Bancos de dados Orientados a Objetos e Objeto-Relacionais

2COP501

Prof. Daniel S. Kaster

Departamento de Computação
Universidade Estadual de Londrina
dskaster@uel.br

Créditos

- Baseado nos slides originais:
  - Elsmasri, Navathe. Database Systems, Pearson.
    - Chapter 20 - Concepts for Object-Oriented Databases
    - Chapter 21 - Object Database Standards, Languages, and Design
  - Banco de Dados II
    - Bacharelado em Ciência da Computação
    - UFCG
    - Prof. Cláudio de Souza Baptista, Ph.D.
- E no documento
  - Oracle Database Object-Relational Developer's Guide 11g Release 2

The OODBMS Manifesto

- “The object-oriented database system manifesto”, or Atkinson'n manifesto (1989)
- This document presented the required feature to a system be considered an OODBMS, such as:
  - manipulation of classes and objects;
  - support for inheritance, encapsulation and polymorphism
  - use of object identifiers and references
  - availability of manipulation languages aware of objects and references (dot/path navigation)

“Popular” OODBMS

- Experimental Systems: Orion at MCC, IRIS at H-P labs, Open-OODB at T.I., ODE at ATT Bell labs, Postgres - Montage - Illustra at UC/B, Encore/Observer at Brown
- Commercial OO Database products: Ontos, Gemstone, O2 (-> Ardent), Objectivity, Objectstore (-> Excelon), Versant, Poet, Jasmine (Fujitsu – GM)

Overview of Object-Oriented Concepts

- MAIN CLAIM: OO databases try to maintain a direct correspondence between real-world and database objects so that objects do not lose their integrity and identity and can easily be identified and operated upon
  - Transient vs. Persistent objects

- Object: Two components: state (value) and behavior (operations).
  - Similar to program variable in programming language, except that it will typically have a complex data structure as well as specific operations defined by the programmer

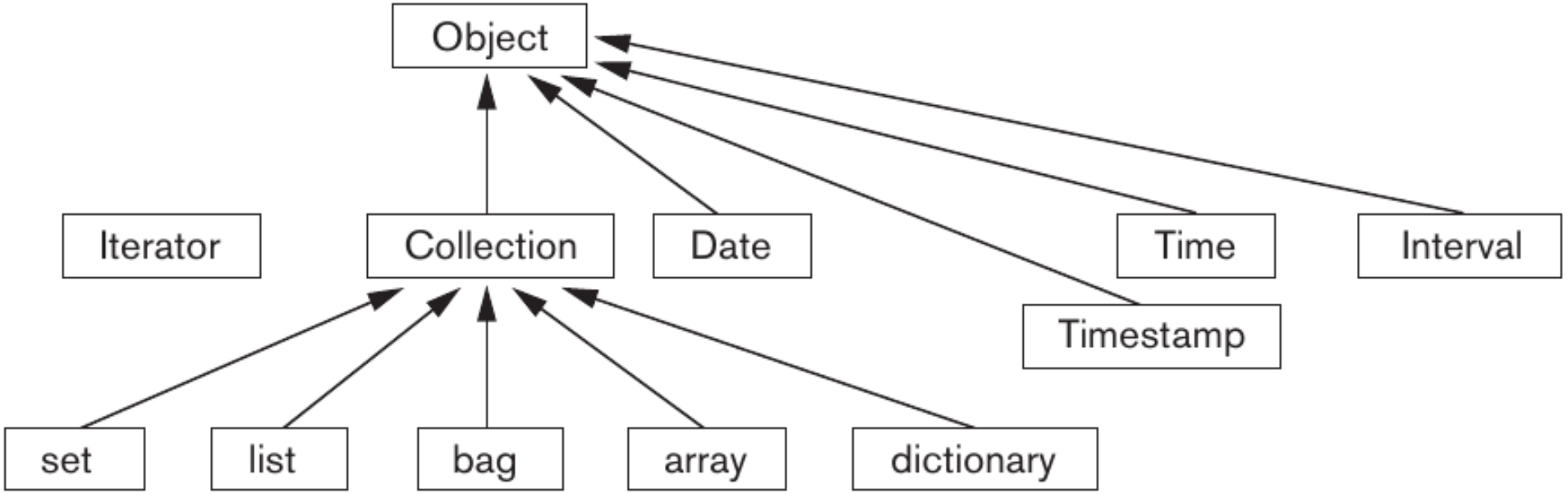
Overview of Object-Oriented Concepts(2)

- In OO databases, objects may have an object structure of arbitrary complexity in order to contain all of the necessary information that describes the object.
- In contrast, in traditional database systems, information about a complex object is often scattered over many relations or records, leading to loss of direct correspondence between a real-world object and its database representation.

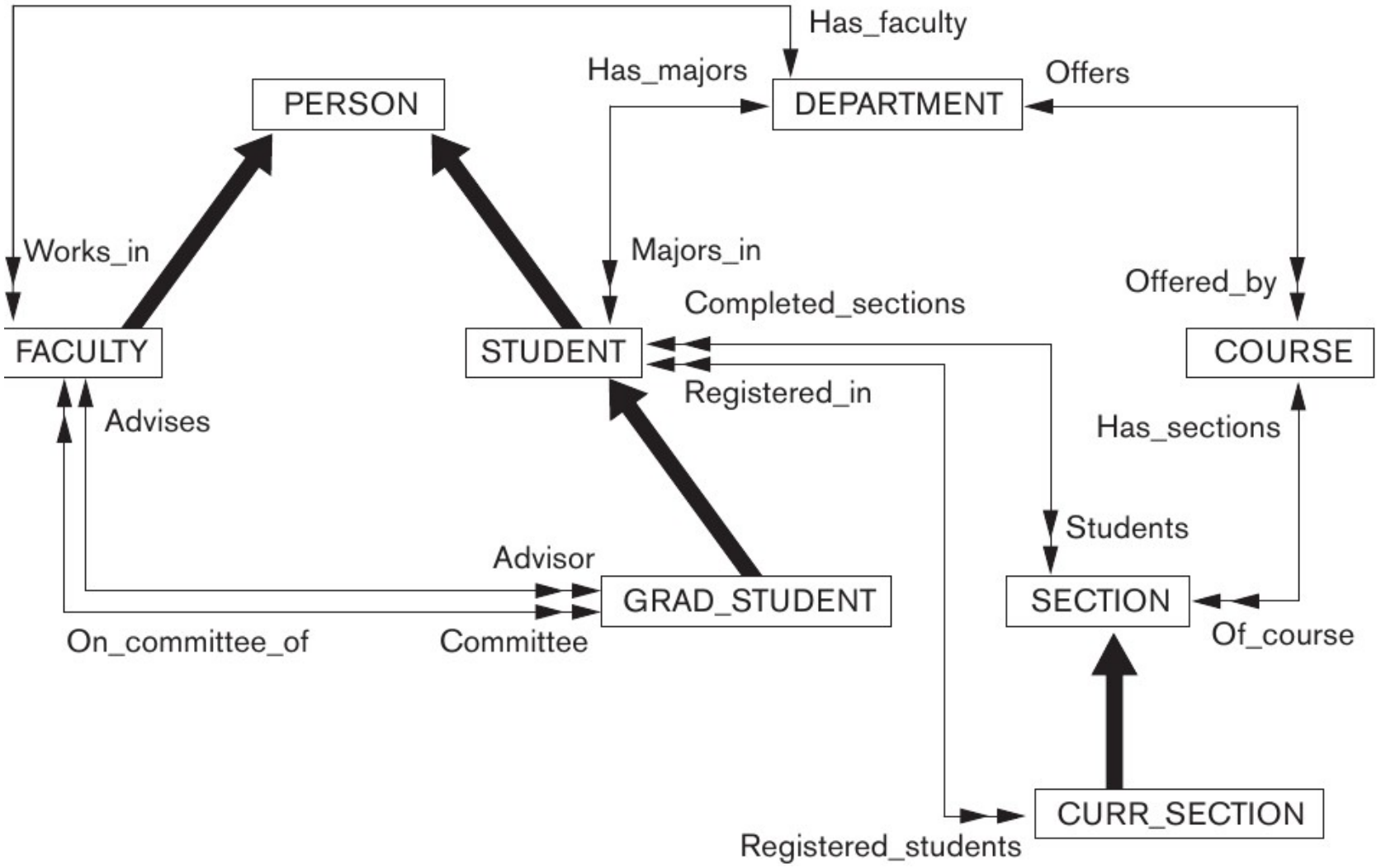
The Object Model of ODMG

- Open Data Management Group standard
  - Provides a standard model for object databases
  - Supports object definition via ODL
  - Supports object querying via OQL
  - Supports a variety of data types and type constructors

Inheritance hierarchy for the ODMG built-in interfaces



Object Definition Language (ODL)



ODL – Example (1)

```
class PERSON
(    extent      PERSONS
     key         Ssn  )
{    attribute   struct Pname {   string   Fname,
                                  string   Mname,
                                  string   Lname  }     Name;
     attribute   string                                 Ssn;
     attribute   date                                   Birth_date;
     attribute   enum Gender{M, F}                      Sex;
     attribute   struct Address {  short    No,
                                   string   Street,
                                   short    Apt_no,
                                   string   City,
                                   string   State,
                                   short    Zip  }     Address;
     short       Age();    };
class FACULTY extends PERSON
(    extent      FACULTY  )
{    attribute   string          Rank;
     attribute   float           Salary;
     attribute   string          Office;
     attribute   string          Phone;
     relationship  DEPARTMENT    Works_in inverse DEPARTMENT::Has faculty;
     relationship  set<GRAD_STUDENT> Advises inverse GRAD_STUDENT::Advisor;
     relationship  set<GRAD_STUDENT> On_committee_of inverse GRAD_STUDENT::Committee;
     void        give_raise(in float raise);
     void        promote(in string new rank);  };
```

ODL – Example (2)

```
class GRADE
(     extent          GRADES  )
{
      attribute       enum GradeValues{A,B,C,D,F,I, P} Grade;
      relationship    SECTION Section inverse SECTION::Students;
      relationship STUDENT Student inverse STUDENT::Completed_sections;  };
class STUDENT extends PERSON
(     extent          STUDENTS  )
{     attribute       string                Class;
      attribute       DEPARTMENT            Minors_in;
      relationship    DEPARTMENT Majors_in inverse DEPARTMENT::Has_majors;
      relationship    set<GRADE> Completed_sections inverse GRADE::Student;
      relationship    set<CURR_SECTION> Registered_in INVERSE CURR_SECTION::Registered_students;
      void            change_major(in string dname) raises(dname_not_valid);
      float           gpa();
      void            register(in short secno) raises(section_not_valid);
      void            assign_grade(in short secno; IN GradeValue grade)
                           raises(section_not_valid,grade_not_valid);  };
```

ODL – Example (3)

```
class DEGREE
{    attribute      string              College;
     attribute      string              Degree;
     attribute      string              Year;     };
class GRAD_STUDENT extends STUDENT
(    extent         GRAD_STUDENTS  )
{    attribute      set<DEGREE>         Degrees;
     relationship   FACULTY Advisor inverse FACULTY::Advises;
     relationship   set<FACULTY>   Committee inverse FACULTY::On_committee_of;
     void           assign_advisor(in string Lname; in string Fname)
                         raises(faculty_not_valid);
     void           assign_committee_member(in string Lname; in string Fname)
                         raises(faculty_not_valid);  };
class DEPARTMENT
(    extent         DEPARTMENTS
     key            Dname )
{    attribute      string              Dname;
     attribute      string              Dphone;
     attribute      string              Doffice;
     attribute      string              College;
     attribute      FACULTY             Chair;
     relationship   set<FACULTY> Has_faculty inverse FACULTY::Works_in;
     relationship   set<STUDENT> Has_majors inverse STUDENT::Majors_in;
     relationship   set<COURSE> Offers inverse COURSE::Offered_by;  };
```

ODL – Example (4)

```
class COURSE
(     extent        COURSES
      key           Cno )
{     attribute     string          Cname;
      attribute     string          Cno;
      attribute     string          Description;
      relationship  set<SECTION> Has_sections inverse SECTION::Of_course;
      relationship  <DEPARTMENT> Offered_by inverse DEPARTMENT::Offers; };
class SECTION
(     extent        SECTIONS )
{     attribute     short           Sec_no;
      attribute     string          Year;
      attribute     enum Quarter{Fall, Winter, Spring, Summer}
                        Qtr;
      relationship  set<GRADE> Students inverse GRADE::Section;
      relationship  course Of_course inverse COURSE::Has_sections;  };
class CURR_SECTION extends SECTION
(     extent        CURRENT_SECTIONS )
{     relationship  set<STUDENT> Registered_students
                        inverse STUDENT::Registered_in
      void          register_student(in string Ssn)
                        raises(student_not_valid, section_full);  };
```

Object Query Language

- OQL is DMG's query language
- OQL works closely with programming languages such as C++
- Embedded OQL statements return objects that are compatible with the type system of the host language
- OQL's syntax is similar to SQL with additional features for objects

Simple OQL Queries

- Basic syntax: select…from…where…

```
select   D.Dname
from     D in DEPARTMENTS
where    D.College = 'Engineering';
```

- An entry point to the database is needed for each query
- An extent name (e.g., departments in the above example) may serve as an entry point

Query results and path expressions

- The result of a query can be of any type that can be expressed in the ODMG object model
- A query does not have to follow the select-from-where structure

Q1: DEPARTMENTS;
Q2: CS_DEPARTMENT.Chair;
Q2A: CS_DEPARTMENT.Chair.Rank;
Q2B: CS_DEPARTMENT.Has_faculty;

* CS_DEPARTMENT is a persistent name given via the database bind operation

Using iterator variables

- Iterator variables is provided to remove the ambiguity regarding the object returned (e.g. a set or a bag)

```
Q3A:   select   F.Rank
       from     F in CS_DEPARTMENT.Has_faculty;

Q3B:   select   distinct F.Rank
       from     F in CS_DEPARTMENT.Has_faculty;
```

Returning complex structures

Q4: CS_DEPARTMENT.Chair.Advises;

Q4A:
```
select struct ( name: struct (last_name: S.name.Lname, first_name:
                              S.name.Fname),
                degrees:( select struct (deg: D.Degree,
                                         yr: D.Year,
                                         college: D.College)
                          from D in S.Degrees ))
from S in CS_DEPARTMENT.Chair.Advises;
```

Equivalent queries using different path expressions

```
Q5A:  select struct ( last_name: S.name.Lname, first_name: S.name.Fname,
                      gpa: S.gpa )
      from          S in CS_DEPARTMENT.Has_majors
      where         S.Class = 'senior'
      order by      gpa desc, last_name asc, first_name asc;

Q5B:  select struct ( last_name: S.name.Lname, first_name: S.name.Fname,
                      gpa: S.gpa )
      from          S in STUDENTS
      where         S.Majors_in.Dname = 'Computer Science' and
                    S.Class = 'senior'
      order by      gpa desc, last_name asc, first_name asc;
```

Aggregate operators over collections

```
Q7: count ( S in Has_minors('Computer Science'));

Q8: avg ( select   S.Gpa
          from     S in STUDENTS
          where    S.Majors_in.Dname = 'Computer Science' and
                   S.Class = 'Senior');

Q17:  select     dept_name, avg_gpa: avg ( select P.gpa from P in partition)
      from       S in STUDENTS
      group by   dept_name: S.Majors_in.Dname
      having     count (partition) > 100;
```

Membership conditions

```
Q10:  select   S.name.Lname, S.name.Fname
      from     S in STUDENTS
      where    'Database Systems I' in
               ( select    C.Section.Of_course.Cname
                 from      C in S.Completed_sections);
```

Abordagem objeto-relacional

- 1989: Manifesto de Atkinson (BDOO)
- 1990: “Third generation database system manifesto”, ou segundo manifesto
  - Reação ao manifesto de Atkinson
  - Argumenta que o modelo relacional é simples, eficiente e que poderia ser facilmente estendido para acomodar objetos complexos
  - Marco inicial dos SGBD objeto-relacionais

Limitações do Modelo Relacional

- Atributos compostos não podem ser representados diretamente
- Atributos multivalorados devem ser diferenciados através de valores únicos e representados em um outro esquema relacional
- Agregações e especializações requerem esquemas de relações individuais equipadas com restrições de integridade especiais
- A introdução de chaves artificiais é necessária se não houverem chaves naturais
- Declarações de tipo são pouco flexíveis
- “Impedance mismatch” com aplicações OO

Proposta da abordagem Objeto-Relacional

- Nova Funcionalidade
  - Aumenta indefinidamente o conjunto de tipos e funções fornecidas pelo SGBD
- Desenvolvimento de aplicações simplificado
  - Reuso de código
- Consistência
  - Permite a definição de padrões, código reusável por várias aplicações

Modelo OR como extensão do relacional

- As extensões incluem mecanismos para permitir aos usuários estender o banco de dados com tipos e funções específicas da aplicação
  - Organização dos dados
  - Linguagem de manipulação

Organização dos dados

- Permite especificar e utilizar tipos abstratos de dados (TADs) da mesma forma que os tipos de dados pré-definidos
  - TADs são tipos de dados definidos pelo usuário que encapsulam comportamento e estrutura interna (atributos)
- A tabela convencional é estendida para permitir a referência de objetos (referência de tipos), TADs e valores alfanuméricos como domínio de colunas

Organização dos dados(2)

- Utiliza referências para representar conexões inter-objetos
  - Torna as consultas baseadas em caminhos de referência mais compactas do que as consultas feitas com junção
- Herança é implementada organizando todos os tipos em hierarquias
- Utiliza os construtores set, list, multiset ou array para organizar coleções de objetos

Linguagem de manipulação

- As extensões incluem consultas envolvendo objetos, atributos multivalorados, TADs, métodos e funções como predicados de busca em uma consulta
- O resultado de uma consulta ainda consiste de relações
  - Um SGBDOR ainda é relacional pois suporta dados armazenados em relações formadas por tuplas e atributos

SQL:1999

- Também conhecida por SQL-3
- Introduziu os conceitos OO no padrão, mas é muito mais do que SQL-92 incrementada com esses conceitos
- Envolve características adicionais que podem ser classificadas em:
  - Relacionais: novos tipos de dados, novos predicados
  - Orientadas a Objetos: tipos de dados definidos pelo usuário, definição de métodos, uso de referências
- É a base para muitos SGBDOR (Oracle, Informix Universal Server, IBM's DB2 Universal Database, entre outros)

Padrão SQL

- Part 1: Framework (SQL/Framework)
- Part 2: Foundation (SQL/Foundation)
- Part 3: Call-Level Interface (SQL/CLI)
- Part 4: Persistent Stored Modules (SQL/PSM)
- Part 9: Management of External Data (SQL/MED)
- Part 10: Object Language Bindings (SQL/OLB)
- Part 11: Information and Definition Schemas (SQL/Schemata)
- Part 13: SQL Routines and Types Using the Java Programming Language (SQL/JRT)
- Part 14: XML-Related Specifications (SQL/XML)

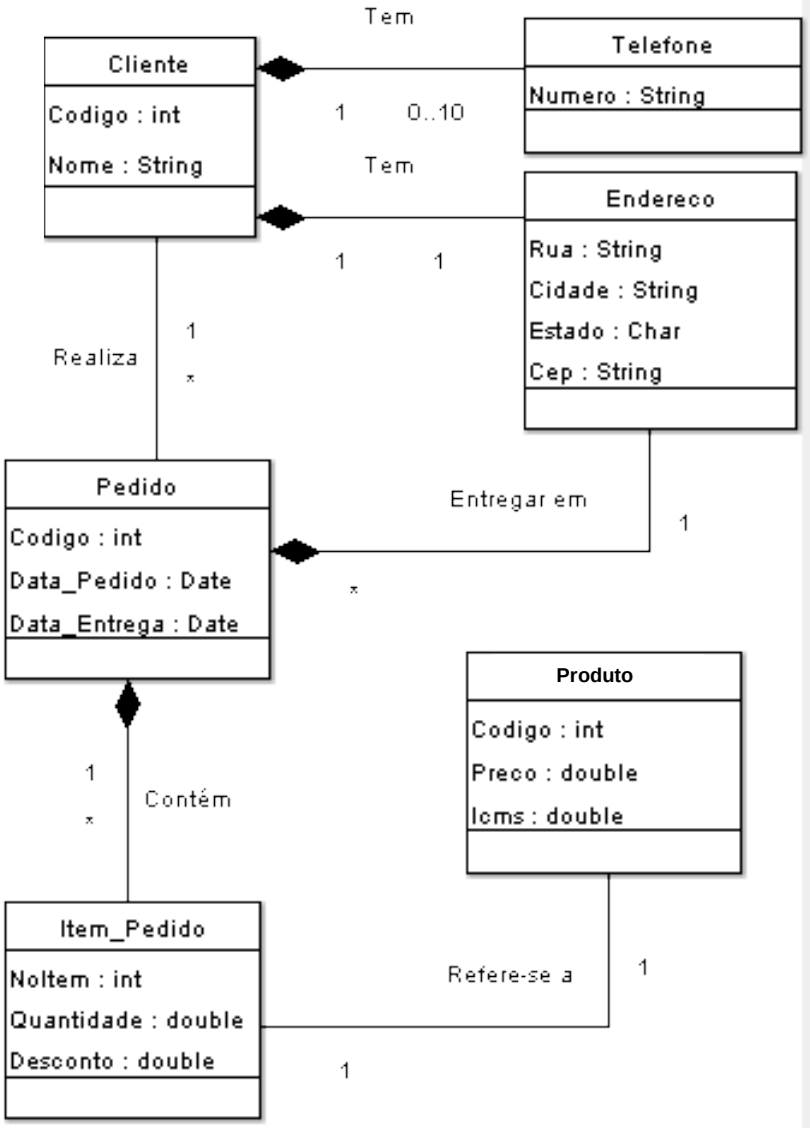
Part 2: SQL/Foundation

- Parte do padrão que trata de Orientação a Objetos:
  - Tipos de dados definidos pelo usuário
  - Atributos & comportamento
  - Encapsulamento: funções & métodos
  - Observers & mutators
  - Hierarquias de tipos (herança simples)
  - User-defined CAST, ordenação
  - Tabelas tipadas & tipos referência

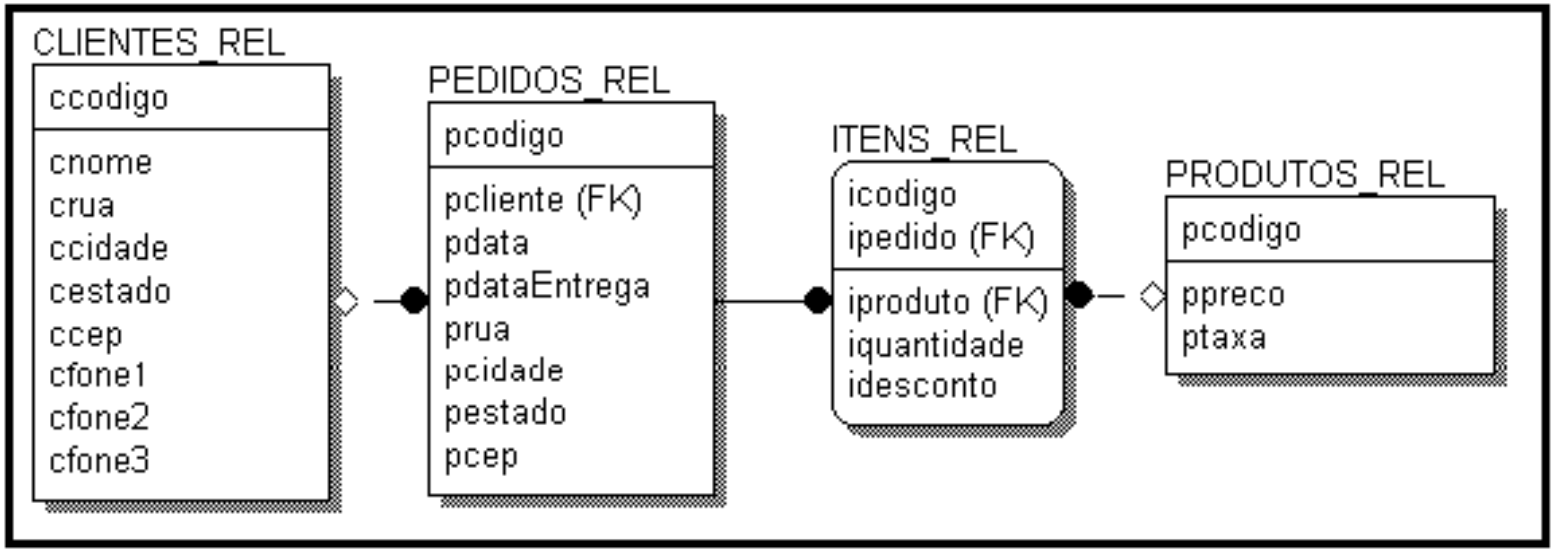
Caso de estudo: Oracle OR

- O Oracle Database é um bom exemplo de implementação de recursos objeto-relacionais do padrão SQL
  - No entanto, há algumas diferenças com relação ao padrão
- O ORACLE oferece diferentes tipos de objetos
  - Tipos de Objetos (TADs)
  - Nested Tables (Tabelas aninhadas)
  - VArrays (Varying Arrays)
  - Large Objects (LOBs)
  - References (REF)
  - Object Views (Visões de Objetos)

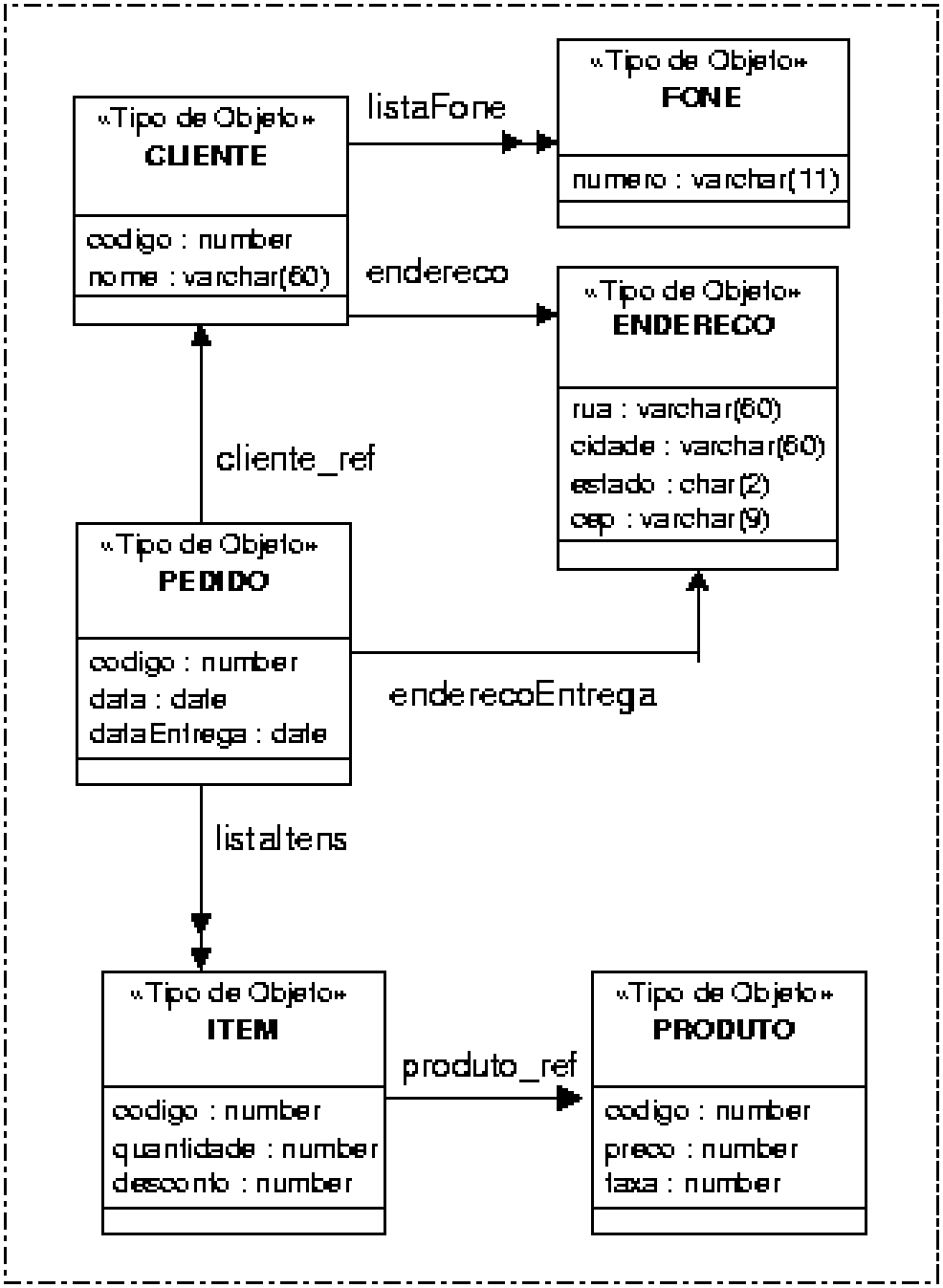
Exemplo - UML


Tem
Cliente
Codigo : int
Nome : String
1
0..10
Telefone
Numero : String
Tem
Endereco
Rua : String
Cidade : String
Estado : Char
Cep : String
1
1
Realiza
1
*
Pedido
Codigo : int
Data_Pedido : Date
Data_Entrega : Date
Entregar em
1
*
Produto
Codigo : int
Preco : double
Icms : double
1
*
Contém
Item_Pedido
NoItem : int
Quantidade : double
Desconto : double
Refere-se a
1
1

Exemplo – Esquema relacional



Exemplo – Esquema objeto-relacional



Exemplo – Implementação no Oracle

```
CREATE TYPE PEDIDO AS OBJECT (
 codigo            NUMBER,
 cliente_ref       REF CLIENTE,
 data              DATE,
 dataEntrega       DATE,
 listaItens        ITEM_LISTA,
 enderecoEntrega ENDERECO );

CREATE TYPE ENDERECO AS OBJECT
(      rua     VARCHAR2(20),
       cidade VARCHAR2(10),
       estado CHAR(2),
       cep     VARCHAR2(10)        );

CREATE TYPE CLIENTE AS OBJECT
(    codigo        NUMBER,
       nome       VARCHAR2(200),
       endereco   ENDERECO,
       listaFone   FONE_LISTA    );
```

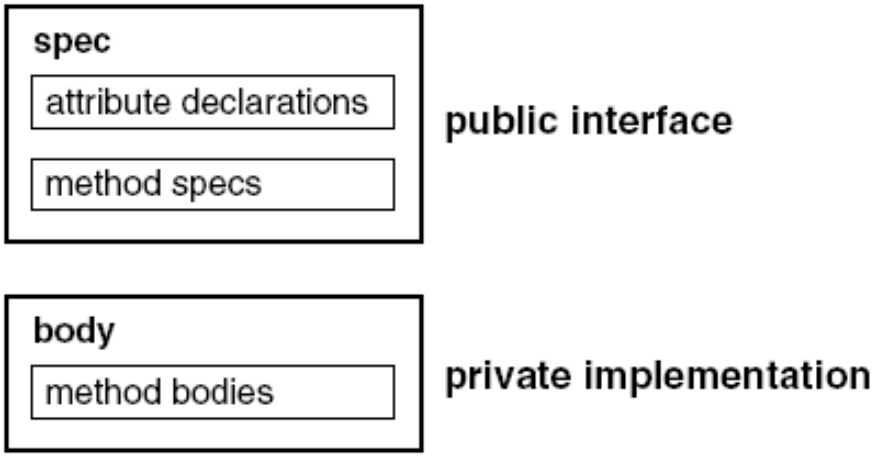
```
CREATE TYPE ITEM AS OBJECT
(   codigo   NUMBER,
    produto_ref  REF PRODUTO,
    quantidade   NUMBER,
    desconto     NUMBER     );

CREATE TYPE ITEM_LISTA AS TABLE
OF ITEM

CREATE TYPE PRODUTO AS OBJECT (
 codigo     NUMBER,
 Preco      NUMBER,
 Taxa       NUMBER  ) ;

CREATE TYPE FONE_LISTA AS
VARRAY(10) OF VARCHAR2(20);
```

Tipos de objetos Oracle

- Tipo de objeto no Oracle é um tipo abstrato de dados (TAD), ou seja um Structured Type em SQL:1999
- TAD é um tipo de dado definido pelo usuário que encapsula propriedades (atributos) e comportamento (métodos)
- Corresponde ao “molde” de um objeto
- Não aloca espaço de armazenamento
- Não pode armazenar dados

Estrutura e sintaxe

- Um tipo de objeto no Oracle possui a seguinte estrutura



- Sintaxe resumida

```
CREATE [OR REPLACE] TYPE nome_tipo AS OBJECT (
[lista de atributos]
[lista de métodos]
);
```

Atributos

- Atributos podem ser
  - De tipos convencionais
  - De tipos de objetos
  - Referências
  - Coleções

Métodos

- São funções ou procedimentos que são declarados na definição de um tipo de objeto
- Exigem o uso de parênteses (mesmo sem parâmetros)
- Podem ser
  - MEMBER ou STATIC
  - MAP ou ORDER (para ordenação)
  - Construtor

Métodos MEMBER

- São os métodos mais comuns
- Implementam as operações das instâncias do tipo
- São invocados através da qualificação de objeto (notação ponto): objeto.método()
- SELF não precisa ser declarado, mas pode ser
  - Neste caso, deve ser sempre o primeiro parâmetro

Métodos construtores

- O construtor padrão é criado implicitamente ao criar um tipo de objeto
- Pode haver mais de um construtor
  - O nome de um método construtor deve ser exatamente igual ao nome do tipo

Métodos STATIC

- São invocados nos tipos de dados, não nas instâncias
- Usados para operações que são globais ao tipo e não precisam se reportar a uma instância particular
- Não possui parâmetro SELF
- Invocado da seguinte forma: type_name.method()

Métodos MAP e ORDER

- São funções opcionais para comparar objetos
- Por padrão, o ORACLE implementa a comparação STATE do SQL:1999
  - Indica se um objeto é igual ou não de outro baseado na comparação de cada atributo
- MAP e ORDER são mutuamente exclusivos!
- O método MAP é mais indicado para ordenar grandes conjuntos de dados, pois mapeia todos os objetos e depois ordena com com base nos valores escalares, enquanto o método ORDER precisa ser chamado repetidamente

Métodos ORDER

- Implementa o RELATIVE WITH do SQL:1999, retornando negativo, zero ou positivo
- Exige como parâmetro um objeto do mesmo tipo
- Compara o objeto corrente com o objeto passado por parâmetro

Métodos MAP

- Implementa MAP do SQL:1999, retornando um valor de tipo built-in
- Não exige parâmetro
- Usado para mapear um tipo de objeto para um tipo ordenável

Exemplo de tipo de objeto

```
CREATE TYPE person_typ AS OBJECT (
  idno            NUMBER,
  first_name      VARCHAR2(20),
  last_name       VARCHAR2(25),
  email           VARCHAR2(25),
  phone           VARCHAR2(20),
  MAP MEMBER FUNCTION get_idno RETURN NUMBER,
  MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ ));

CREATE TYPE BODY person_typ AS
  MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
  BEGIN
    RETURN idno;
  END;
  MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ ) IS
  BEGIN
    -- use the PUT_LINE procedure of the DBMS_OUTPUT package to display details
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(idno) || ' ' || first_name || ' ' || last_name);
    DBMS_OUTPUT.PUT_LINE(email || ' '  || phone);
  END;
END;
```

Tabelas de objetos

- O Oracle suporta 2 tipos de tabelas:
  - Tabela Relacional
  - Tabela de Objetos (Object Table)
- Uma tabela de objetos é um tipo especial de tabela que lida com objetos (“row objects”) e fornece uma visão relacional dos atributos desses objetos
- Pelo princípio da substituição (herança de tipos), uma coluna/linha do tipo t pode conter instâncias de quaisquer subtipos de t

Tabelas de objetos vs. Tabelas relacionais

- Uma tabela de objetos difere de uma tabela relacional em vários aspectos:
  - Armazenam apenas objetos
  - Cada linha de uma tabela de objetos possui um identificador de objeto (OID), definido pelo Oracle quando a linha é inserida na tabela
  - Um OID é um ponteiro para um objeto “linha” (ROW Object);
  - Os objetos (row objects) de uma tabela de objetos podem ser referenciadas por outros objetos do banco de dados
  - Provê uma visão relacional dos objetos armazenados
- Uma tabela relacional pode ter atributos de tipos de objetos

Exemplo: tabela relacional com um atributo de tipo de objeto

```
CREATE TABLE contacts (
  contact          person_typ,
  contact_date     DATE );

INSERT INTO contacts VALUES (
  person_typ (65, 'Verna', 'Mills', 'vmills@example.com', '1-650-555-0125'),
 '24 Jun 2003' );

SELECT c.contact.get_idno() FROM contacts c;
```

Exemplo: tabela de objetos

```
CREATE TABLE person_obj_table OF person_typ;

INSERT INTO person_obj_table VALUES (
       person_typ(101, 'John', 'Smith', 'jsmith@example.com', '1-650-555-0135') );

SELECT VALUE(p) FROM person_obj_table p
        WHERE p.last_name = 'Smith';

DECLARE
  person person_typ;
BEGIN -- PL/SQL block for selecting a person and displaying details
  SELECT VALUE(p) INTO person FROM person_obj_table p WHERE p.idno = 101;
  person.display_details();
END;
```

Identificadores de objetos

- Uma tabela de objetos contém uma coluna gerada pelo SGBD contendo o OID do “row object”
  - O oid de um objeto é único e imutável.
- Sobre essa coluna de OID é também criado automaticamente um índice para prover acesso eficiente sobre o objeto através do OID
  - A coluna de OID é equivalente a se ter uma coluna extra de 16 bytes para chave primária.
- Um OID permite que um “row object” seja referenciado em atributos de outros objetos ou em colunas de tabelas relacionais
  - Um tipo pré-definido REF é capaz de representar tais referências.

Referências a objetos

- São ponteiros lógicos para “row objects”
- São usadas para fazer referência a partir do OID do objeto
- Oferece acesso rápido/direto ao objeto
- Não garante integridade referencial
  - Uma REF pode apontar para qualquer objeto do tipo apontado
  - É possível definir restrições referenciais (no estilo chave estrangeira)

Exemplo de uso de REFs

```
CREATE TYPE emp_person_typ AS OBJECT (
  name     VARCHAR2(30),
  manager  REF emp_person_typ );
/
CREATE TABLE emp_person_obj_table OF emp_person_typ;

INSERT INTO emp_person_obj_table VALUES (
   emp_person_typ ('John Smith', NULL));

INSERT INTO emp_person_obj_table
  SELECT emp_person_typ ('Bob Jones', REF(e))
    FROM emp_person_obj_table e
    WHERE e.name = 'John Smith';

select * from emp_person_obj_table e;


NAME        MANAGER
----------- ----------------------------------------------------------
John Smith
Bob Jones   0000220208424E801067C2EABBE040578CE70A0707424E8010
            67C1EABBE040578CE70A0707
```

Definindo o escopo de uma REF

- É possível restringir o escopo de uma referência a uma tabela de objetos específica
- Scoped REFs exigem menos espaço de armazenamento e normalmente permitem um acesso mais eficiente
  - O ponteiro não precisa identificar a tabela de destino, ocupando menos bytes, e o escopo de busca é limitado

```
CREATE TABLE contacts_ref (
  contact_ref   REF person_typ SCOPE IS person_obj_table,
  contact_date  DATE );

INSERT INTO contacts_ref
  SELECT REF(p), '26 Jun 2003'
    FROM person_obj_table p
    WHERE p.idno = 101;
```

Deferência de referências

- O operador DEREF retorna um objeto referenciado por uma coluna do tipo REF

```
SELECT DEREF(e.manager) FROM emp_person_obj_table e;

DEREF(E.MANAGER)(NAME, MANAGER)
--------------------------------------------------------------------------------
EMP_PERSON_TYP('John Smith', NULL)
```

- Dereferência implícita de uma REF

```
SELECT e.name, e.manager.name FROM emp_person_obj_table e
  WHERE e.name = 'Bob Jones';
```

Dangling REFs

- É possível que uma referência torne-se inválida (dangling), caso o objeto referenciado seja excluído ou caso o usuário não tenha os privilégios necessários para acessar o objeto
- Dereferenciar uma referência dangling retorna um objeto null
- Para testar dangling REFs usa-se o predicado SQL

```
IS DANGLING
```

Obtendo referências para objetos

```
DECLARE
  person_ref REF person_typ;
  person person_typ;
BEGIN

  SELECT REF(p) INTO person_ref
    FROM person_obj_table p
    WHERE p.idno = 101;

   select deref(person_ref) into person from dual;
   person.display_details();

END;
```

Exemplo: redefinindo o tipo person

```
CREATE OR REPLACE TYPE person_typ AS OBJECT (
  idno            NUMBER,
  name            VARCHAR2(30),
  phone           VARCHAR2(20),
  MAP MEMBER FUNCTION get_idno RETURN NUMBER,
  MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ ) );
/

CREATE OR REPLACE TYPE BODY person_typ AS
  MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
  BEGIN
    RETURN idno;
  END;
  MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ ) IS
  BEGIN
    -- use the PUT_LINE procedure of the DBMS_OUTPUT package to display details
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(idno) || ' - '  || name || ' - '  || phone);
  END;
END;
```

Tratamento de nulos em objetos

```
CREATE TABLE contacts (
  contact        person_typ,
  contact_date   DATE );
```

- Row object com atributos nulos

```
INSERT INTO contacts VALUES (
  person_typ (NULL, NULL, NULL), '24 Jun 2003' );
```

- Row object nula

```
INSERT INTO contacts VALUES (
  NULL, '24 Jun 2003' );
```

Restrições para tabelas de objetos

- Chave primária

```
CREATE OR REPLACE TYPE location_typ AS OBJECT (
  building_no  NUMBER,
  city         VARCHAR2(40) );
/

CREATE OR REPLACE TYPE office_typ AS OBJECT (
  office_id    VARCHAR(10),
  office_loc   location_typ,
  occupant     person_typ );
/

CREATE TABLE office_tab OF office_typ (
             office_id      PRIMARY KEY );
```

Restrições para tabelas de objetos(s)

- Restrições UNIQUE e CHECK

```
CREATE TABLE department_mgrs (
  dept_no      NUMBER PRIMARY KEY,
  dept_name    CHAR(20),
  dept_mgr     person_typ,
  dept_loc     location_typ,
  CONSTRAINT   dept_loc_cons1
      UNIQUE (dept_loc.building_no, dept_loc.city),
  CONSTRAINT   dept_loc_cons2
       CHECK (dept_loc.city IS NOT NULL) );

INSERT INTO department_mgrs VALUES
          ( 101, 'Physical Sciences',
           person_typ(65,'Vrinda Mills', '1-1-650-555-0125'),
           location_typ(300, 'Palo Alto'));
```

Índices em atributos de tipos de objeto

- É possível definir índices em atributos escalares de ”leaf-level” de atributos de tipos de objeto

```
CREATE TABLE department_loc (
  dept_no     NUMBER PRIMARY KEY,
  dept_name   CHAR(20),
  dept_addr   location_typ );

CREATE INDEX  i_dept_addr1
         ON   department_loc (dept_addr.city);

INSERT INTO department_loc VALUES
          ( 101, 'Physical Sciences',
           location_typ(300, 'Palo Alto'));
INSERT INTO department_loc VALUES
          ( 104, 'Life Sciences',
           location_typ(400, 'Menlo Park'));
INSERT INTO department_loc VALUES
          ( 103, 'Biological Sciences',
           location_typ(500, 'Redwood Shores'));
```

Triggers em tabelas de objetos

```
CREATE TABLE movement (
     idno            NUMBER,
     old_office      location_typ,
     new_office      location_typ );

CREATE TRIGGER trigger1
  BEFORE UPDATE
              OF  office_loc
              ON  office_tab
   FOR EACH ROW
           WHEN  (new.office_loc.city = 'Redwood Shores')
   BEGIN
     IF :new.office_loc.building_no = 600 THEN
      INSERT INTO movement (idno, old_office, new_office)
       VALUES (:old.occupant.idno, :old.office_loc, :new.office_loc);
     END IF;
   END;
/
INSERT INTO office_tab VALUES
    ('BE32', location_typ(300, 'Palo Alto' ),person_typ(280, 'John Chan',
       '415-555-0101'));

UPDATE office_tab set office_loc =location_typ(600, 'Redwood Shores')
  where office_id = 'BE32';
```

Uso de aliases para qualificação de nomes

- Aliases são opcionais caso seja acessado um atributo ”top-level” e usualmente requeridos, caso contrário

```
#1 SELECT idno FROM person_obj_table;          --Correct

#2 SELECT contact.idno FROM contacts;          --Illegal
#3 SELECT contacts.contact.idno FROM contacts; --Illegal
#4 SELECT p.contact.idno FROM contacts p;      --Correct
```

Métodos MEMBER

```
CREATE OR REPLACE TYPE solid_typ AS OBJECT (
  len    INTEGER,
  wth    INTEGER,
  hgt    INTEGER,
  MEMBER FUNCTION surface RETURN INTEGER,
  MEMBER FUNCTION volume RETURN INTEGER,
  MEMBER PROCEDURE display (SELF IN OUT NOCOPY solid_typ) );
/

CREATE OR REPLACE TYPE BODY solid_typ AS
  MEMBER FUNCTION volume RETURN INTEGER IS
  BEGIN
    RETURN len * wth * hgt;
 -- RETURN SELF.len * SELF.wth * SELF.hgt; -- equivalent to previous line
  END;
  MEMBER FUNCTION surface RETURN INTEGER IS
  BEGIN -- not necessary to include SELF in following line
    RETURN 2 * (len * wth + len * hgt + wth * hgt);
  END;
  MEMBER PROCEDURE display (SELF IN OUT NOCOPY solid_typ) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Length: ' || len || ' - '  || 'Width: ' || wth
                         || ' - '  || 'Height: ' || hgt);
    DBMS_OUTPUT.PUT_LINE('Volume: ' || volume || ' - ' || 'Surface area: '
                         || surface);
  END;
END;
```

Acesso aos métodos de um tipo de objeto

```
CREATE TABLE solids of solid_typ;
INSERT INTO solids VALUES(10, 10, 10);
INSERT INTO solids VALUES(3, 4, 5);
SELECT * FROM solids;
SELECT s.volume(), s.surface() FROM solids s WHERE s.len = 10;
DECLARE
  solid solid_typ;
BEGIN -- PL/SQL block for selecting a solid and displaying details
  SELECT VALUE(s) INTO solid FROM solids s WHERE s.len = 10;
  solid.display();
END;
```

Métodos MAP

```
CREATE OR REPLACE TYPE rectangle_typ AS OBJECT (
  len NUMBER,
  wid NUMBER,
  MAP MEMBER FUNCTION area RETURN NUMBER);
/

CREATE OR REPLACE TYPE BODY rectangle_typ AS
  MAP MEMBER FUNCTION area RETURN NUMBER IS
  BEGIN
     RETURN len * wid;
  END area;
END;
```

Métodos ORDER

```
DROP TYPE location_typ FORCE;
-- above necessary if you have previously created object
CREATE OR REPLACE TYPE location_typ AS OBJECT (
  building_no  NUMBER,
  city         VARCHAR2(40),
  ORDER MEMBER FUNCTION match (l location_typ) RETURN INTEGER );
/
CREATE OR REPLACE TYPE BODY location_typ AS
  ORDER MEMBER FUNCTION match (l location_typ) RETURN INTEGER IS
  BEGIN
    IF building_no < l.building_no THEN
      RETURN -1;                -- any negative number will do
    ELSIF building_no > l.building_no THEN
      RETURN 1;                 -- any positive number will do
    ELSE
      RETURN 0;
    END IF;
  END;
END;
```

Herança de tipos

- O Oracle suporta herança simples de tipos
- Há uma diferença do padrão SQL:1999, pois o Oracle não requer herança explicitamente nas tabelas, mas apenas nos tipos
  - Modelo mais simples, embora menos flexível
- Os tipos derivados (sub-tipos) herdam os atributos e métodos dos tipos ancestrais (super-tipos)
- Os sub-tipos podem acrescentar novos atributos ou métodos e/ou redefinir os métodos dos super-tipos
- FINAL e NOT FINAL
  - Definem se um tipo pode ser especializado ou um método sobrescrito
  - Por padrão, tipos são FINAL e métodos NOT FINAL

Exemplo de herança de tipos

```
DROP TYPE person_typ FORCE;
-- above necessary if you have previously created object

CREATE OR REPLACE TYPE person_typ AS OBJECT (
 idno            NUMBER,
 name            VARCHAR2(30),
 phone           VARCHAR2(20),
 MAP MEMBER FUNCTION get_idno RETURN NUMBER,
 MEMBER FUNCTION show RETURN VARCHAR2)
 NOT FINAL;
/

CREATE OR REPLACE TYPE BODY person_typ AS
 MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
 BEGIN
   RETURN idno;
 END;
-- function that can be overriden by subtypes
 MEMBER FUNCTION show RETURN VARCHAR2 IS
 BEGIN
   RETURN 'Id: ' || TO_CHAR(idno) || ', Name: ' || name;
 END;
```

Definindo subtipos

```
CREATE TYPE student_typ UNDER person_typ (
   dept_id NUMBER,
   major VARCHAR2(30),
   OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2)
   NOT FINAL;
CREATE TYPE BODY student_typ AS
 OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2 IS
 BEGIN
    RETURN (self AS person_typ).show || ' -- Major: ' || major ;
 END;
```

```
END;
```

```
DROP TYPE employee_typ FORCE;
-- if previously created
CREATE OR REPLACE TYPE employee_typ UNDER person_typ (
    emp_id NUMBER,
    mgr VARCHAR2(30),
    OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2);

    CREATE OR REPLACE TYPE BODY employee_typ AS
      OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2 IS
      BEGIN
        RETURN (SELF AS person_typ).show|| ' -- Employee Id: '
               || TO_CHAR(emp_id) || ', Manager: ' || mgr ;
      END;

    END;
```

Invocação/Expressão generalizada

- Usando invocação generalizada

```
DECLARE
 myvar student_typ := student_typ(100, 'Sam', '6505556666', 100, 'Math');
 name VARCHAR2(100);
BEGIN
 name := (myvar AS person_typ).show; --Generalized invocation
END;
```

- Usando expressão generalizada

```
DECLARE
 myvar2 student_typ := student_typ(101, 'Sam', '6505556666', 100, 'Math');
 name2 VARCHAR2(100);
BEGIN
 name2 := person_typ.show((myvar2 AS person_typ)); -- Generalized expression
END;
```

Invocação com sobrecarga

```
INSERT INTO person_obj_table
  VALUES (person_typ(12, 'Bob Jones', '650-555-0130'));

INSERT INTO person_obj_table
  VALUES (student_typ(51, 'Joe Lane', '1-650-555-0140', 12, 'HISTORY'));

INSERT INTO person_obj_table
  VALUES (employee_typ(55, 'Jane Smith', '1-650-555-0144',
                       100, 'Jennifer Nelson'));

SELECT p.show() FROM person_obj_table p;
```

Id: 12, Name: Bob Jones
Id: 51, Name: Joe Lane -- Major: HISTORY
Id: 55, Name: Jane Smith -- Employee Id: 100, Manager: Jennifer Nelson

Limitação de substituição

- Usa-se o predicado IS OF

```
SELECT VALUE(p)
  FROM person_obj_table p
WHERE VALUE(p) IS OF (student_typ);

DECLARE
  var person_typ;
BEGIN
  var := employee_typ(55, 'Jane Smith', '1-650-555-0144', 100, 'Jennifer Nelson');
  IF var IS OF (employee_typ, student_typ) THEN
     DBMS_OUTPUT.PUT_LINE('Var is an employee_typ or student_typ object.');
  ELSE
     DBMS_OUTPUT.PUT_LINE('Var is not an employee_typ or student_typ object.');
  END IF;
END;
```

Limitação de substituição

- O predicado IS OF também pode ser usado para limitar a substituição na definição de tabelas de objetos

```
DROP TABLE office_tab;
-- if previously created
CREATE TABLE office_tab OF office_typ
  COLUMN occupant IS OF (ONLY employee_typ);
```

- O predicato TREAT permite tratar um objeto como um subtipo

```
SELECT TREAT(VALUE(p) AS student_typ)
  FROM person_obj_table p
WHERE VALUE(p) IS OF(ONLY student_typ);

SELECT name, TREAT(VALUE(p) AS student_typ).major major
  FROM person_obj_table p;
```

Tipos de coleção

- O Oracle suporta 2 tipos de coleção
  - VARRAY: coleção ordenada de elementos, com número fixo de elementos
    - Adequado para manipular a coleção toda de uma vez
  - Nested tables: coleção não ordenada de elementos e ”ilimitada” quanto ao número de elementos
    - Permite acessar elementos individuais da coleção
    - Permite indexação da coleção
    - Acesso usualmente convertido em JOIN

Coleções nested table

```
CREATE TYPE people_typ AS TABLE OF person_typ; -- nested table type
/
CREATE TABLE people_tab (
    group_no NUMBER,
    people_column people_typ )  -- an instance of nested table
    NESTED TABLE people_column STORE AS people_column_nt; -- storage table for NT

INSERT INTO people_tab VALUES (
            100,
            people_typ( person_typ(1, 'John Smith', '1-650-555-0135'),
                        person_typ(2, 'Diane Smith', NULL)));
```

Uso de nested tables para representar associações com cardinalidade *

```
CREATE TABLE department_persons (
  dept_no    NUMBER PRIMARY KEY,
  dept_name  CHAR(20),
  dept_mgr   person_typ DEFAULT person_typ(10,'John Doe',NULL),
  dept_emps  people_typ DEFAULT people_typ() ) -- instance of nested table type
  NESTED TABLE dept_emps STORE AS dept_emps_tab;

INSERT INTO department_persons VALUES
   ( 101, 'Physical Sciences', person_typ(65,'Vrinda Mills', '1-650-555-0125'),
          people_typ( person_typ(1, 'John Smith', '1-650-555-0135'),
                      person_typ(2, 'Diane Smith', NULL) ) );
INSERT INTO department_persons VALUES
  ( 104, 'Life Sciences', person_typ(70,'James Hall', '1-415-555-0101'),
    people_typ() ); -- an empty people_typ table
```

Nesting/unnesting

- Uso do predicado TABLE permite uma visão relacional de um row object

```
SELECT d.dept_emps
  FROM department_persons d;

DEPT_EMPS(IDNO, NAME, PHONE)
-----------------------------------------------------------
PEOPLE_TYP(PERSON_TYP(1, 'John Smith', '1-650-555-0135'),
PERSON_TYP(2, 'Diane Smith', '1-650-555-0135'))

SELECT e.*
  FROM department_persons d, TABLE(d.dept_emps) e;

 IDNO        NAME                              PHONE
---------- ------------------------------ ---------------
         1 John Smith                     1-650-555-0135
         2 Diane Smith                    1-650-555-0135
```

Coleções VARRAY

- Cada elemento de um VARRAY é acessível diretamente indexando a sua posição no array
- Um VARRAY é tipicamente armazenado in-row
  - Estilo BLOB
  - Se o tamanho é grande (maior que 4k), pode ser armazenado off-row

Manipulando VARRAYs

```
CREATE TYPE dnames_var IS VARRAY(7) OF VARCHAR2(30);
/
CREATE TABLE depts (region VARCHAR2(25), dept_names dnames_var);
BEGIN
   INSERT INTO depts VALUES('Europe', dnames_var('Shipping','Sales','Finance'));
   INSERT INTO depts VALUES('Americas', dnames_var('Sales','Finance','Shipping'));
   INSERT INTO depts
     VALUES('Asia', dnames_var('Finance','Payroll','Shipping','Sales'));
   COMMIT;
END;
/
DECLARE
   new_dnames dnames_var := dnames_var('Benefits', 'Advertising', 'Contracting',
                                       'Executive', 'Marketing');
   some_dnames dnames_var;
BEGIN
   UPDATE depts SET dept_names  = new_dnames WHERE region = 'Europe';
   COMMIT;
   SELECT dept_names INTO some_dnames FROM depts WHERE region = 'Europe';
   FOR i IN some_dnames.FIRST .. some_dnames.LAST
   LOOP
      DBMS_OUTPUT.PUT_LINE('dept_names = ' || some_dnames(i));
   END LOOP;
END;
```

Coleções nested table multinível

```
CREATE TYPE location_typ AS OBJECT (
  location_id      NUMBER(4),
  street_address   VARCHAR2(40),
  postal_code      VARCHAR2(12),
  city             VARCHAR2(30),
  state_province   VARCHAR2(25));
/

CREATE TYPE nt_location_typ AS TABLE OF location_typ;  -- nested table type
/

CREATE TYPE country_typ AS OBJECT (
  country_id     CHAR(2),
  country_name   VARCHAR2(40),
  locations      nt_location_typ); -- inner nested table
/

CREATE TYPE nt_country_typ AS TABLE OF country_typ;  -- multilevel collection type
/

CREATE TABLE region_tab (
  region_id     NUMBER,
  region_name   VARCHAR2(25),
  countries     nt_country_typ) -- outer nested table
  NESTED TABLE countries STORE AS nt_countries_tab
   (NESTED TABLE locations STORE AS nt_locations_tab);
```

Unnesting coleções nested table multinível

```
SELECT r.region_name, c.country_name, l.location_id
  FROM region_tab r, TABLE(r.countries) c, TABLE(c.locations) l;
```

REGION_NAME	COUNTRY_NAME	LOCATION_ID
Europe	Italy	1000
Europe	Italy	1100
Europe	Switzerland	2900
Europe	Switzerland	3000
Europe	United Kingdom	2400
Europe	United Kingdom	2500
Europe	United Kingdom	2600

table

```
SELECT l.location_id, l.city
  FROM region_tab r, TABLE(r.countries) c, TABLE(c.locations) l;
```

LOCATION_ID	CITY
1000	Roma
1100	Venice
2900	Geneva
3000	Bern
2400	London
2500	Oxford
2600	Stretford

Operações acessando elementos individuais de uma coleção

```
INSERT INTO TABLE(SELECT d.dept_emps
                  FROM department_persons d
                  WHERE d.dept_no = 101)
   VALUES (5, 'Kevin Taylor', '1-408-555-0199');

UPDATE TABLE(SELECT d.dept_emps
                  FROM department_persons d
                  WHERE d.dept_no = 101) e
   SET VALUE(e) = person_typ(5, 'Kevin Taylor', '1-408-555-0199')
   WHERE e.idno = 5;

DELETE FROM TABLE(SELECT d.dept_emps
                  FROM department_persons d
                  WHERE d.dept_no = 101) e
   WHERE e.idno = 5;
```

Visões de objetos

- São tabelas de objetos (object table) virtuais
- Cada linha (row) de uma Object View é um objeto, podendo-se, portanto, invocar seus métodos e acessar seus atributos
- São úteis no mapeamento de Relacional par a Objetos, pois dá uma “cara” de objetos a uma tabela puramente relacional.

Exemplo de visão de objetos

```
CREATE TABLE dept (
    deptno        NUMBER PRIMARY KEY,
    deptname      VARCHAR2(20),
    deptstreet    VARCHAR2(20),
    deptcity      VARCHAR2(10),
    deptstate     CHAR(2),
    deptzip       VARCHAR2(10));

CREATE TYPE address_t AS OBJECT (
   street    VARCHAR2(20),
    city     VARCHAR2(10),
    state    CHAR(2),
    zip      VARCHAR2(10));
/
CREATE TYPE dept_t AS OBJECT (
   deptno      NUMBER,
   deptname    VARCHAR2(20),
   address     address_t );
/

CREATE VIEW dept_view OF dept_t WITH OBJECT IDENTIFIER (deptno) AS
    SELECT d.deptno, d.deptname,
      address_t(d.deptstreet,d.deptcity,d.deptstate,d.deptzip) AS
      deptaddr
      FROM dept d;
```

Exemplo de visão de objetos(2)

```
insert into dept values(1,'Sales','500 Oracle pkwy','Redwood S','CA','94065');
insert into dept values(2,'ST','400 Oracle Pkwy','Redwood S','CA','94065');
insert into dept values(3,'Apps','300 Oracle pkwy','Redwood S','CA','94065');

select * from dept_view;

    DEPTNO DEPTNAME
---------- --------------------
ADDRESS(STREET, CITY, STATE, ZIP)
------------------------------------------------------------
         1 Sales
ADDRESS_T('500 Oracle pkwy', 'Redwood S', 'CA', '94065')

         2 ST
ADDRESS_T('400 Oracle Pkwy', 'Redwood S', 'CA', '94065')

         3 Apps
ADDRESS_T('300 Oracle pkwy', 'Redwood S', 'CA', '94065')
```