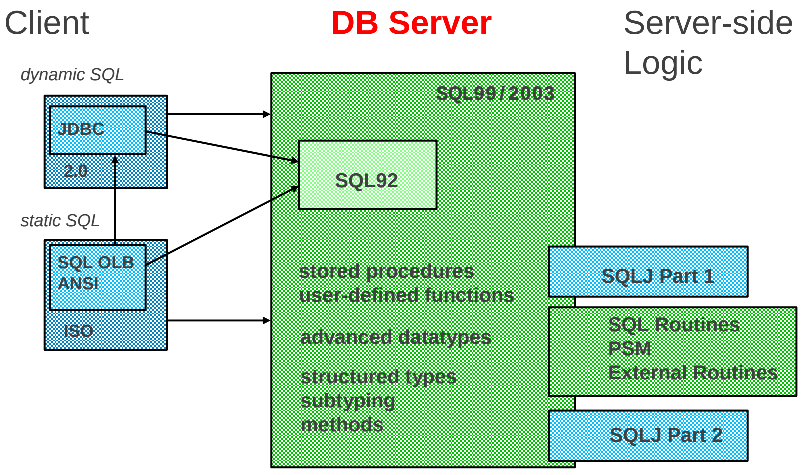
Prof. Dr.-Ing. Stefan Deßloch
AG Heterogene Informationssysteme
Geb. 36, Raum 329
Tel. 0631/205 3275
dessloch@informatik.uni-kl.de

Technische Universität Kaiserslautern

Chapter 1 – User-defined Types and Typed Tables

Outline

The "Big Picture"



Objects Meet Databases (Atkinson et. al.)

- Object-oriented features to be supported by an (OO)DBMS
  - Complex objects
    - type constructors: tuple, set, list, array, ...
  - Object identity
    - object exists independent of its value (i.e., identical ≠ equal)
  - Encapsulation
    - separate specification (interface) from implementation
  - Types and classes
    - "abstract data types", static type checking
    - class as an "object factory", extension (i.e., set of "instances")
  - Type or class hierarchies
    - inheritance, specialization
  - Overloading, overriding, late binding
    - same name for different operations or implementations
  - Computational completeness
    - use DML to express any computable function (-> method implementation)
  - Extensibility
    - user-defined types (structure and operations) as first class citizens
    - strengthens some capabilities defined above (encapsulation, types)

User-defined Types: Key Features

- New functionality
  - Users can indefinitely increase the set of provided types
  - Users can indefinitely increase the set of operations on types and extend SQL to automate complex operations/calculations
- Flexibility
  - Users can specify any semantics and behavior for a new type
- Consistency
  - Strong typing insures that functions are applied on correct types
- Encapsulation
  - Applications do not depend on the internal representation of the type
- Performance
  - Potential to integrate types and functions into the DBMS as "first class citizens"

User-defined Types: Benefits

- Simplified application development
  - Code Re-use - allows reuse of common code
  - Overloading and overriding - makes application development easier -- single function name for a set of operations on different types, e.g., area of circles, triangles, and rectangles
- Consistency
  - Enables definition of standard, reusable code shared by all applications (guarantee consistency across all applications using type/function)
- Easier application maintenance
  - Changes are isolated: if application model changes, only the corresponding types/functions need to change instead of code in each application program

Strong Typing

- Well-known concept in programming languages
  - every value has a type (pre-defined or user-defined) with supported operations
  - system checks type correctness when performing operations (often statically)
- Before SQL99, columns could only be defined with the existing built-in data types
  - There was no **strong typing**
  - Logically incompatible variables could be compared, assigned to each other, …

```
CREATE TABLE RoomTable (
RoomID          CHAR(10),
RoomLength      INTEGER,
RoomWidth       INTEGER,
RoomArea        INTEGER,
RoomPerimeter   INTEGER);
```

```
UPDATE RoomTable
SET RoomArea = RoomLength;
```

No Error Results

User-defined Distinct Types

- Each DT is logically incompatible with all other types

```
CREATE TYPE plan.roomtype
AS CHAR(10);

CREATE  TYPE plan.meters
AS INTEGER;

CREATE  TYPE plan.squaremeters
AS INTEGER;

CREATE TABLE RoomTable (
RoomID          plan.roomtype,
RoomLength      plan.meters,
RoomWidth       plan.meters,
RoomPerimeter   plan.meters,
RoomArea        plan.squaremeters);
```

```
UPDATE RoomTable
SET RoomArea =
RoomLength;
```

ERROR

```
UPDATE RoomTable
SET RoomLength =
RoomWidth;
```

NO ERROR RESULTS

User-defined Distinct Types

- Renamed type, with different behavior than its source type.
  - Shares internal representation with its source type
  - Source and distinct type are not directly comparable

CREATE TYPE EURO AS DECIMAL (9,2)

- Strong typing
- Operations on distinct types (behavior)
  - Comparison/ordering (automatically created by default)
    - Can be based on the comparison/ordering of their source type
  - Casting (automatically created by default)
    - Used to explicitly cast instances of the distinct type and instances of source type to and from one another
    - Used to obtain "literals"
  - Methods and functions
  - No inheritance or subtyping

Cast Functions for Distinct Types

- Automatically defines cast functions to and from the source type for a user-defined distinct type
  - Casts will also be allowed from any type that is promotable to the source type of the user-defined type (i.e., that has the source type in its type precedence list)
    - Casting from a SMALLINT to a UDT sourced on an integer is OK

```
CREATE TYPE plan.meters
AS INTEGER
CAST (SOURCE AS DISTINCT) WITH meters
CAST (DISTINCT AS SOURCE) WITH integer
```

Implicit Cast Functions created:

```
plan.meters(integer) returns meters;
plan.integer(meters) returns integer;
```

Example Casting Expressions:

```
... SET RoomWidth =
    CAST (integerCol AS meters)
or
    meters(integerCol)
or
    meters(smallintCol)
```

Cast Functions: Comparison Rules

- Casts must be used to compare distinct type values with source-type values.
  - Constants are always considered to be source type values
  - You may cast from source type to UDT, or vice-versa

```
SELECT * FROM RoomTable
  WHERE RoomID = 'Bedroom';
        ERROR
SELECT * FROM RoomTable
  WHERE RoomID =  roomtype('Bedroom');
 or
SELECT * FROM RoomTable
  WHERE char(RoomID) = 'Bedroom';
     No Error Results
```


Cast Functions: Assignment Rules

- In general, source-type values may not be assigned to user-defined type targets (i.e., explict casting is required)
- The strong typing associated with UDTs is relaxed for assignment operations, IF AND ONLY IF a cast function between source and target type has been defined with the AS ASSIGNMENT clause (this is the default)

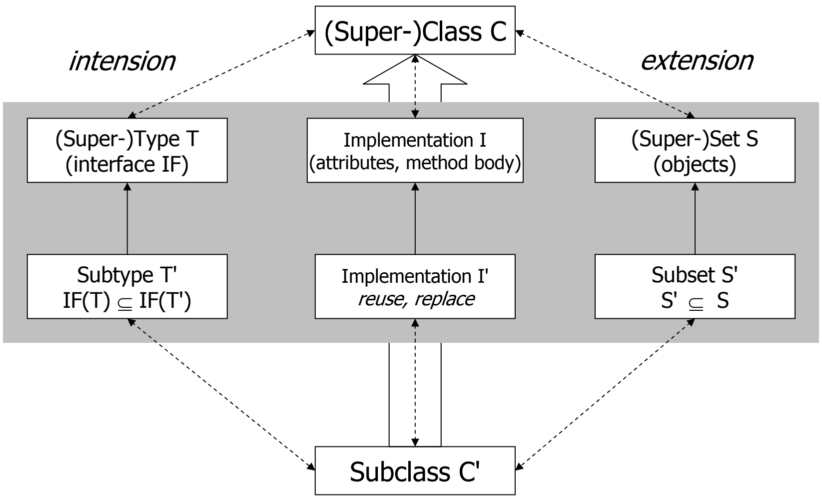
```
CREATE TYPE plan.meters
 AS INTEGER
 CAST (SOURCE AS DISTINCT) WITH meters
 CAST (DISTINCT AS SOURCE) WITH integer

CREATE CAST (plan.meters AS integer) WITH
 integer AS ASSIGNMENT

 CREATE CAST (integer AS plan.meters) WITH
 meters AS ASSIGNMENT
```

```
Select  RoomLength, RoomWidth
 INTO :int_hv1, :int_hv2
 FROM RoomTable

Upate RoomTable
Set RoomLength = 10

    No Error Results
```

Distinct Types vs. Domains

- Distinct types are comparable to value domains in the RM
  - see, e.g., Date/Darwen "Foundation for Object/Relational Databases"
    - "domain" and "type" used interchangably
- Warning: domains in SQL are different!
  - domain definition
    - name
    - data type (similar to source type)
    - constraint (optional)
    - default (optional)
    - collation (optional)
  - comparable to macros in programming languages
    - no notion of strong typing

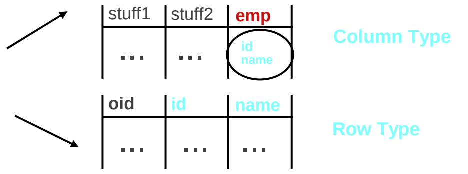
Object Types and Classes



User-defined Structured Types (ST)

- User-defined, complex data types (intensional)
  - definition of state (attributes) and behavior (methods, *see chapter 3*)
  - can be used as data type wherever predefined data types can be used
    - type of domains or columns in tables
    - attribute type of other structured types
    - type of parameters of functions, methods, and procedures
    - type of SQL variables
  - strong typing
  - no object identity
- Structured Types can be used to define typed tables (extensional)
  - types and functions for rows of tables
    - for modeling entities with relationships & behavior
  - explicit object identifier column

```
CREATE TYPE employee AS
(id      INTEGER,
name  VARCHAR (20))
```



Structured Types: Example

```
CREATE TYPE address AS
(street             CHAR (30),
city                CHAR (20),
state               CHAR (2),
zip                 INTEGER) NOT FINAL

CREATE TYPE bitmap AS BLOB

CREATE TYPE real_estate AS
(owner              CHAR(40),
price               money,
rooms               INTEGER,
size                DECIMAL(8,2),
location            address,
text_description    text,
front_view_image    bitmap,
document            doc) NOT FINAL
```

Creating Instances of Structured Types

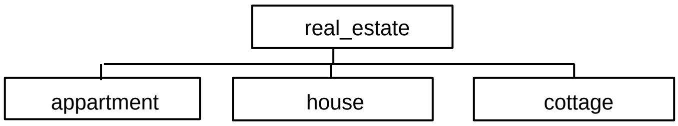
- System-supplied constructor function
  - address () -> address or real_estate () -> real_estate
    - Returns new instance with attributes initialized to their default
- NEW operator
  - NEW <type name>
    - Invokes constructor function
- Example
  - CREATE TABLE people (..., addr address, ...)
  - INSERT INTO properties VALUES (... , **NEW address**, ...)

Accessing Attributes

- "Dot" notation must be used to access attributes
- Example
  - SELECT location.street, location.city, location.state, location.zip
    FROM properties
    WHERE price < 100000
- Support for several `levels' of dot notation (a.b.c.d.e)
- Allow "navigational" access to structured type values
- Assignment syntax
  - DECLARE r real_estate;
    ...
  - SET r.size = 2540.50;
    ...
  - SET ... = r.location.state;
  - SET r.location.city = 'Los Angeles';

Subtyping and Inheritance

- Structured types can be a subtype of another ST
- STs inherit structure (attributes) and behavior (methods) from their supertypes
- Example
  - CREATE TYPE real_estate ... NOT FINAL
  - CREATE TYPE appartment UNDER real_estate ... NOT FINAL
  - CREATE TYPE house UNDER real_estate ... NOT FINAL



Noninstantiable and Final Types

- Structured types may be **noninstantiable**
  - Like abstract classes in OO languages
    - No system-supplied constructor function is generated
    - Type does not have instances of its own
  - Instances can be defined on subtypes
- By default, structured types are instantiable
- Distinct types are always instantiable

```
CREATE TYPE person AS
(name          VARCHAR (30),
address        address,
sex            CHAR (1)) NOT INSTANTIABLE NOT FINAL
```

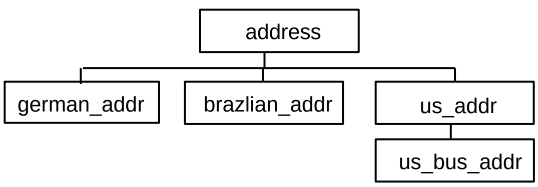
- User-defined types may be **final**
  - no subtypes can be defined
    - distinct types have to be **FINAL**
    - structured types have to be **NOT FINAL**
  - future version of SQL may introduce more flexibility

Subtyping and Substitutability

- Based on the intensional aspects of OO (i.e., type interface)
- B is a subtype of A
  - B supports at least the operations defined for A
  - data types for method parameters or (public) attributes inherited from A are not redefined/specialized in B
  - whereever an instance of A can be used, and instance of B can be used instead
- In object-oriented programming languages
  - instances of type B can be
    - assigned to variables of type A
    - used in a method invocation as a parameter of type A
    - produced as a result of an expression or operation with result type A
  - enables static typing
- Should be supported in a database context as well

Value Substitutability

```
CREATE TYPE address AS
    (street CHAR(30), city CHAR(20), state CHAR(2), zip INTEGER) NOT FINAL
CREATE TYPE german_addr UNDER address
    (family_name VARCHAR(30) ) NOT FINAL
CREATE TYPE brazilian_addr UNDER address
    (neighborhood VARCHAR(30) ) NOT FINAL
CREATE TYPE us_addr UNDER address
    (area_code INTEGER, phone INTEGER) NOT FINAL
CREATE TYPE us_bus_addr UNDER us_address
    (bus_area_code INTEGER, bus_phone INTEGER) NOT FINAL
```



Value Substitutability

- Each row can have a column value of a different subtype!

  INSERT INTO properties (price, owner, location)
  VALUES (US_dollar (100000), REF('Mr.S.White'), NEW us_addr ('1654 Heath Road', 'Heath', 'OH', 45394, ...))

  INSERT INTO properties (price, owner, location)
  VALUES (real (400000), REF('Mr.W.Green'), NEW brazilian_addr ('245 Cons. Xavier da Costa', 'Rio de Janeiro', 'Copacabana'))

  INSERT INTO properties (price, owner, location)
  VALUES (german_mark (150000), REF('Mrs.D.Black'), NEW german_addr ('305 Kurt-Schumacher Strasse', 'Kaiserslautern', 'Schwarz'))

price	owner	location
<us_dollar amount 100,000	'Mr. S. White'	<us_addr '1654 Heath ...'
<real amount 400,000	'Mr. W. Green'	<brazilian_addr> '245 Cons. Xavier ...'
<german_mark amount 150,000	'Mrs. D. Black'	<german_addr> '305 Kurt-Schumacher ...'

internal type tag

Type Predicate

- Allows determination of **dynamic** type (**most specific** type)
- Purpose
  - Allows row selection by specific subtypes (e.g. US_ADDR)
    - **IS OF**
  - Allows to prune off certain subtypes (e.g. US_BUS_ADDR)
    - **ONLY**
- Example: Find items from properties table that have a US address (but not a US business address or any other subtype of US address):

  SELECT * FROM properties
  WHERE location **IS OF ONLY** (US_ADDR)

Structured Types and Typed Tables

- Structured types can be used to define typed tables
  - Attributes of type become columns of table
  - In addition, a typed table has a so-called self-referencing column
    - holds a value that uniquely identifies the row (similar to an object id)
      - (more details later)

```
CREATE TYPE real_estate AS
(owner                  CHAR(40),
price                   money,
rooms                   INTEGER,
size                    DECIMAL(8,2),
location                address,
text_description        text,
front_view_image        bitmap,
document                doc) NOT FINAL …

CREATE TABLE properties OF real_estate
(REF IS oid …)
```

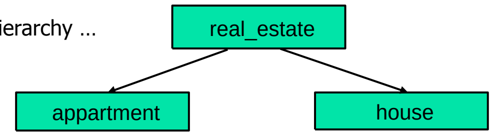

Manipulating Attributes

- Queries over typed tables access attributes (columns)
- Update statements on typed tables modify attributes

```
CREATE TABLE properties OF  real_estate ...

SELECT owner, price
FROM properties
WHERE address = NEW address  ('1543 3rd Ave. North, Sacramento, CA 93523')

UPDATE properties
SET price = 350000
WHERE address = NEW address  ('1543 3rd Ave. North, Sacramento, CA 93523')
```

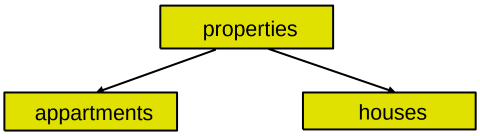
Subtables: Table Hierarchies

- Typed tables can have subtables
  - Inherit columns, contraints, triggers, ... from the supertable
- Example
  - Given the following type hierarchy ...



  - Create a table hierarchy:

CREATE TABLE properties OF real_estate (...)
CREATE TABLE appartments OF appartment **UNDER** properties
CREATE TABLE houses OF house **UNDER** properties



Relationship to Type Hierarchies

- Each table T(i) in the hierarchy must correspond to a type ST(i) of a single type hierarchy
- Relationships must match
  - T(i) UNDER T(j) => ST(i) UNDER ST(j)
- Not all types in the hierarchy have to have corresponding tables in the table hierarchy
- Multiple table hierarchies may be defined, based on the same type hierarchy

type hierarchy

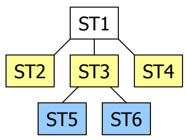
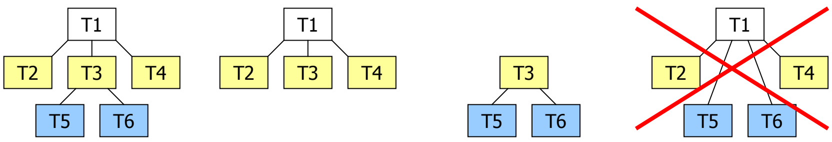
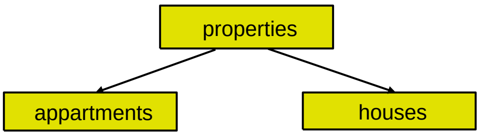


table hierarchies



Queries and Table Hierarchies

- Query involving a table in a hierarchy also ranges over the rows of every subtable
  SELECT price, location.city, location.state FROM properties
  WHERE address.city = 'Sacramento'
  - Returns rows from properties, appartments, and houses
  - Can only return columns defined for properties
    - E.g., SELECT * FROM properties will not return attributes created in the appartment type
- Queries on a subtable require SELECT privilege on that subtable
  SELECT * FROM appartments...
- Update, Delete statements also range over the subtables
  - UPDATE properties SET price = … WHERE … may update appartments and houses



Object Identity

- Object identity
  - a property that distinguishes an object from all other objects
  - independent of the object state (identity vs. equality)
  - remains unchanged for the lifetime of the object
- Object IDs as a foundation for representing relationships, references, sharing
  - dangling references should be prohibited/avoided
- What's wrong with keys?
  - artificial keys need to be generated by the user/application
  - a meaningful attribute used as a key is part of the object state and may change
    - may cause problems in maintaining relationships
  - "nonuniformity" of keys across multiple tables, in data integration scenarios, or during DB lifetime
  - "unnatural" joins when "traversing" relationships or references
    SELECT p.price, o.phone FROM properties p, person o WHERE p.owner=o.name
    instead of
    SELECT price, owner->phone FROM properties

Variations of Object Identity

- External or internal
  - external IDs visible, inspectable by an application
  - internal, maintained and used by the system, operations may implicitly involve IDs (e.g., establishing/traversing reference relationships)
- Scope of uniqueness
  - collection/table, table hierarchy, type extension, database, cluster of databases, …
  - very much depends on the intended functionality
- Reusable or not
  - reuse of IDs may result in erroneous relationships, unless dangling references can be avoided by the system
- System-generated or user-generated
  - system-generated during object creation
    - system can guarantee uniqueness and avoid reuse easily
    - user/application does not have to generate IDs
  - user-generated
    - more suitable for inserting object "networks" (unless a tight PL coupling with pointer swizzling is supported) from application or using import/export/load utilities
    - compatible with object-oriented views that preserve relationships
    - can be combined with DBMS function for creating unique initial values

SQL Reference Types

- Structured types have a corresponding reference type
  - REF(<structured type name>)
  - used to identify/reference instances of the structured type stored in types tables
    - identifier: stored in the self-referencing column of a typed table
      - has to be unique within the table hierarchy
  - Can be used wherever other types can be used
- Representation
  - User generated (REF USING <predefined type>)

    ```
    CREATE TYPE real_estate AS
       (owner  REF (person), ...)
       NOT FINAL REF USING INTEGER
    ```

  - System generated (REF IS SYSTEM GENERATED)
    - Default is system generated
  - Derived from a list of attributes (REF (<list of attributes>)

    ```
    CREATE TYPE person AS
       (ssn     INTEGER,
        name    CHAR(30),...)
       NOT FINAL REF (ssn)
    ```

Reference Type Representation

self-referencing column (SRC)	SYSTEM GENERATED	USER GENERATED	DERIVED (c1, …, cn)
uniqueness	UNIQUE, NOT NULL is implicit	UNIQUE, NOT NULL is implicit	UNIQUE, NOT NULL is implicit, (c1, …cn) has to be UNIQUE, NOT NULL as well
id value generation	automatic	by user/application during INSERT	automatic (based on c1, …cn)
id update	not permitted	not permitted	not permitted, but c1, …, cn can be updated
id reuse	can be prohibited by the DBS	has to be prohibited/controlled by application	has to be prohibited/controlled by application

More on Reference Types

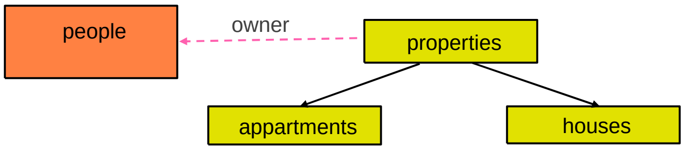
- Inserting reference values
  - USER GENERATED
    - value is provided by the application, just like any other column/attribute value
    - appropriate CAST functions are available
  - SYSTEM GENERATED or DERIVED reference
    - value needs to be retrieved from the database (in a subquery)

      ```
      INSERT INTO properties
      VALUES ( (SELECT pers-oid FROM people where ssn = '123-456-7890'), …)
      ```
- References are strongly typed
  - only references to the same/compatible type can be compared, assigned, …
- References support substitutability
  - for REF(T), a reference to an instance of a subtype of T is permitted

Reference Types

- Reference values can be scoped
  - typed table in which the referenced objects are supposed to exist
  - scoped reference can be checked
    - existence of referenced object is guaranteed by the DBMS
    - ON DELETE clause, similar to referential integrity constraint
- Example:

```
CREATE TYPE person (ssn INTEGER, name    ...) NOT FINAL
CREATE TYPE real_estate (owner REF (person), ...) NOT FINAL
CREATE TABLE people OF person ( ...)
CREATE TABLE properties OF real_estate
    (owner WITH OPTIONS SCOPE people REFERENCES ARE CHECKED)
```



Path Expressions

- Scoped references can be used in path expressions

```
SELECT prop.price, prop.owner->name FROM properties prop
WHERE prop.owner->address.city = "Hollywood"
```

- Authorization checking follows SQL authorization model
  - user must have SELECT privilege on people.name and people.address
  - the above statement is equivalent to

```
SELECT prop.price, (SELECT name FROM people p WHERE p.oid = prop.owner)
FROM properties prop
WHERE (SELECT p.address.city FROM people p WHERE p.oid = owner) = "Hollywood"

SELECT prop.price, p.name
FROM properties prop LEFT JOIN people p ON (prop.owner = p.oid)
WHERE p.address.city = "Hollywood"
```

Reference Scope and Uniqueness – Discussion

- Current SQL:2003 restrictions/limitations
  - Self-referencing columns (i.e., OIDs) are unique only within a table hierarchy
  - Dereferencing, path expressions only allowed for scoped references
  - Reference scope is limited to a single table
- Consequences of removing these restrictions in an object-relational DBMS
  - Dereferencing, path expressions for unscoped references
    - The table to be accessed is not known statically
      - object reference representation/implementation needs to incorporate table identifier
      - dynamic authorization checking, on a per tuple basis
    - Update of referenced object – which table to update?
      - either requires additional functionality to find out the table in which the object is stored, and results in complex update logic in the application
      - or requires a more flexible UPDATE statement that does not require a table name
    - Allowing multiple tables in a reference scope still does not solve the update problem
  - Broadening the scope of object identifier uniqueness
    - To all objects of a given type (hierarchy) – required for flexible dereferencing
    - To all objects in a DB? – is this useful for typed references?

Reference Resolution: Nesting

- References can be used to obtain the structured type value that is being referenced
  - Enables nesting of structured types

    ```
    SELECT prop.price, DEREF(prop.owner) AS ownerval
    FROM properties.prop
    ```
  - Column ownerval in the result table has static type person
- DEREF nests rows from subtables, respecting value substitutability
  - most specific type of ownerval values may be a subtype of person

Reference Types vs. Referential Constraints

- References do not have the same semantics as referential constraints

```
CREATE TABLE T1
    (C1        REAL PRIMARY KEY, ...
CREATE TABLE T2
    (C2        DECIMAL (7,2) PRIMARY KEY, ...
CREATE TABLE T
    (C INTEGER, ...
    FOREIGN KEY (C) REFERENCES T1 (C1) NO ACTION,
    FOREIGN KEY (C) REFERENCES T2 (C2) NO ACTION)
```

- Referential constraints specify inclusion dependencies
  - It is unclear which table to access during dereferencing
- There is no notion of strong typing

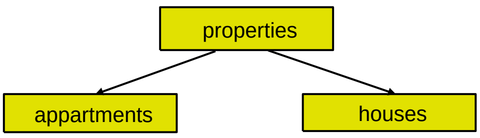
Type Predicate and ONLY on Typed Tables

- Type predicate can be used to restrict selected rows

```
SELECT price, location.city, location.state
FROM properties
WHERE address.city = 'Sacramento'
AND DEREF (oid) IS OF (house)
```

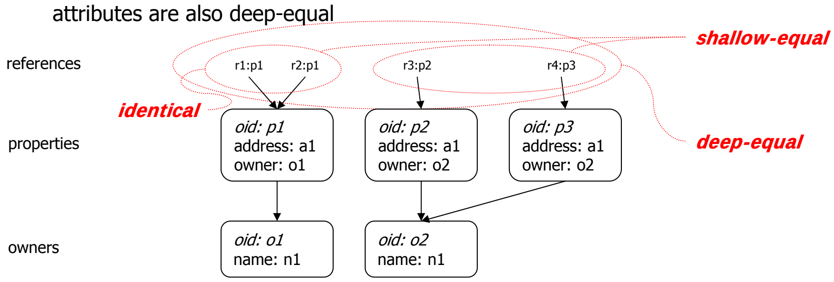
- ONLY restricts selected rows to rows whose most specific type is the type of the typed table

```
SELECT price, location.city, location.state
FROM ONLY (properties)
WHERE address.city = 'Sacramento'
```



Comparing Objects

- Object identity independent from object state
  - separate notions of identical and equal objects
- Identical
  - check whether object identifiers/references are the same
- Forms of equality
  - shallow-equal: object state (attributes, including reference attributes) is the same
  - deep-equal: object state is the same for non-reference attributes, and reference attributes are also deep-equal



Comparing "Objects" in SQL:2003

- Identical: comparing reference values

```
SELECT *
FROM properties p1, p2
WHERE p1.owner = p2.owner
```

- Shallow-equal
  - comparing referenced UDT values
  - need to create an ORDERING for the UDTs with ordering category STATE

```
SELECT *
FROM properties p1, p2
WHERE DEREF(p1.owner) = DEREF(p2.owner)
```

- Deep-qual:
  - comparing referenced UDT values
  - no "built-in" capabilities, requires an ordering based on a user-defined function

Comparison of UDT Values

- CREATE ORDERING statement specifies
  - Which comparison operations are allowed for a user-defined type
  - How such comparisons are to be performed.
    - CREATE ORDERING FOR employee
      EQUALS ONLY BY STATE;
    - CREATE ORDERING FOR complex
      ORDER FULL BY RELATIVE
      WITH FUNCTION complex_order (complex,complex);
- Ordering form:
  - EQUALS ONLY
    - Only comparison operations allowed are =, <>
  - ORDER FULL
    - All comparison operations are allowed

Comparison of UDT Values (cont.)

- Ordering category
  - STATE
    - An ordering function is implicitly created with two UDT parameters and returning Boolean
    - Compares pairwise the UDT attributes
  - RELATIVE
    - User must specify an ordering function with two UDT parameters and returning INTEGER
    - 0 for equal, positive for >, negative for <
  - MAP
    - User must specify an ordering function with one UDT parameter and returning a value of a predefined type
    - Comparisons are made based on the value of the predefined type

Comparison of UDT Values (cont.)

- Ordering category - Rules:
  - STATE cannot be specified for distinct types.
  - STATE and RELATIVE must be specified for the maximal supertype in a type hierarchy.
  - MAP can be specified for more than one type in a type hierarchy, but all such types must specify MAP and all such types must have the same ordering form.
  - STATE is allowed only for EQUALS ONLY.
  - If ORDER FULL is specified, then RELATIVE or MAP must be specified.

Comparison of UDT Values (cont.)

- Comparison type of a given type:
  - The nearest supertype for which a comparison was defined.
  - Comparison form, comparison category, and comparison function of a type are the ordering form, ordering category, ordering function of its comparison type.
- A value of type T1 is comparable to a value of type T2 if
  - T1 and T2 are in the same subtype family.
  - Comparison types of T1 and T2 both specify the same comparison category (i.e., STATE, RELATIVE, or MAP)
- Example
  - Person has subtypes: emp and mgr
  - Person has an ordering form, ordering category, and an ordering function
    - emp and mgr types have none
  - Person is the comparison type of emp and mgr
  - Two emp values, two mgr values, or a value of emp and a value of mgr can be compared.

Comparison of UDT Values (cont.)

- No comparison operations are allowed on values of structured types by default.
- All comparison operations are allowed on values of distinct types by default.
  - Based on the comparison of values of source type.
  - Whenever a distinct type is created, a CREATE ORDERING statement is implicitly executed (SDT is the source type).
  - The ordering function is the system-generated cast function

```
CREATE ORDERING FOR DT
ORDER FULL BY MAP WITH FUNCTION SDT(DT);
```


Comparison of UDT Values (cont.)

- A predicate of the form "V1 = V2" is transformed into the following expression depending on the comparison category:
  - STATE
    - "SF(V1,V2) = TRUE"
    - SF is the comparison function
  - MAP
    - "MF1(V1) = MF2(V2)"
    - MF1 and MF2 are comparison functions
  - RELATIVE
    - "RF(V1,V2) = 0"
    - RF is the comparison function

Comparison of UDT Values (cont.)

- DROP ORDERING
  - Removes the ordering specification for an UDT
    DROP ORDERING FOR employee RESTRICT;
- RESTRICT implies
  - There cannot be any
    - SQL- invoked routine
    - View
    - Constraint
    - Assertion
    - Trigger
  - that has a predicate involving employee values or values of subtypes thereof.

User-defined Casts

- Allow a value of one type to be cast into a value of another type
  - At least one of the types in a user-defined cast must be a user-defined type or a reference type.
    CREATE CAST(t1 AS t2) WITH FUNCTION foo (t1);
    SELECT CAST(c1 AS t2) FROM TAB1;
- May optionally be tagged AS ASSIGNMENT
    CREATE CAST(t1 AS t2) WITH FUNCTION foo (t1) AS ASSIGNMENT;
  - Such casts get invoked implicitly during assignment operations.
  - Above user-defined cast makes the following assignment legal:
    DECLARE v1 t1, v2 t2;
    SET V2 = V1;

User-defined Casts (cont.)

- DROP CAST
  - Removes the user-defined cast
  - Does not delete the corresponding function (only its cast flag is removed)

    DROP CAST (T1 AS T2) RESTRICT;
- RESTRICT implies:
  - There cannot be any
    - Routine
    - View
    - Constraint
    - Assertion
    - Trigger
  - that has
    - An expression of the form "CAST(V1 AS T2)" where V1 is of type T1 or any subtype of T1;
    - A DML statement that implicitly invokes the user-defined cast function.

Summary

- Object-oriented features for a DBMS
  - Extensibility
    - user-defined types, static type checking
  - Types and classes
    - "abstract data types",
    - class as an "object factory", extension (i.e., set of "instances")
  - Type or class hierarchies
    - inheritance, specialization
  - Object identity
    - object exists independent of its value (i.e., identical ≠ equal)
- … still to come
  - Complex objects
    - type constructors: tuple, set, list, array, …
  - Encapsulation
  - Overloading, overriding, late binding
  - Computational completeness
- SQL:2003
  - User-defined types
    - distinct types
  - User-defined structured types
    - value type
    - typed tables
  - Type and table hierarchies
    - inheritance, substitutability
  - Reference types, self-referencing column
    - ordering for UDTs
- … see next chapters