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

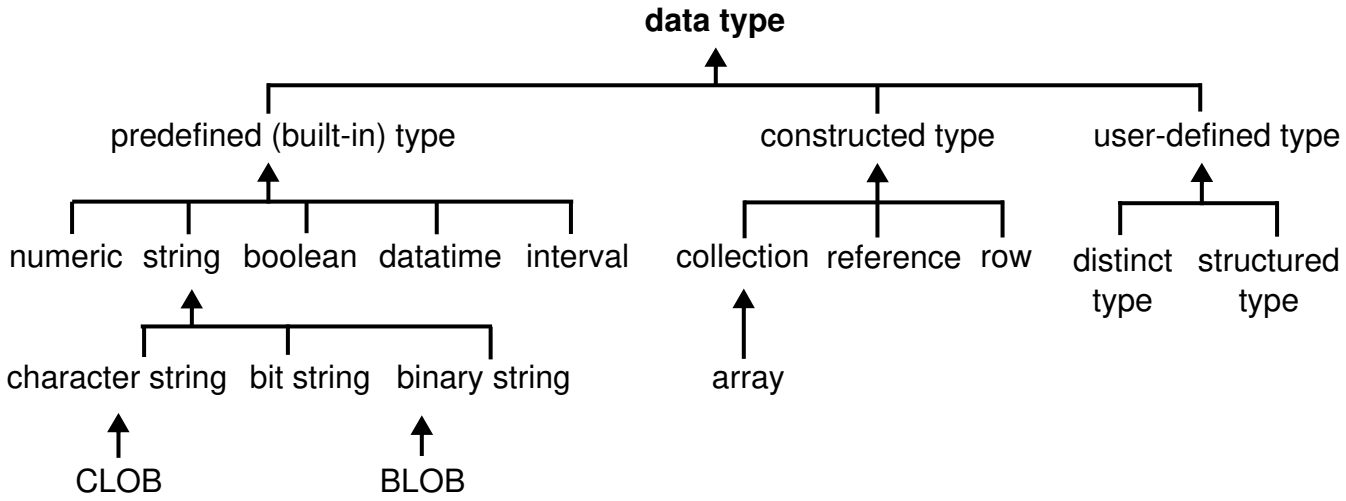
- **Leistungsverhalten**

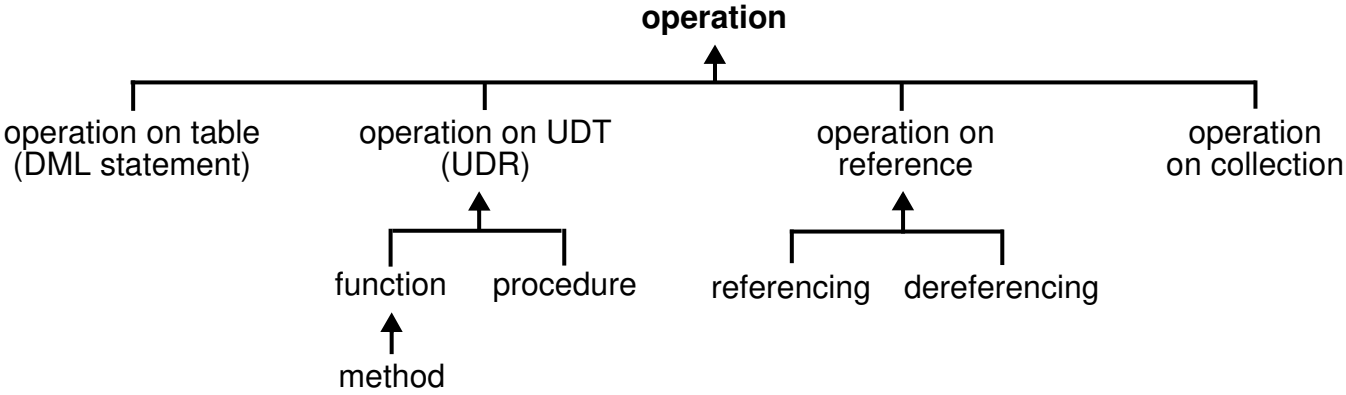
  Möglichkeit zur DBMS-Integration von Typen und Funktionen als „first class citizen“ (volle Verarbeitungsmächtigkeit)

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**



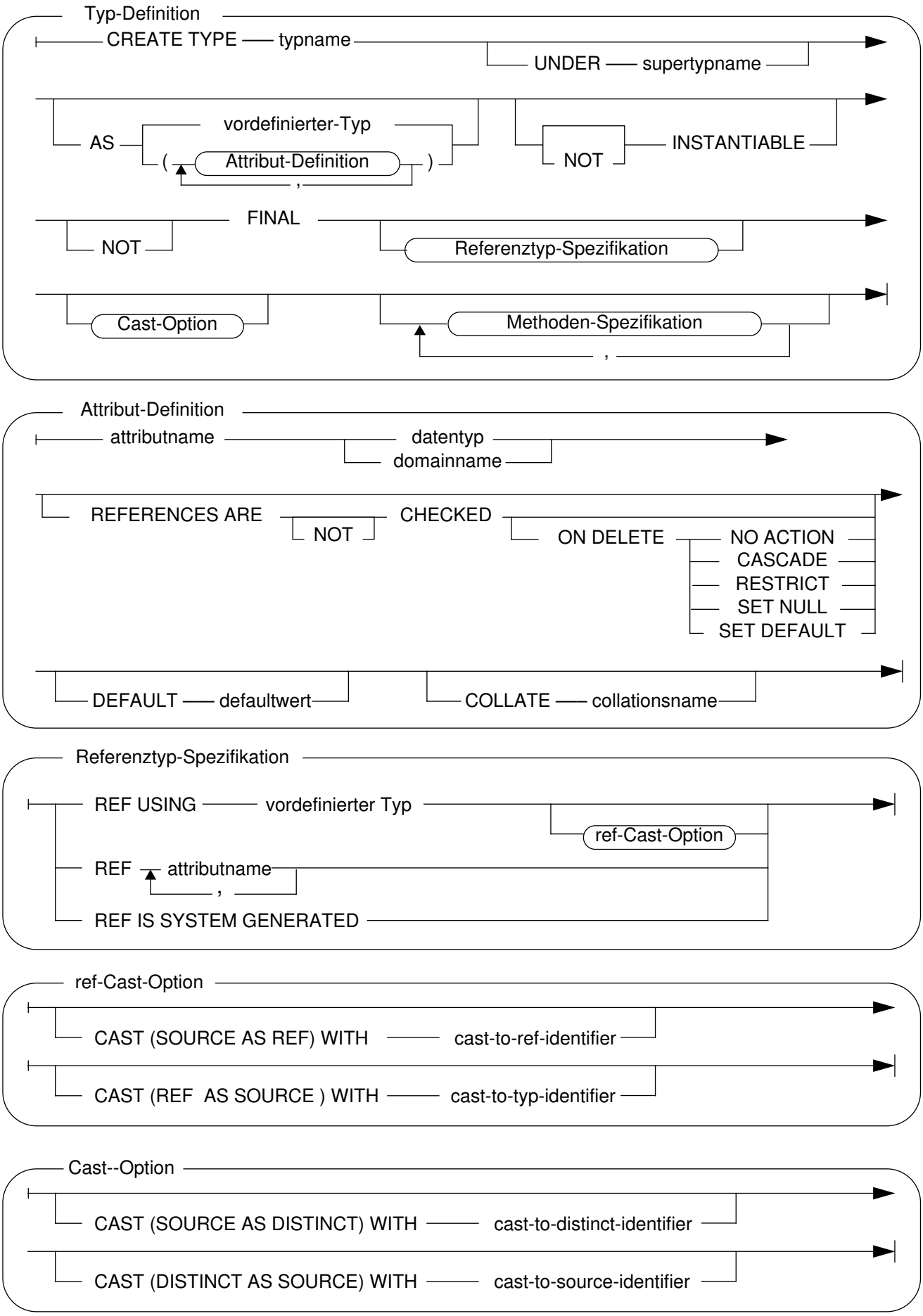


Benutzerdefinierte Typen (4)

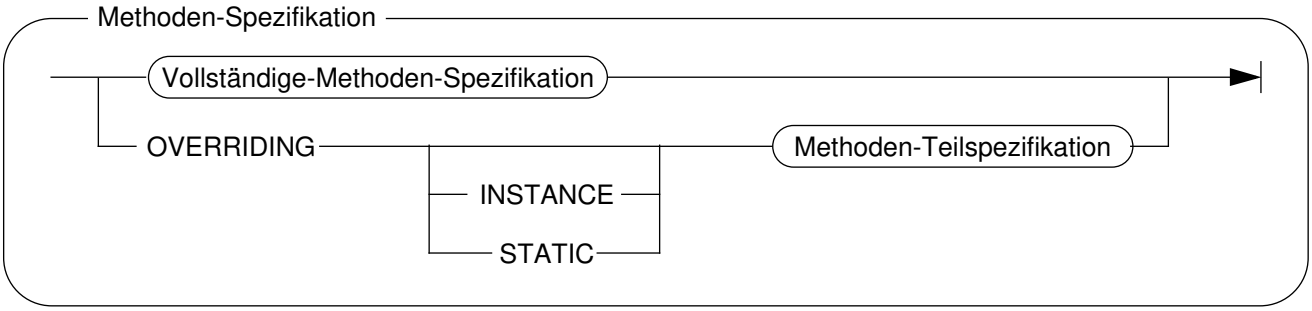
- **Definition des Datentyps**
  - CREATE TYPE definiert einen neuen strukturierten Typ (ADT, UDT): eine benamte Menge von gültigen Datenwerten
  - Festlegung von bis zu 12 Aspekten (ggf. über Defaultwerte) erforderlich
    - Name, Beziehung zu anderen Typen
    - Basistyp bei umbenannten Typen, Attributdeskriptoren 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)

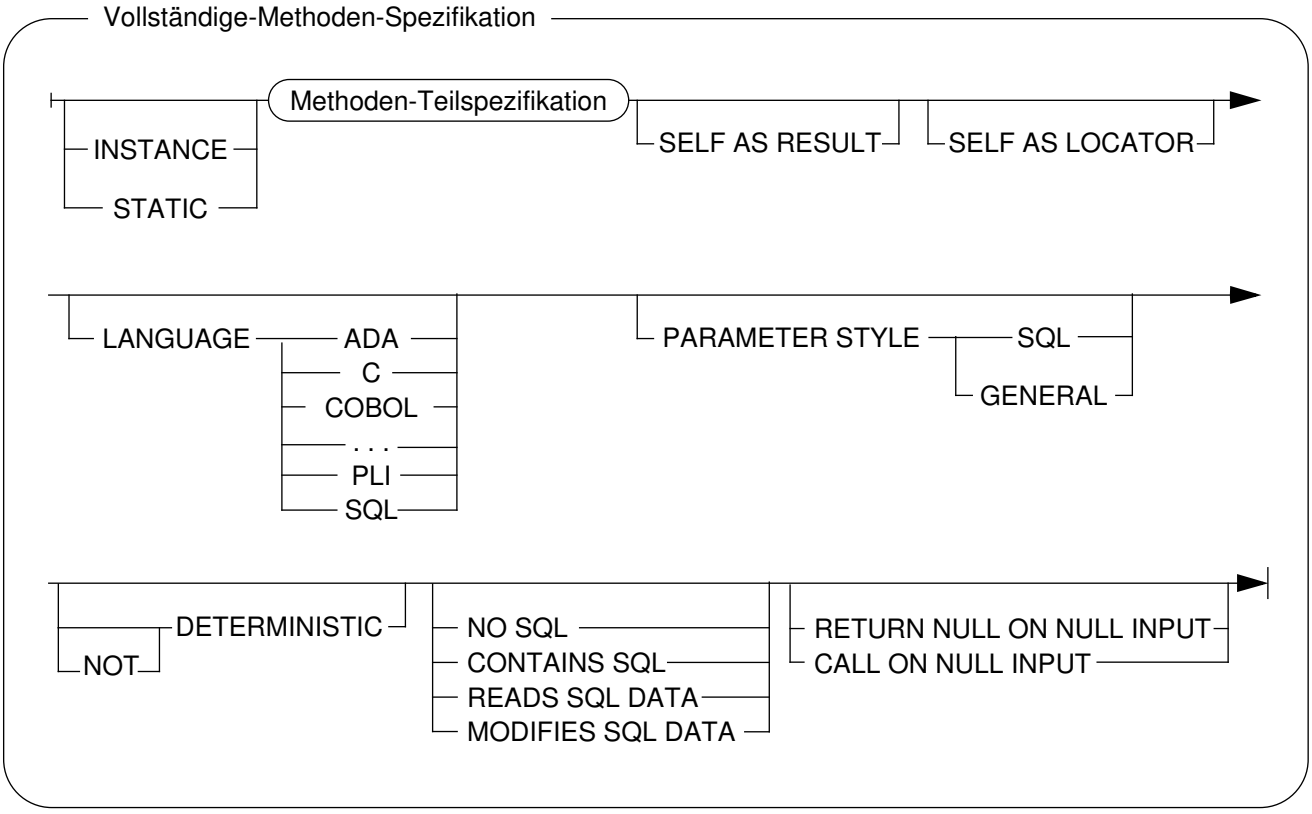
- **Definition von Methoden (und Funktionen)**
  - Name
  - Signatur (d.h. Parameterliste)
  - Ergebnis
  - Implementierung

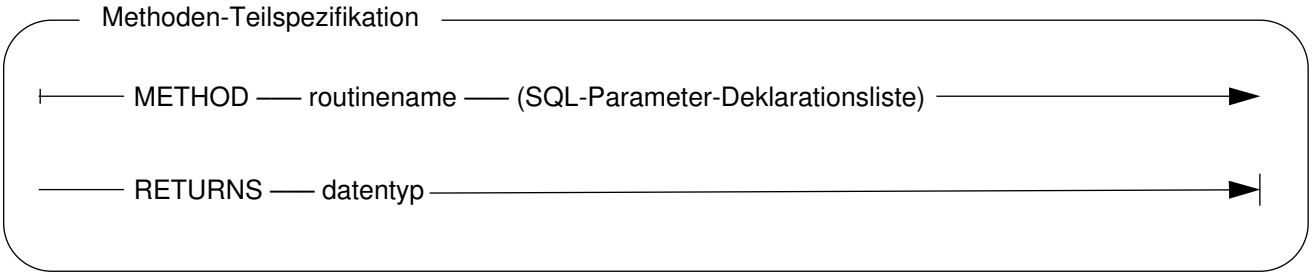
Typdefinition



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)

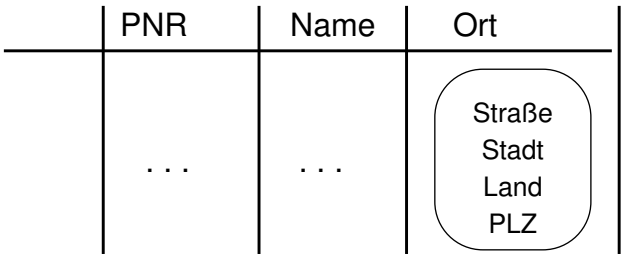
PNR	Name	Ort
. . .	. . .	Straße Stadt Land PLZ

Staße	Stadt	Land	PLZ
. . .	. . .	. . .	. . .

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

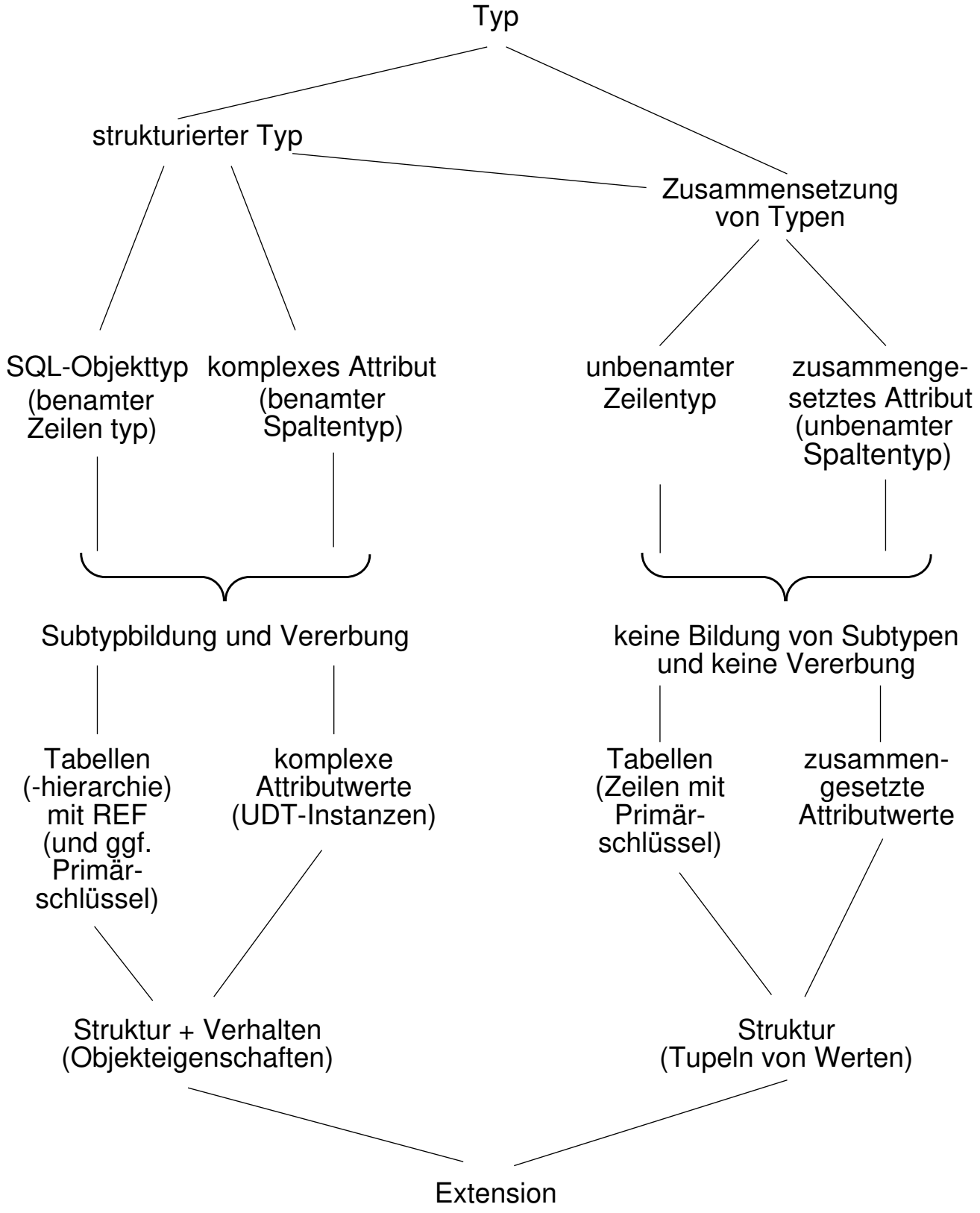


Staße	Stadt	Land	PLZ
. . .	. . .	. . .	. . .

  - **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:

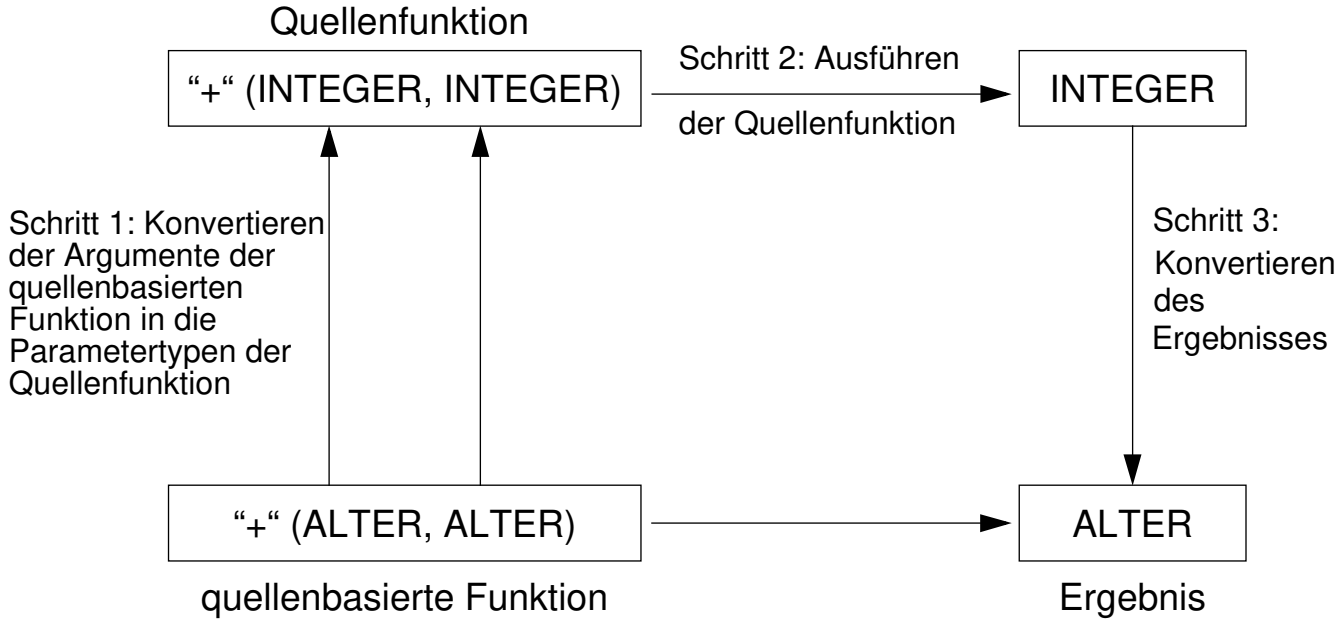
    SELECT Alter1 + alter (Iq1) AS Kennziffer

  - 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 Iq1 > 150
  - SET Iq1 = Iq1 · 2
  - WHERE Alter1 + Iq1 > Alter2

Umbenannte Typen (5)

- **Anwendungsbeispiel**

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

CREATE TABLE D_Verkäufe
    ( Kunden_Nr     INTEGER,
      Vertrags_Nr   INTEGER,
      Gesamt        D_MARK);

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

ALTER TABLE D_Verkäufe
ADD Bonus D_MARK
```

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

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

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

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

  - Hier sind offensichtlich spezielle Konversionsroutinen, die Semantik berücksichtigen, erforderlich!
    US_Dollar_To_D_Mark (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)
  - frühere SQL-Versionen unterschieden zwischen ADTs und ROW-Types

  ➥ 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**

Name	Alter	Ort
. . .	. . .	Straße Stadt Land PLZ

Staße	Stadt	Land	PLZ
. . .	. . .	. . .	. . .

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

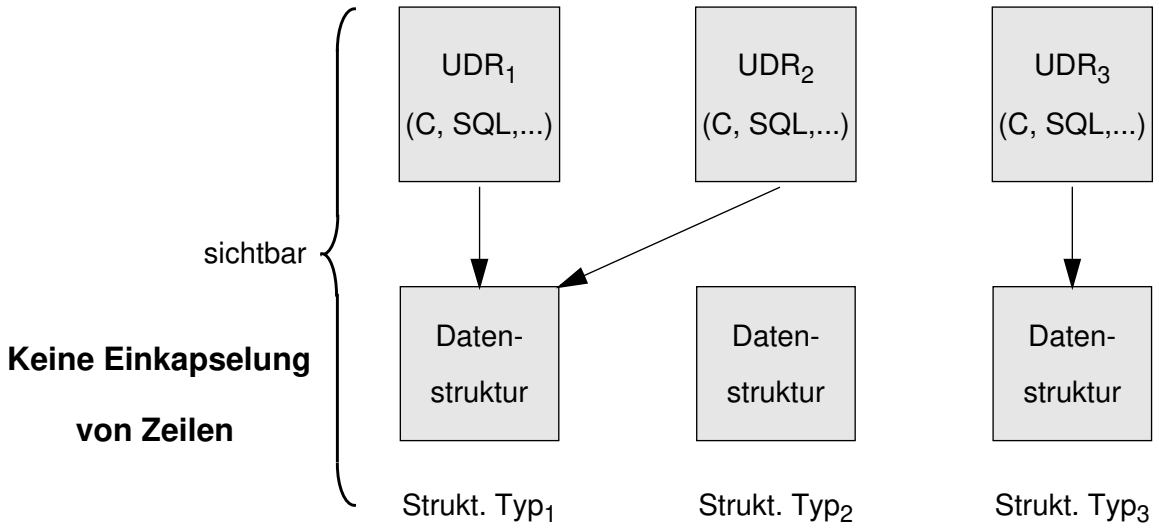
  CREATE TYPE Adresse **AS**
  (**internal length** = 56 **VARYING**,
  input = Adreßeingabe,
  output = Adreßausgabe)

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

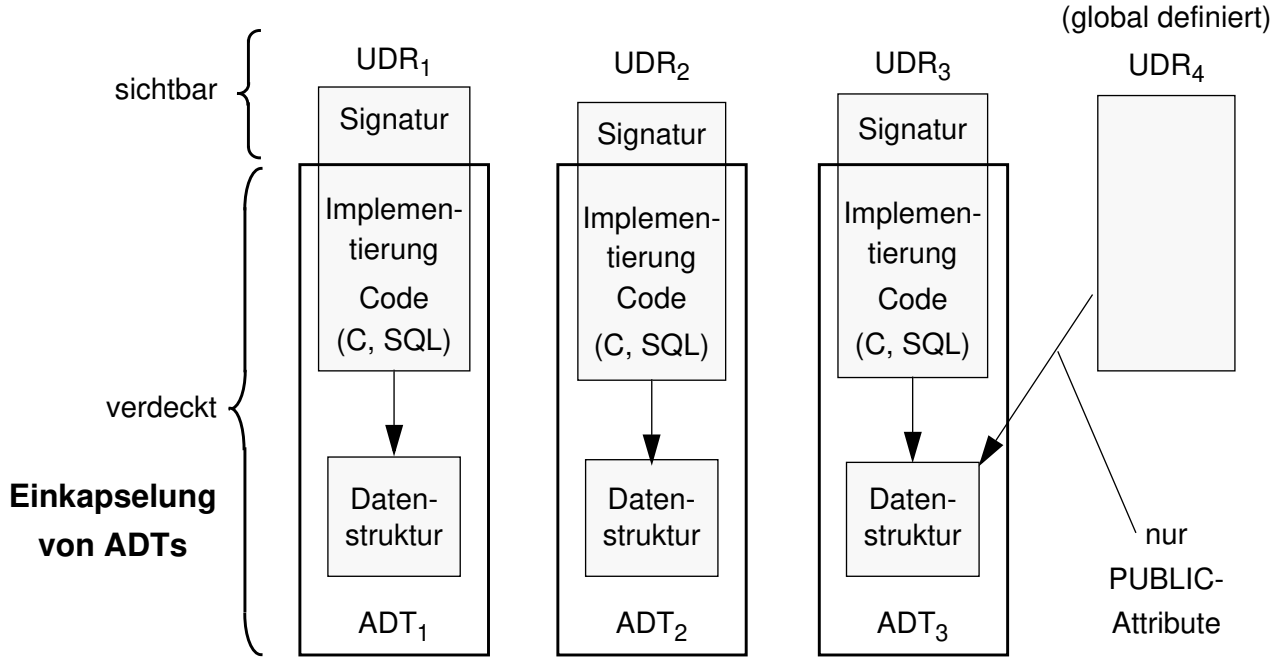
- **Eigenschaften**
  - Plattformabhängigkeit
  - DBS „versteht“ nicht die Objektsemantik
    - keine Unterstützung bei der Suche
    - keine Maßnahmen zur Integritätssicherung
  - oft mit Hilfe von LOBs implementiert

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. (a sei Adressen-Variable/Ausdruck)
    - nicht überladen werden.

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, D_Mark 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    D_Mark (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**

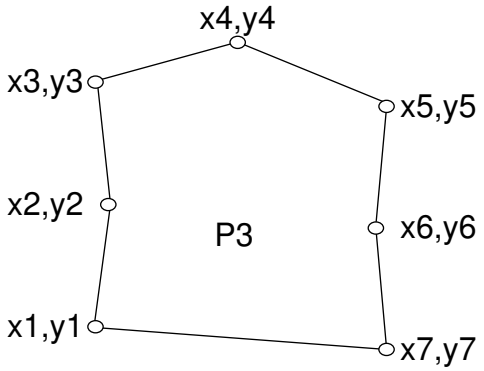
```
DECLARE im Immob _t;
. . .
/* Erzeugen eines im-Wertes */
INSERT INTO ImmoListe
        VALUES (123, im);

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      D_Mark(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)

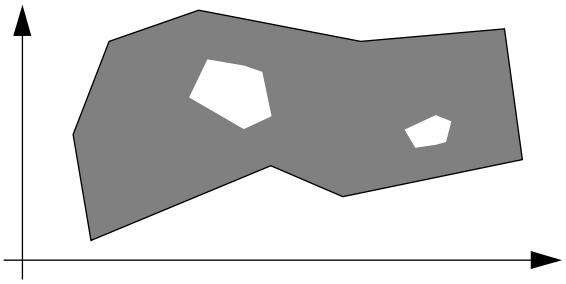
  METHOD X_Koord () **RETURNS** (**DECIMAL** (7,3))

  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!

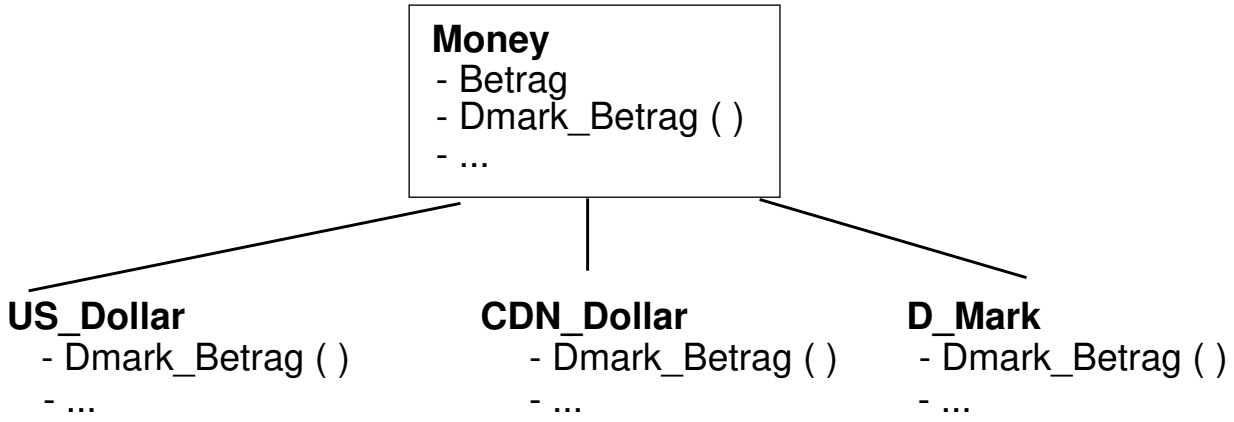
- **Verwendung von Operationen auf strukturierten Typen in SQL-Anfragen**

```
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>
<D_Mark> Betrag: 150,000	‘D.Schwarz‘	<Adresse>

```
SELECT   Besitzer, Dmark_Betrag(Preis)
FROM     Immobilien_Info
WHERE    Dmark_Betrag(Preis) <
         D_Mark (500000)
```

- **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

- **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**

- Autobahnen

Name	Länge	Breite	Gebühr
A6	324.00	18.20	<D_Mark> 20

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