Use of the common dot notation to refer to components: addr.city

- An **array type** may be specified as in
  ```sql
  create type company_type as
  (comp_name varchar(20), location varchar(20) array[10]);
  ```
  location[1] refers to the first value in the array

- Corresponding relations can be created by
  ```sql
  create table employee of employee_type;
  create table company of company_type;
  ```

- Use of an explicit *row* constructor
  ```sql
  create table employee as
  (name varchar(35),
   addr row(street varchar(35), city varchar(25), zipcode char(5)),
   age int);
  ```

- Insertion
  ```sql
  insert into employee values
  ('John Smith', row('Mall Avenue', 'Gainesville', '32711'), 36);
  ```
insert into company values
    (‘XYZ’, array[‘Mall Avenue’, ‘Sales Street’, ‘Sellers Drive’]);
Alternatively:
insert into company values (‘XYZ’, array[]);

update company
    set location = array[‘Mall Avenue’, ‘Sales Street’, ‘Sellers Drive’]
where comp_name = ‘XYZ’;

Later:
update company
    set location[2] = ‘Salesman Street’ where comp_name = ‘XYZ’;

An extension allows one to specify that and how object identifiers are created.

create table employee of employee_type ref is emp_id system generated;
create table company of company_type
    (ref is comp_id system generated, primary key (comp_name));

Syntax: ref is <oid_attribute> <value_generation_method>

Attribute named oid_attribute is used to identify individual tuples in the table. DBMS generates a unique identifier for each tuple.
A component attribute of one tuple may be a reference (specified by using the keyword ref) to a tuple of another table.

Example

```sql
create type employment_type as
  (empl ref (employee_type) scope(employee) references are checked,
   comp ref (company_type) scope(company) references are checked);
create table employment of employment_type;
```

Keyword scope specifies the name of the table whose tuples can be referenced by the reference attribute. This is similar to a foreign key. But the system generated value is used rather than the primary key.

Keyword references are checked indicates that dangling references (i.e., invalid reference values) are not allowed.

Path expressions are built by applying the dot notation. However, for an attribute whose type is ref, the dereferencing symbol -> is used (similar as in the C programming language).
Example

```sql
select e.empl->name
from employment as e
where e.comp->comp_name = 'XYZ';
```

- `r` a reference to a tuple `t`, an attribute in `t`, `r->a` denotes value of `a` in `t`.

Object identifiers can be explicitly declared in the type definition rather than in the table declaration.

Example (change of the definition of `employee_type`)

```sql
create type employee_type as
    (name char(35), age int, emp_id ref (employee_type));
create table employee of employee_type
values for emp_id are system generated;
```

- encapsulation of operations in SQL
- user can create a named user-defined type with own behavioral specification by specifying user-defined functions (methods, operations) in addition to the attributes

In SQL3, a UDT (ADT) is generally defined through
- the specification of a set of declarations for stored attributes that represent the value of the UDT,
– Operations defining equality (equal) and an order (less than) on the UDT,
– Operations defining the behavior of the ADT.

Example: Specify a method for “Extract the apartment number from a string that forms the street attribute of the address_type row type declared before.”

```
create type address_type as (street varchar(35), city varchar(25), zipcode char(5));
method apt_no() returns char(8);
```

The code for implementing the method still has to be written:

```
method
create function apt_no() returns char(8) for address_type as
external name '/x/y/aptno.class' language 'java';
```

Java is the implementation language, the code is stored in a file with the specified pathname.

UDT can have a number of user-defined functions associated with it. Syntax:

```
method <name>(<argument_list>) returns <type>;
```
Built-in functions for UDTs

- **Constructor function** `type_t()` creates and returns a new object (instance) of type `type_t`
- **Observer function** `A` is implicitly created for each attribute `A` to read its value. `A(X)` or `X.A` returns the value of attribute `A` of `type_t` if `X` is of type `type_t`.
- **Mutator function** sets the value of the attribute to a new value (update).

Two types of functions: internal SQL and external

- **Internal functions** are written in the extended SQL/PSM (Persistent Stored Modules) language (not discussed in this class, similar to Oracle’s PL/SQL).
- **External functions** are written in a host language (e.g., Java, C++). Only their signature (interface) appears in the UDT definition. Syntax:

  ```
  declare external <function_name> <signature> language <language_name>;
  ```

Many ORDBMs provide the user with packages of abstract data types (ADTs). They are purchased separately from the basic system.

- Data Blades in Informix Universal Server
- Data Cartridges in Oracle
- Extenders in DB2
UDTs can be used as attribute types in SQL and as parameter types in a function or procedure.

Encapsulation of components (attributes and functions) at different levels
- **public**: These components form the interface of the ADT and are visible outside the ADT definition for all authorised users.
- **private**: These components are totally encapsulated and are only visible within the definition of the ADT containing them.
- **protected**: These components are partially encapsulated. They are visible within their ADT and within the definition of all subtypes of the ADT.

Example
```sql
create type employee_type
  (public
    name char(29), b_address address_type, manager employee_type,
    hiredate date;
  private
    base_salary decimal(7, 2), commission decimal(7, 2);
  public
    function working_years (p employee_type) returns int
    < program code to compute the number of working years >
```
public
  function working_years (p employee_type, y years) returns employee_type
  < program code to update the number of working years >
public
  function salary (p employee_type) returns decimal
  < program code to compute the salary of an employee >
);

Data types for large unstructured complex objects

- Data type `blob` for binary large objects
- Data type `clob` for character large objects
- Example

  create table employees (resume clob(75K), signature blob(1M), picture blob(12M));
- only read and write operations for byte ranges available, no random access to sub-structures
Inheritance

- regarding types and relations expressed by means of the `under` keyword
- Example regarding relations

```sql
create table person (name char(20), sex char(1), age integer);
create table employee under person as (salary float);
create table customer under person as (account integer);
```

Employee inherits all attributes (and methods) of `person` and has an additional attribute `salary`.

- Rules
  - All attributes are inherited.
  - The order of supertypes in the `under` clause determines the inheritance hierarchy.
  - An instance of a subtype can be used in every context in which a supertype instance is used.
  - A subtype can redefine any function that is defined in its supertype, with the restriction that the signature be the same.
  - When a function is called, the best match is selected based on the types of all arguments.
13.3 Informix Universal Server

Introduction

- Combination of relational and object database technologies of two previously existing products
  - relational DBS *Informix*
  - Illustra, originated from the *Postgres* DBMS as a research project at the University of California at Berkeley

- Extension of the relational data model by
  - support for additional or *extensible data types*
  - support for user-defined routines (procedures or functions)
  - inheritance
  - support for indexing extensions
  - *Data Blades* API