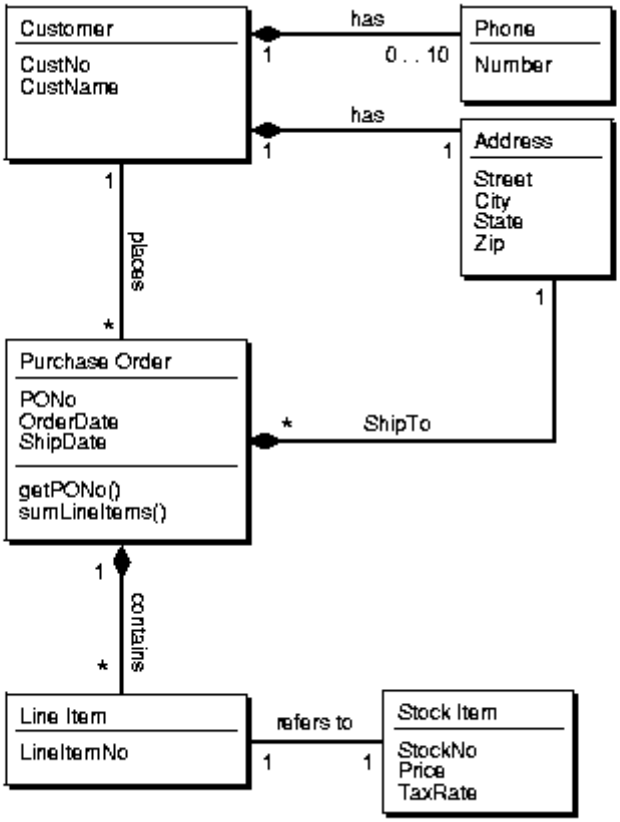
Customer
CustNo
CustName
has
1
0 . . 10
Phone
Number
has
1
1
Address
Street
City
State
Zip
1
places
*
1
Purchase Order
PONo
OrderDate
ShipDate
getPONo()
sumLineItems()
*
ShipTo
1
contains
*
Line Item
LineItemNo
refers to
1
1
Stock Item
StockNo
Price
TaxRate

```
CREATE TABLE Customer_reltab (
  CustNo              NUMBER NOT NULL,
  CustName            VARCHAR2(200) NOT NULL,
  Street              VARCHAR2(200),
  City                VARCHAR2(200),
  State               CHAR(2),
  Zip                 VARCHAR2(20),
  Phone1              VARCHAR2(20),
  Phone2              VARCHAR2(20),
  Phone3              VARCHAR2(20),
  PRIMARY KEY (CustNo)
  ) ;

CREATE TABLE PurchaseOrder_reltab (
  PONo                NUMBER NOT NULL,
  CustNo              NUMBER NOT NULL,
  OrderDate           DATE NOT NULL,
  ShipDate            DATE,
  ShipTo_Street       VARCHAR2(200),
  …
  PRIMARY KEY (PONo),
  FOREIGN KEY (CustNo) REFERENCES Customer_reltab
  );

CREATE TABLE StockItem_reltab (
  StockNo             NUMBER NOT NULL,
  Price               NUMBER NOT NULL,
  …
  PRIMARY KEY (StockNo)
  );

CREATE TABLE LineItem_reltab (
  PONo                NUMBER NOT NULL,
  LineItemNo          NUMBER NOT NULL,
  StockNo             NUMBER NOT NULL,
  …
  PRIMARY KEY (PONo, LineItemNo),
  FOREIGN KEY (PONo) REFERENCES PurchaseOrder_reltab,
  FOREIGN KEY (StockNo) REFERENCES StockItem_reltab
);
```

Objektrelationale DBS unterstützen folgende Datentypen:

- **Strukturen / Klassen (Objekttypen)**
- **Referenzen**
- **Listen**

Implementierung ist noch(?) nicht standardisiert!

Oracle:

Objekttypen:
CREATE TYPE ... AS OBJECT …

Referenzen:
REF *Objekttyp*

Listen:
CREATE TYPE … AS … VARRAY(n) OF …
CREATE TYPE … AS TABLE OF …

Objekttypen

CREATE TYPE ... AS OBJECT:

Definition von

- **Attributen**
- **Methoden (Funktionen / Prozeduren)**
  - **Instanzmethoden**
  - **Klassenmethoden**

Implementierung der Methoden

- **PL/SQL (im Type Body)**
- **Java (siehe Anhang)**
- **C/C++ (hier nicht weiter betrachtet)**

Löschen und Ändern von Objekttypen kann aufgrund bestehender Abhängigkeiten zu Problemen führen.
Daher sollte das Objekttyp Design stabil sein!

AUTHID Klausel beim CREATE TYPE bestimmt, ob die Methoden im Schema

- **des aktueller Benutzers (CURRENT_USER) => Standard**
  oder
- **des Objekttyps (DEFINER)**

ausgeführt werden

Objekttabellen werden via Objekttyp definiert

OBJECTID kann definiert werden als

- **PRIMARY KEY**
- **SYSTEM GENERATED**

Zugriff auf Objekttabellen muß für Attribute und Instanzmethoden über über Tupelvariable ((Tabellen)Alias) erfolgen:

Tupelvariable.Attribut
Tupelvariable.Methode (Instanzmethoden)
(Zugriff auf Klassenmethoden mit Objekttyp.Methode)

REF:

Referenz auf einen Objekttyp(!)

Auf Tabellenebene ist Einschränkung auf eine referenzierte Tabelle möglich

- **SCOPE Klausel**
- **FOREIGN KEY Constraint**

Im Gegensatz zu FOREIGN KEY Constraints sind bei Verwendung von SCOPE dangling references möglich!
(Abfrage mit IS DANGLING)

REF(v) liefert zu einer Tupelvariablen die Referenz
DEREF(r) liefert zu einer Referenz das Objekt
VALUE(v) liefert zu einer Tupelvariablen das Objekt

Zugriff via Referenz.Attribut / Instanzmethode

Listentypen (VARRAY; Nested TABLE):

Basistyp kann jeder objektrelationale Datentyp sein (Oracle 9i)
Falls der Basistyp kein Objekttyp ist, wird intern der Pseudo Objekttyp COLUMN_VALUE verwendet!

Lesender Zugriff über TABLE Konstrukt

Ergebnis eines Selects kann durch
CAST (MULTISET(select ...) AS Listentyp)
einem Attribut vom Listentyp zugewiesen werden

VARRAY:

CREATE TYPE ... AS VARRAY (n) OF …

(Maximale) Größe des Arrays wird bei der Definition festgelegt

Speicherung erfolgt als RAW oder BLOB

VARRAYs können nur als Ganzes geändert werden
Konstruktor

NESTED TABLE:

CREATE TYPE … AS TABLE OF …

Kardinalität nicht beschränkt

Speicherung in Tabellen => STORE Klausel
RETURN AS LOCATOR / VALUE

Datenmanipulation über TABLE Konstrukt möglich

```
CREATE TYPE Address_objtyp AS OBJECT (
  Street                VARCHAR2(200),
  City                  VARCHAR2(200),
  State                 CHAR(2),
  Zip                   VARCHAR2(20)
  )
/

CREATE TYPE PhoneList_vartyp AS VARRAY(3) OF VARCHAR2(20)
/

CREATE TYPE Customer_objtyp AS OBJECT (
  CustNo                NUMBER,
  CustName              VARCHAR2(200),
  Address_obj           Address_objtyp,
  PhoneList_var         PhoneList_vartyp
  )
/

CREATE TABLE Customer_objtab OF Customer_objtyp (
  CustNo PRIMARY KEY,
  CustName NOT NULL
)
  OBJECT IDENTIFIER IS PRIMARY KEY
/
```

```
 CREATE TYPE StockItem_objtyp AS OBJECT (
  StockNo NUMBER,
  Price NUMBER,
  …
/

CREATE TABLE StockItem_objtab OF StockItem_objtyp (
  StockNo Primary Key,
  Price NOT NULL
)
  OBJECT IDENTIFIER IS PRIMARY KEY
/

CREATE TYPE LineItem_objtyp AS OBJECT (
  LineItemNo NUMBER,
  Stock_ref REF StockItem_typ,
  Quantity NUMBER
)
/

CREATE TYPE LineItemList_tabtyp AS TABLE OF LineItem_objtyp
/
```

```
CREATE TYPE PurchaseOrder_objtyp AS OBJECT (
  PONo NUMBER,
  Cust_ref REF Customer_objtyp,
  OrderDate DATE,
  ShipDate DATE,
  LineItemList_tab LineItemList_tabtyp,
  ShipToAddr_obj Address_objtyp
)
/

CREATE TABLE PurchaseOrder_objtab OF PurchaseOrder_objtyp
  (
  Cust_ref NOT NULL,
  OrderDate NOT NULL,
  PRIMARY KEY (PONo),
  FOREIGN KEY (Cust_ref) REFERENCES Customer_objtab
  )
  OBJECT IDENTIFIER IS PRIMARY KEY
  NESTED TABLE LineItemList_tab STORE AS PoLine_tab (
    (PRIMARY KEY (NESTED_TABLE_ID, LineItemNo)))
  RETURN AS LOCATOR;

ALTER TABLE PoLine_tab ADD (
SCOPE FOR (Stock_ref) is Stockitem_objtab
);
```

```
ALTER TYPE PurchaseOrder_objtyp
  ADD  MEMBER FUNCTION sumLineItems RETURN NUMBER
;

CREATE OR REPLACE TYPE BODY PurchaseOrder_objtyp AS

MEMBER FUNCTION getPONo RETURN NUMBER IS
  BEGIN
  RETURN PONo;
  END;

MEMBER FUNCTION sumLineItems RETURN NUMBER IS
  I INTEGER;
  StockItem StockItem_objtyp;
  Total NUMBER := 0;
  BEGIN
  IF (UTL_COLL.IS_LOCATOR(LineItemList_tab))
  THEN
    SELECT SUM(L.Quantity * L.Stock_ref.Price) INTO Total
    FROM
      TABLE (CAST (LineItemList_tab AS LineItemList_tabtyp)) L;
  ELSE
    FOR I in 1..SELF.LineItemList_tab.COUNT LOOP
      UTL_REF.SELECT_OBJECT
        (LineItemList_tab(i).Stock_ref, StockItem);
      Total:=
        Total +SELF.LineItemList_tab(i).Quantity * StockItem.Price;
    END LOOP;
  END IF;
  RETURN Total;
  END;

END;
/
```

Migration Relational => Objektrelational (Datenebene):

```
INSERT INTO Customer_objtab
SELECT
  CustNo,
  CustName,
  Address_objtyp(street, city, state, zip),
  PhoneList_vartyp(Phone1, Phone2, Phone3)
FROM customer_reltab;

INSERT INTO StockItem_objtab
SELECT
  StockNo,
  Price,
  ...
FROM Stockitem_reltab;

INSERT INTO PurchaseOrder_objtab
SELECT
  po.PONo,
  REF(c),
  po.OrderDate,
  po.ShipDate,
  CAST (
    MULTISET (
      select
        li.LineItemNo,
        REF(si),
        li.Quantity
      FROM LineItem_reltab li, StockItem_reltab si
      WHERE li.PONo = po.PONo
      AND li.StockNo = si.StockNo
      )
    AS LineItemList_tabtyp),
  Address_objtyp (
      po.ShipTo_street, po.ShipTo_city,
      po.ShipTo_state, po.ShipTo_zip
  )
FROM PurchaseOrder_reltab po, Customer_objtab c
WHERE po.CustNo = c.CustNo;
```

Migration durch Verwendung von Objekt Views:

- **Views, die einem Objekttyp zugeordnet sind**
- **können i.a. nicht direkt geändert werden**
- **Änderungen sind aber – wie für alle Views – möglich durch INSTEAD OF Trigger**

```
CREATE VIEW Customer_objview OF Customer_objtyp
  WITH OBJECT IDENTIFIER (CustNo) AS
SELECT
  CustNo,
  CustName,
  Address_objtyp(street, city, state, zip),
  PhoneList_vartyp(Phone1, Phone2, Phone3)
FROM customer_reltab;

CREATE VIEW StockItem_objview OF StockItem_objtyp
  WITH OBJECT IDENTIFIER (StockNo) AS
SELECT
  StockNo,
  Price,
  ...
FROM Stockitem_reltab;

CREATE VIEW PurchaseOrder_objview OF PurchaseOrder_objtyp
  WITH OBJECT IDENTIFIER (PONo) AS
SELECT
  po.PONo,
  MAKE_REF(Customer_objview, po.Custno)
  po.OrderDate,
  po.ShipDate,
  CAST (
    MULTISET (
      SELECT
        li.LineItemNo,
        MAKE_REF(StockItem_objview, li.Stockno),
        li.Quantity
      FROM LineItem_reltab li
      WHERE li.PONo = po.PONo
       )
    AS LineItemList_tabtyp
    ),
    Address_objtyp (
      po.ShipTo_street, po.ShipTo_city,
      po.ShipTo_state, po.ShipTo_zip)
FROM PurchaseOrder_reltab po;
```

SQL Beispiele:

Bestimme Kundennummer und –name der Kunden aus Mannheim:

```
SELECT c.CustNo, c.CustName
FROM Customer_objtab c
WHERE c.address_obj.city = 'Mannheim';
```

Ändere die Telefonnummer des Kunden mit der Kundennummer 1 auf ('111','222','333'):

```
UPDATE Customer_objtab c
SET c.phonelistvar = phonelist_vartyp('111','222','333')
WHERE c.CustNo = 1;
```

Bestimme alle Kunden, die die Telefonnummer '111' haben:

```
SELECT c.*
FROM Customer_objtab c
WHERE '111' in (SELECT * from c.phonelist_var);
```

Füge den Auftrag mit der Auftragsnummer 1 für den Kunden mit der Kundennummer 1 mit aktuellem Datum als Auftragsdatum, keinen Lieferinformationen und leerer Auftragspositionsliste ein:

```
INSERT into PurchaseOrder_objtab p
SELECT  1, REF(c), sysdate, null, LineItemList_tabtyp(), null
FROM Customer_objtab c
WHERE c.Custno = 1;
```

Füge eine Auftragsposition ein (Positionsnummer 1, Teilnummer 1, Menge 100):

```
INSERT into TABLE
(
SELECT p.LineItemList_tab
FROM PurchaseOrder_objtab p
WHERE p.PONo=1
)
SELECT 1, REF(s), 100
FROM StockItem_objtab s
WHERE s.StockNo=1;
```

Ändere die Menge auf 200:

```
UPDATE TABLE
(
SELECT p.LineItemList_tab
FROM PurchaseOrder_objtab p
WHERE p.PONo=1
) t
SET t.Quantity = 200
WHERE t.LineItemNo = 1;
```

oder

```
SET VALUE(t) =
LineItemList_tabtyp(
ListItem_objtyp(
1,
(SELECT REF(s) FROM StockItem_objtab s WHERE s.StockNo=1),
200));
```

Füge den Auftrag 2 für den gleichen Kunden ein, inklusive einer Auftragsposition für Teil 2 mit Menge 100:

```
INSERT INTO PurchaseOrder_objtab p
SELECT 2, REF(c), sysdate, null,
CAST (MULTISET(select 1, REF(s), 100 FROM StockItem_objtab s
where s.StockNo=2) AS LineItemList_tabtyp),
null
FROM Customer_objtab c where c.CustNo=1;
```

Bestimme zu dem Kunden 1 Auftragsnummer, Auftragsdatum und –volumen aller Aufträge:

```
SELECT p.PONo, p.OrderDate, p.sumLineItems()
FROM PurchaseOrder_objtab p
WHERE p.Cust_ref.CustNo = 1;
```

Bestimme statt des Auftragsvolumens sämtliche Auftragspositionen:

Geschachtelt:

```
SELECT p.PONo, p.OrderDate, p.LineItemList_tab
FROM PurchaseOrder_objtab p
WHERE p.Cust_ref.CustNo = 1;
```

Flach:

```
SELECT p.PONo, p.OrderDate, li.LineItemNo, li.Stock_ref.StockNo,
li.Quantity, li.Stock_ref.Price
FROM PurchaseOrder_objtab p, TABLE (p.LineItemList_tab) li
WHERE p.Cust_ref.CustNo = 1;
```

Bestimme Kundennummer und Namen der Kunden, die Teil 1 bestellt haben:

```
SELECT DISTINCT p.Cust_ref.CustNo, p.Cust_ref.CustName
FROM PurchaseOrder_objtab p, TABLE(p.LineItemList_tab) li
WHERE li.Stock_ref.StockNo=1;
```

JAVA:

JDBC definiert Interfaces, die Objekttypen, Referenzen und Listentypen entsprechen:

java.sql.Struct
java.sql.Ref
java.sql.Array

Allerdings werden diese Interfaces von den von Oracle mitgelieferten JDBC Treibern nicht vollständig unterstützt, so daß proprietäre Erweiterungen im Umgang mit Referenzen und Listentypen notwendig sind.

Ferner gibt es die Möglichkeit, Objekttypen auf JAVA Klassen abzubilden (die das SQLData Interface implementieren), allerdings muß für Referenzen und Listentypen wieder auf proprietäre Erweiterungen zurückgegriffen werden.

Ist man bereit, dies zu akzeptieren, so bietet das Oracle Tool JPublisher (im Oracle Modus) die Möglichkeit, zu jedem Objekttyp eine zugehörige JAVA Klasse zu erzeugen, so daß ein Objekt des Objekttyps auf eine Instanz dieser JAVA Klasse abgebildet wird und ein einfacher Zugriff auf sämtliche Attribute möglich ist (inklusive Attributen, die Objekttypen, Referenzen und Listentypen repräsentieren)

jpub
-user=…/…
-sql=Objekttyp:Klasse
...
-usertypes=oracle
-methods=none
-url=…

Vererbung

CREATE TYPE Subtype UNDER Supertype
Attribute
Methoden

Objekttypen und Methoden können als

- **[NOT] FINAL (default: FINAL)**
- **[NOT] INSTANTIABLE (default: INSTANTIABLE)**

definiert werden

Redefinition von Methoden:
OVERRIDING ...

Ersetzbarkeit:

Supertyp erlaubt => Subtypen erlaubt

Ausnahmen:
- NOT SUBSTITUTABLE AT ALL LEVELS
für Objekttabellen und objektwertige Attribute
bzw.
- Typüberprüfung für objektwertige Attribute

Typüberprüfung:
***Objekt* / *Referenz* IS [NOT] OF (Kommaliste *von Objekttypen)* | (ONLY *Objekttyp*)**

Downcast:
TREAT (*Objekt* AS *Objekttyp*)
TREAT (*Referenz* AS REF *Objekttyp*)

Vererbung ist möglich auf der Ebene von
Objekttypen,
Objekt Views (mit Extent Inklusion),
nicht für Objekt Tabellen!

Beispiel:

Supertyp Partner
Subtypen Kunde, Lieferant

(Annahme: Kunden und Lieferanten stellen eine disjunkte Zerlegung von Partner dar)

```
Create TYPE partner_objtyp AS OBJECT (
PartnerId          NUMBER,
PartnerName     VARCHAR2(20),
...
) NOT FINAL INSTANTIABLE;

CREATE TYPE kunde_objtyp UNDER partner_objtyp (
Kreditvolumen  NUMBER
);

CREATE TYPE lieferant_objtyp UNDER partner_objtyp (
Lieferantstatus  CHAR(1)
);
```

Modellierungsvariante 1:

Eine Objekttabelle, in der alle Partner stehen (heterogen!)

```
CREATE TABLE partner_objtab OF partner_objtyp
  (PRIMARY KEY (PartnerId)
  OBJECT IDENTIFIER IS PRIMARY KEY
;
```

```
INSERT INTO partner_objtab VALUES (partner_objtyp(…));
INSERT INTO partner_objtab VALUES (kunde_objtyp(…));
INSERT INTO partner_objtab VALUES (lieferant_objtyp(…));
```

```
SELECT VALUE(p)
FROM partner_objtab p
```

liefert alle Partner Objekte (heterogen)!

```
SELECT *
FROM partner_objtab
```

=> Einschränkung auf Attribute von partner_objtyp

```
SELECT p.PartnerId, p.PartnerName,
  TREAT (VALUE(p) AS kunde_objtyp).Kreditvolumen
FROM partner_objtab p
WHERE VALUE(p) IS OF (kunde_objtyp))
```

liefert Id, Name, Kreditvolumen aller Kunden

Modellierungvariante 2:

Objekt Tabellen Kunde, Lieferant
(Globale Eindeutigkeit der Partner Id muß sichergestellt sein!)

```
CREATE TABLE kunde_objtab of kunde_objtyp
(PRIMARY KEY (PartnerId))
  OBJECT IDENTIFIER IS PRIMARY KEY
  ...
CREATE TABLE lieferant_objtab of lieferant_objtyp
(PRIMARY KEY (PartnerId))
  OBJECT IDENTIFIER IS PRIMARY KEY
  ...


CREATE VIEW partner_objview of TYPE partner_objtyp
  WITH OBJECT IDENTIFIER (PartnerId) AS
/* Leere Tabelle vom Typ partner_objtyp */;

CREATE VIEW kunde_objview of  TYPE kunde_objtyp
UNDER partner_objview
AS
SELECT *
FROM kunde_objtab p
;

CREATE VIEW lieferant_objview of  TYPE lieferant_objtyp
UNDER partner_objview
AS
SELECT *
FROM lieferant_objtab p
;
```

Abfragen funktionieren analog zu Modellierungsvariante 1,
allerdings mit den Objekt Views anstelle der Objekttabellen.

Modellierungsvariante 3:

Relationale Tabellen Kunde, Lieferant
(Globale Eindeutigkeit der Partner Id muß sichergestellt sein!)

```
CREATE VIEW partner_objview OF partner_objtyp
WITH OBJECT IDENTIFIER (PartnerId)
AS
/* Leere Tabelle vom Typ partner_objtyp */

CREATE VIEW kunde_objview OF kunde_objtyp
UNDER partner_view
AS
SELECT kunde_objtyp(...)
FROM kunde;

CREATE VIEW lieferant_objview OF lieferant_objtyp
UNDER partner_view
AS
SELECT lieferant_objtyp(...)
FROM lieferant;
```

Abfragen funktionieren analog zu Modellierungsvariante 2.

JAVA

Unterscheidung der Objekttypen in einer Objekthierarchie

- **SQL Funktion SYS_TYPEID, die auf ein Objekt anzuwenden ist und den Objekttypen liefert**

- **JAVA Reflection**

Anhang: Implementierung von Methoden in Java

// Java Klasse

```
import java.sql.*;
import oracle.jdbc.driver.*;
import java.math.*;

public class Employee implements SQLData {

  // Attribute
  private BigDecimal empno;
  private String ename;
  ...
  private BigDecimal sal;
  private BigDecimal comm;

  // wages liefert Gehalt plus Kommision
  public BigDecimal wages() {
    BigDecimal pay = sal;
    if (comm != null) pay = pay.add(comm);
    return pay;
  }

  // Gehaltserhöhung um einen gegebenen Wert
  public void raiseSal(BigDecimal amount) {
    sal = sal.add(amount);
  }

  // Implementierung SQLData interface.

  String sql_type;

  public String getSQLTypeName() throws SQLException {
    return sql_type;
  }

  public void readSQL(SQLInput stream, String typeName)
    throws SQLException {
    sql_type = typeName;
    empno = stream.readBigDecimal();
    ename = stream.readString();
    ...
  }

  public void writeSQL(SQLOutput stream) throws SQLException {
    stream.writeBigDecimal(empno);
    stream.writeString(ename);
  ...
  }
}
```

Installieren von Java Objekten in der Datenbank

loadjava dient zum Installieren der Java-Klasse(n) in der Datenbank (Source-Datei, Class-Datei, Archiv):

loadjava –user .../... *Java-Klasse(n)*

Registrierung des SQL Objekt Typen

```
CREATE TYPE Employee AS OBJECT (
  empno    NUMBER(4),
  ename    VARCHAR2(10),
  ...
  sal      NUMBER(7,2),
  comm     NUMBER(7,2)

  MEMBER FUNCTION wages RETURN NUMBER
    AS LANGUAGE JAVA
    NAME 'Employee.wages() return java.math.BigDecimal',
  MEMBER PROCEDURE raise_sal (r NUMBER)
    AS LANGUAGE JAVA
    NAME 'Employee.raiseSal(java.math.BigDecimal)'
);
```