST ( CHAR(20) NOT NULL ) ) NOT NULL,
   Anschriften     ROW   (  PLZ CHAR(5), Ort CHAR(20) ) ARRAY[3],
   Mailing-Listen  SET   (  LIST (INTEGER NOT NULL REFERENCES Emp ) ) NOT NULL,
   Kinder          NESTED TABLE    (  Vornamen LIST (CHAR(20) NOT NULL ),
                                      Gebdat DATE ) NOT NULL,
   Bonus           MULTISET (DECIMAL(9, 2) NOT NULL ) )
```

- Es ergeben sich natürlich komplexere integritätsbedingungen:
  Ein Angestellter darf nicht zwei gleiche Vornamen haben

```
ALTER TABLE Pers
   ADD CONSTRAINT VornamenDuplikate CHECK ( UNIQUE
         (  SELECT p.ID, f.Vorname
            FROM Pers p, UNNEST (p.Name.Vornamen ) AS f ( Vorname ) ) )
```

Kollektionstypen - Vorschläge (2)

- **Beispiel** (Fortsetzung)
  - Nutzung von UNNEST auf Kollektionen:
    Angestellte die ihren ersten Vornamen ihren Kindern gegeben haben

```
SELECT   p.ID, p.Name
FROM     Pers p, UNNEST ( p.Name.Vornamen) WITH ORDINALITY
         AS f (Erster, Index ), UNNEST (p.Kinder ) AS k ( Kind )
WHERE    f.Index = 1 AND f.Erster IN k.Kind.Vornamen
```

  - Angestellte, die ein Kind haben und deren erste zwei Vornamen
    Peter und Paul sind

```
SELECT   p.ID, p.Name
FROM     Pers p
WHERE    CARDINALITY ( p.Kinder ) = 1 AND
         SUBLIST ( p.Name.Vornamen FROM 1 FOR 2) = LIST ( 'Peter', 'Paul' )
```

  - Einfügen eines neuen Angestellten

```
INSERT INTO Pers VALUES
   (   0815, ROW ( 'Abel', LIST ( 'Jens', 'Rainer' ) ),
       ARRAY [ROW ('67663', 'KL' ), NULL, ROW ( '67653', 'KL' ) ],
       SET ( LIST (4711, 0815 ), LIST (4712, 0815 ) ),
       NESTED TABLE (), MULTISET ( 10000,00 ) )
```

  - Änderung einer Adresse und Hinzufügen eines Bonus

```
UPDATE Pers
   SET Anschriften[3] = ROW ('67661', 'KL' ),
        Bonus = Bonus UNION ALL MULTISET ('1000,00')
   WHERE ID = 0815
```


Überladen von Routinen

- **Benutzer können eigene Datentypen und Routinen definieren**
  - mehrere Funktionen mit demselben Namen (foo)
  - Funktionen in unterschiedlichen Schemata (S1, S2), die alle die Parametertypen des Aufrufs akzeptieren
  - unqualifizierter Aufruf von Funktionen ist oft vorteilhaft (aufwärtskompatible Anwendungen, ohne Namen ändern zu müssen)

  ➥ Funktionsresolution (Routinenbestimmung)

- **Überladen:**
  Mehrere Routinen mit demselben unqualifizierten Namen

  S1.foo (p1 INT, p2 REAL)
  S1.foo (p1 REAL, p2 INT)
  S2.foo (p1 INT, p2 REAL)

- **Innerhalb desselben Schemas:**
  Jede überladene Routine muß eine eindeutige Signatur besitzen

  S1.foo (p1 INT, p2 REAL)
  S1.foo (p1 REAL, p2 INT)

- **Schemaübergreifend:**
  Überladene Routinen können dieselbe Signatur besitzen

  S1.foo (p1 INT, p2 REAL)
  S2.foo (p1 INT, p2 REAL)

Bestimmung der Routine

- **Auswahl der aufzurufenden Funktion** erfolgt aufgrund
  - der Datentypen aller Argumente (zur Übersetzungszeit)
  - der Typpräzedenzliste der Datentypen der Argumente
  - des SQL-Pfades

- **Typpräzedenzliste** legt zulässiges Propagieren von Datentypen für Funktionsargumente fest

  - Vordefinierte Typen
    - SMALLINT → INTEGER → DECIMAL → NUMERIC → REAL → FLOAT → DOUBLE
    - CHAR → VARCHAR → CLOB

  - Benutzerdefinierte Typen
    - Propagieren wird bestimmt durch die Subtyp-Supertyp-Beziehung
    - B sei Subtyp von A und C Subtyp von B; die Typpräzedenzliste für C ist dann (C, B, A)

- **SQL-Pfad ist eine Liste von Schemanamen**

  CREATE SCHEMA Schema 3
  PATH Schema1, Schema2, ...;

Bestimmung der Routine (2)

1. **Bestimme die Menge der Kandidaten-Funktionen** für einen gegebenen Funktionsaufruf F(a1, a2, ..., an):
   - Wenn der Name voll qualifiziert ist, also S1.F(a1, a2, ..., an), dann werden alle Funktionen aus S1 mit dem Namen F und n Parametern ausgewählt.
   - Bei nicht voll qualifizierten Namen werden alle Funktionen in jedem Schema des anwendbaren SQL-Pfades ausgewählt, die den Namen F und n Parameter besitzen.

2. **Eliminiere unpassende Kandidaten-Funktionen**
   a) Der Aufrufer hat kein EXECUTE-Privileg.
   b) Der Datentyp des i-ten Parameters ist nicht in der Typpräzedenzliste des statischen Typs für das i-te Argument.

3. **Wähle die besten Kandidaten** aus den verbleibenden Funktionen aus
   a) Überprüfe den Typ des ersten Parameters jeder Funktion und behalte nur solche Funktionen, deren erster Parametertyp am besten zum statischen Typ des ersten Argumentes paßt.
   b) Wiederhole diese Überprüfung für die nachfolgenden Parameter, bis entweder eine Funktion übrig bleibt oder alle Parameter überprüft sind.

4. **Bestimme die anzuwendende Funktion** (subject function)
   Falls mehrere Funktionen übrig sind, wähle diejenige aus, deren Schema zuerst im anwendbaren SQL-Pfad aufgelistet ist.