Advance Database Management System
PL/SQL Tutorial

Learning Objectives

To know about:

- Procedure
- Function
- Cursor
- Record
- Trigger
- Package

Procedure

Procedure

- Named PL/SQL block which performs one or more specific task
- Similar to a procedure in other programming languages

Further Information:

- http://plsql-tutorial.com/plsql-procedures.htm
- https://www.tutorialspoint.com/plsql/plsql_procedures.htm

Procedure Example


```
CREATE OR REPLACE PROCEDURE adjust_hisal(
   in_hisal IN salgrade.hisal%TYPE
)
IS
BEGIN

  UPDATE salgrade
  SET hisal ='8888'
  WHERE hisal= in_hisal;
END;


begin
adjust_hisal('9999');
end

select * from salgrade;
rollback
```

Function

Function

- A function is a named PL/SQL Block which is similar to a procedure
- The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value

Further Information:

- http://plsql-tutorial.com/plsql-functions.htm
- https://www.tutorialspoint.com/plsql/plsql_functions.htm

Function Example


```
CREATE OR REPLACE FUNCTION totalemp
RETURN number IS
  total number(12) := 0;
BEGIN
  SELECT count(*) into total
  FROM emp;
  RETURN total;
END;
/
DECLARE
  c number(12);
BEGIN
  c := totalemp();
  dbms_output.put_line('Total No of Employees: ' || c);
END;
```

Cursor

Cursor

- A cursor is a temporary work area created in the system memory when a SQL statement is executed
- This temporary work area is used to store the data retrieved from the database, and manipulate this data

Further Information:

- http://plsql-tutorial.com/plsql-cursors.htm
- https://www.tutorialspoint.com/plsql/plsql_cursors.htm

Cursor Example One Row Print


```
declare
d_name dept.dname%type;
d_loc dept.loc%type;
cursor c_dept is
select dname,loc from dept;
begin
open c_dept;
fetch  c_dept into d_name,d_loc;
dbms_output.put_line(d_name||' '||d_loc);
close c_dept;
end
/
```

Cursor Example Multiple Row Print


```
declare
d_name dept.dname%type;
d_loc dept.loc%type;
cursor c_dept is
select dname,loc from dept;
begin
open c_dept;
loop
fetch  c_dept into d_name,d_loc;
exit when c_dept%notfound;
dbms_output.put_line(d_name||' '||d_loc);
end loop;
close c_dept;
end
/
```

Cursor Example Multiple Row Print


```
declare
lo_sal salgrade.losal%type;
cursor c_salgrade is
select losal from salgrade;
begin
open c_salgrade;
loop
fetch c_salgrade into lo_sal;
exit when c_salgrade%notfound;
dbms_output.put_line(lo_sal);
end loop;
close c_salgrade;
end
```

Record

Record

- A record is a data structure that can hold data items of different kinds
- Records consist of different fields, similar to a row of a database table

Further Information:

- http://plsql-tutorial.com/plsql-records.htm
- https://www.tutorialspoint.com/plsql/plsql_records.htm

Record Example One row print


```
declare
dept_rec dept%rowtype;
begin
select * into dept_rec from dept
where dname='ACCOUNTING';
dbms_output.put_line(dept_rec.loc)
  ;
end
/
```

Record Example multiple row print


```
declare
dept_rec dept%rowtype;
begin
for dept_rec
in(select * from dept)
loop
dbms_output.put_line(dept_rec.loc||'
  '||dept_rec.dname);
end loop;
end
/
```

Cursor Based record multiple/single row print


```
declare
cursor c_emp is
select * from emp;
rec_emp emp%rowtype;
begin
open c_emp;
--loop
fetch c_emp into rec_emp;
--exit when c_emp%notfound;
dbms_output.put_line(rec_emp.ename);
--end loop;
close c_emp;
end
/
```

Trigger

Trigger

- A trigger is a pl/sql block structure which is fired when a DML statements like Insert, Delete, Update is executed on a database table
- A trigger is triggered automatically when an associated DML statement is executed

Further Information:

- http://plsql-tutorial.com/plsql-triggers.htm
- https://www.tutorialspoint.com/plsql/plsql_triggers.htm

Trigger Example


```
CREATE OR REPLACE TRIGGER salgrade_added
after INSERT ON salgrade
FOR EACH ROW
BEGIN
  dbms_output.put_line('New Salgrade Added');
END;
/
select * from salgrade;
insert into salgrade values ('6','1234','9999');
rollback
```

Package

Package

- Packages are schema objects that groups logically related PL/SQL types, variables, and subprograms
- A package will have two mandatory parts –
  - Package specification
  - Package body or definition

Further Information:

- https://www.tutorialspoint.com/plsql/plsql_packages.htm

Package Example


```
CREATE PACKAGE emp_pack AS
  PROCEDURE display_ename(e__id
  emp.empno%type);
END emp_pack;
/
```

Package Example


```
CREATE OR REPLACE PACKAGE BODY emp_pack AS

  PROCEDURE display_ename(e__id emp.empno%TYPE) IS
  e_nam emp.ename%TYPE;
  BEGIN
    SELECT ename INTO e_nam
    FROM emp
    WHERE empno = e__id;
    dbms_output.put_line('Employee Name: '|| e_nam);
  END display_ename;
END emp_pack;
/
```

Package Example


```
begin
emp_pack.display_ename('7369');
end

select * from emp;
```

THANK YOU

Supplementary


```
create table author(a_id number(10)primary key, a_name varchar2(20));
create table book(b_id number(10),b_name varchar2(20), isbn varchar2(20),edition
varchar2(20), c_id number(10),a_id number(10), primary key(b_id,edition) );
create table category(c_id number(10)primary key, c_name varchar2(20));
alter table book add constraint fk_category foreign key (c_id) references category(c_id);
alter table book add constraint fk_author foreign key (a_id) references author(a_id);

insert into author values('1','J.K. Rowling');
insert into author values('2','Stephenie Meyer');
insert into author values('3','Dan Brown');
insert into author values('4','Humayun Ahmed');
insert into author values('5','Zafar Iqbal');

insert into category values('11','Fantasy');
insert into category values('22','Romance');
insert into category values('33','Thriller');
insert into category values('44','Anti-logic');
insert into category values('55','Science Fiction');

insert into book values('111','HP...Deathly Hallows','978-3-16-148410-0','10','11','1');
insert into book values('222','Breaking Dawn','979-3-16-148410-0','10','22','2');
insert into book values('333','Origin','980-3-16-148410-0','10','33','3');
insert into book values('444','Holud HimuKalo RAB','981-3-16-148410-0','10','44','4');
insert into book values('555','Obonil','982-3-16-148410-0','10','55','5');
```

Advance Database Management System

Relational Algebra Tutorial

Learning Objectives

To know about:

- Relational Algebra
- Example Queries

Relational Algebra

- Created by Edgar F. Codd
- Defined as a family of algebras with a well-founded semantics used for modeling the data stored in relational databases, and defining queries on it
- Main application is to provide a theoretical foundation for relational databases
- Relational database systems are expected to be equipped with a query language that can assist its users to query the database instances
- There are two kinds of query languages –

    1. Relational algebra and

    2. Relational calculus

.

Relational Algebra

- Procedural query language
- Takes instances of relations as input and yields instances of relations as output
- Uses operators to perform queries
- Operator can be either **unary** or **binary**
- Accept relations as their input and yield relations as their output
- Relational algebra is performed recursively on a relation and intermediate results are also considered relations

Relational Algebra

- Procedural language
- Six basic operators

  1. select: σ

  2. project: Π

  3. union: $\cup$

  4. set difference: –

  5. Cartesian product: x

  6. rename: ρ

- The operators take one or two relations as inputs and produce a new relation as a result.

Select Operation – Example

- Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

- $\sigma_{A=B \wedge D > 5}(r)$

A	B	C	D
α	α	1	7
β	β	23	10

Select Operation

- Notation: $\sigma_p(r)$
- p is called the **selection predicate**
- Defined as:

$$\sigma_p(\mathbf{r}) = \{t \mid t \in r \textbf{ and } p(t)\}$$

Where p is a formula in propositional calculus consisting of **terms**
connected by : $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)
Each **term** is one of:

<attribute> *op* <attribute> or <constant>

where *op* is one of: $=, \neq, >, \geq. <. \leq$

- Example of selection:

$$\sigma_{branch\_name=\text{“}Perryridge\text{”}}(account)$$

Project Operation – Example

- Relation *r*:

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

$\prod_{A,C}(r)$

A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1
β	2

Project Operation

- Notation: $\prod_{A_1, A_2, \ldots, A_k} (r)$

  where A_1, A_2 are attribute names and r is a relation name.
- The result is defined as the relation of k columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets
- Example: To eliminate the *branch_name* attribute of *account*

$$\prod_{account\_number,\ balance} (account)$$

Composition of Relational Operations

- Find the customer who live in Harrison
  - $\Pi_{customer\_name}(\sigma_{customer\_city="Harrison"}(customer))$

- Notice that instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation

Union Operation – Example

- Relations *r, s:*

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r \cup s$:

A	B
α	1
α	2
β	1
β	3

Union Operation

- Notation: $r \cup s$
- Defined as:

$$r \cup s = \{t \mid t \in r \text{ or } t \in s\}$$

- For $r \cup s$ to be valid.
  1. *r, s* must have the *same* **arity** (same number of attributes)
  2. The attribute domains must be **compatible** (example: 2nd column of *r* deals with the same type of values as does the 2nd column of *s*)
- Example: to find all customers with either an account or a loan

$$\Pi_{customer\_name}\,(depositor) \;\cup\; \Pi_{customer\_name}\,(borrower)$$

Set Difference Operation – Example

- Relations *r*, *s*:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- *r* – *s*:

A	B
α	1
β	1

Set Difference Operation

- Notation $r - s$
- Defined as:

$$r - s = \{t \mid t \in r \textbf{ and } t \notin s\}$$

- Set differences must be taken between **compatible** relations.
  - r and s must have the same arity
  - attribute domains of r and s must be compatible

Cartesian-Product Operation – Example

- Relations *r*, *s*:

A	*B*
α	1
β	2

r

C	*D*	*E*
α	10	a
β	10	a
β	20	b
γ	10	b

s

- *r* x *s*:

A	*B*	*C*	*D*	*E*
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Cartesian-Product Operation

- Notation *r* x *s*
- Defined as:

$$r \text{ x } s = \{t\, q \mid t \in r \textbf{ and } q \in s\}$$

- Assume that attributes of r(R) and s(S) are disjoint. (That is, $R \cap S = \varnothing$).
- If attributes of *r(R)* and *s(S*) are not disjoint, then renaming must be used.

Composition of Operations

- Can build expressions using multiple operations, Example: $\sigma_{A=C}(r \times s)$
- $r \times s$

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

- $\sigma_{A=C}(r \times s)$

A	B	C	D	E
α	1	α	10	a
β	2	β	10	a
β	2	β	20	b

Rename Operation

- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.
- Example:

$$\rho_X(E)$$

returns the expression E under the name X

- If a relational-algebra expression E has arity n, then

$$\rho_{X(A_1, A_2, \ldots, A_n)}(E)$$

returns the result of expression E under the name X, and with the attributes renamed to $A_1, A_2, \ldots, A_n$.

Banking Example

branch (branch_name, branch_city, assets)

customer (customer_name, customer_street, customer_city)

account (account_number, branch_name, balance)

loan (loan_number, branch_name, amount)

depositor (customer_name, account_number)

borrower (customer_name, loan_number)

Example Queries

- Find all loans of over \$1200

$$\sigma_{amount\ >\ 1200}\ (loan)$$

- Find the loan number for each loan of an amount greater than \$1200

$$\Pi_{loan\_number}\ (\sigma_{amount\ >\ 1200}\ (loan))$$

- Find the names of all customers who have a loan, an account, or both, from the bank

$$\Pi_{customer\_name}\ (borrower) \cup \Pi_{customer\_name}\ (depositor)$$

Example Queries

- Find the names of all customers who have a loan at the Perryridge branch.

$$\Pi_{customer\_name}\ (\sigma_{branch\_name=\text{"Perryridge"}}\ (\sigma_{borrower.loan\_number\ =\ loan.loan\_number}(borrower\ x\ loan)))$$

- Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank.

$$\Pi_{customer\_name}\ (\sigma_{branch\_name\ =\ \text{"Perryridge"}}\ (\sigma_{borrower.loan\_number\ =\ loan.loan\_number}(\text{borrower x loan})))\ -\ \Pi_{customer\_name}(\text{depositor})$$

Example Queries

- Find the names of all customers who have a loan at the Perryridge branch.
  - Query 1

$$\Pi_{customer\_name} (\sigma_{branch\_name = \text{"Perryridge"}} (\sigma_{borrower.loan\_number = loan.loan\_number} (borrower \times loan)))$$

  - Query 2

$$\Pi_{customer\_name}(\sigma_{loan.loan\_number = borrower.loan\_number} ((\sigma_{branch\_name = \text{"Perryridge"}} (loan)) \times borrower))$$

Example Queries

- Find the largest account balance
  - Strategy:
    - Find those balances that are *not* the largest
      - Rename *account* relation as *d* so that we can compare each account balance with all others
    - Use set difference to find those account balances that were *not* found in the earlier step.
  - The query is:

$$\Pi_{balance}(account) - \Pi_{account.balance}(\sigma_{account.balance < d.balance}(account \times \rho_d(account)))$$

Formal Definition

- A basic expression in the relational algebra consists of either one of the following:
  - A relation in the database
  - A constant relation
- Let E_1 and E_2 be relational-algebra expressions; the following are all relational-algebra expressions:
  - $E_1 \cup E_2$
  - $E_1 - E_2$
  - $E_1 \times E_2$
  - $\sigma_p (E_1)$, P is a predicate on attributes in E_1
  - $\prod_S(E_1)$, S is a list consisting of some of the attributes in E_1
  - $\rho_x (E_1)$, x is the new name for the result of E_1

Additional Operations

We define additional operations that do not add any power to the relational algebra, but that simplify common queries.

- Set intersection
- Natural join
- Division
- Assignment

Set-Intersection Operation

- Notation: $r \cap s$
- Defined as:
- $r \cap s = \{ t \mid t \in r \textbf{ and } t \in s \}$
- Assume:
  - r, s have the *same arity*
  - attributes of r and s are compatible
- Note: $r \cap s = r - (r - s)$

Set-Intersection Operation – Example

- Relation *r, s*:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r \cap s$

A	B
α	2

Natural-Join Operation

- Notation: $r \bowtie s$

- Let r and s be relations on schemas R and S respectively. Then, $r \bowtie s$ is a relation on schema $R \cup S$ obtained as follows:
  - Consider each pair of tuples t_r from r and t_s from s.
  - If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result, where
    - t has the same value as t_r on r
    - t has the same value as t_s on s
- Example:

  $R = (A, B, C, D)$

  $S = (E, B, D)$
  - Result schema = (A, B, C, D, E)
  - $r \bowtie s$ is defined as:

$$\Pi_{r.A,\, r.B,\, r.C,\, r.D,\, s.E}\left(\sigma_{r.B = s.B \wedge r.D = s.D}\left(r \times s\right)\right)$$

Natural Join Operation – Example

- Relations r, s:

A	*B*	*C*	*D*
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

r

B	*D*	*E*
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	$\in$

s

r $\bowtie$ s

A	*B*	*C*	*D*	*E*
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

Division Operation

- Notation: $r \div s$
- Suited to queries that include the phrase "for all".
- Let *r* and *s* be relations on schemas *R* and *S* respectively where
  - $R = (A_1, \ldots, A_m, B_1, \ldots, B_n)$
  - $S = (B_1, \ldots, B_n)$

  The result of $r \div s$ is a relation on schema

  $R - S = (A_1, \ldots, A_m)$

  $$r \div s = \{ t \mid t \in \prod_{R\text{-}S}(r) \wedge \forall u \in s \, (tu \in r) \}$$

  Where *tu* means the concatenation of tuples *t* and *u* to produce a single tuple

Examples of Division A/B

sno	pno
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

A

pno
P2

B1

pno
P2
P4

B2

pno
P1
P2
P4

B3

sno
S1
S2
S3
S4

A/B1

sno
S1
S4

A/B2

sno
S1

A/B3

Bank Example Queries

- Find the names of all customers who have a loan and an account at bank.

$$\Pi_{customer\_name}\ (borrower) \cap \Pi_{customer\_name}\ (depositor)$$

- Find the name of all customers,loan_no, loan amount who have a loan at the bank

$$\Pi_{customer\_name,\ loan\_number,\ amount}\ (borrower \bowtie loan)$$

Bank Example Queries

- Find all customers who have an account from at least the "Downtown" and the Uptown" branches.

$$\Pi_{customer\_name}\,(\sigma_{branch\_name\,=\,\text{“Downtown”}}\,(depositor \bowtie account)) \cap$$

$$\Pi_{customer\_name}\,(\sigma_{branch\_name\,=\,\text{“Uptown”}}\,(depositor \bowtie account))$$

Extended Relational-Algebra-Operations

- Generalized Projection
- Aggregate Functions
- Outer Join

Generalized Projection

- Extends the projection operation by allowing arithmetic functions to be used in the projection list.

$$\prod_{F_1, F_2, \ldots, F_n}(E)$$

- E is any relational-algebra expression
- Each of $F_1, F_2, \ldots, F_n$ are are arithmetic expressions involving constants and attributes in the schema of E.
- Given relation *credit_info(customer_name, limit, credit_balance),* find how much more each person can spend:

$$\prod_{customer\_name,\ limit\ -\ credit\_balance}(credit\_info)$$

Aggregate Functions and Operations

- **Aggregation function** takes a collection of values and returns a single value as a result.

  avg: average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values

- **Aggregate operation** in relational algebra

$$_{G_1, G_2, \ldots, G_n}\vartheta_{F_1(A_1), F_2(A_2), \ldots, F_n(A_n)}(E)$$

  E is any relational-algebra expression
  - $G_1, G_2, \ldots, G_n$ is a list of attributes on which to group (can be empty)
  - Each F_i is an aggregate function
  - Each A_i is an attribute name

Aggregate Operation – Example

- Relation *r*:

A	*B*	*C*
α	α	7
α	β	7
β	β	3
β	β	10

- $\mathcal{g}_{\mathbf{sum(c)}}(\mathrm{r})$

sum(*c*)
27

Aggregate Operation – Example

- Relation *account* grouped by *branch-name*:

branch_name	*account_number*	*balance*
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

$$\textit{branch\_name}\ \mathcal{G}\ _{\textbf{sum}(\textit{balance})}\ (\textit{account})$$

branch_name	**sum**(*balance*)
Perryridge	1300
Brighton	1500
Redwood	700

Aggregate Functions (Cont.)

- Result of aggregation does not have a name
  - Can use rename operation to give it a name
  - For convenience, we permit renaming as part of aggregate operation

$$_{branch\_name}\ \mathcal{G}\ _{\mathbf{sum}(balance)\ \mathbf{as}\ sum\_balance}\ (account)$$

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values:
  - *null* signifies that the value is unknown or does not exist
  - All comparisons involving *null* are (roughly speaking) **false** by definition.
    - We shall study precise meaning of comparisons with nulls later

Outer Join – Example

- Relation *loan*

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

- Relation *borrower*

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

Outer Join – Example

- Join

loan ⋈ *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

- Left Outer Join

loan ⟕ *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*

Outer Join – Example

- Right Outer Join

⟖ *loan* *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

- Full Outer Join

⟗ *loan* *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*.
- Aggregate functions simply ignore null values (as in SQL)
- For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same (as in SQL)

Null Values

- Comparisons with null values return the special truth value: *unknown*
  - If *false* was used instead of *unknown*, then *not (A < 5)* would not be equivalent to *A >= 5*
- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*) = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - AND: (*true* **and** *unknown*) = *unknown*,
    (*false* **and** *unknown*) = *false*,
    (*unknown* **and** *unknown*) = *unknown*
  - NOT: (**not** *unknown*) = *unknown*
  - In SQL "*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*
- Result of select predicate is treated as *false* if it evaluates to *unknown*

Modification of the Database

- The content of the database may be modified using the following operations:
  - Deletion
  - Insertion
  - Updating
- All these operations are expressed using the assignment operator.

Deletion

- A delete request is expressed similarly to a query, except instead of displaying tuples to the user, the selected tuples are removed from the database.
- Can delete only whole tuples; cannot delete values on only particular attributes
- A deletion is expressed in relational algebra by:

$$r \leftarrow r - E$$

where r is a relation and E is a relational algebra query.

Deletion Examples

- Delete all account records in the Perryridge branch.

$$account \leftarrow account - \sigma_{branch\_name = \text{“Perryridge”}} (account)$$

- Delete all loan records with amount in the range of 0 to 50

$$loan \leftarrow loan - \sigma_{amount \geq 0 \text{ and } amount \leq 50} (loan)$$

- Delete all accounts at branches located in Needham.

$$r_1 \leftarrow \sigma_{branch\_city = \text{“Needham”}} (account \bowtie branch)$$

$$r_2 \leftarrow \Pi_{account\_number,\ branch\_name,\ balance} (r_1)$$

$$r_3 \leftarrow \Pi_{customer\_name,\ account\_number} (r_2 \bowtie depositor)$$

$$account \leftarrow account - r_2$$

$$depositor \leftarrow depositor - r_3$$

Insertion

- To insert data into a relation, we either:
  - specify a tuple to be inserted
  - write a query whose result is a set of tuples to be inserted
- in relational algebra, an insertion is expressed by:

$$r \leftarrow r \cup E$$

  where r is a relation and E is a relational algebra expression.
- The insertion of a single tuple is expressed by letting E be a constant relation containing one tuple.

Insertion Examples

- Insert information in the database specifying that Smith has $1200 in account A-973 at the Perryridge branch.

$$account \leftarrow account \cup \{(\text{“A-973”}, \text{“Perryridge”}, 1200)\}$$

$$\text{depositor} \leftarrow depositor \cup \{(\text{“Smith”}, \text{“A-973”})\}$$

- Provide as a gift for all loan customers in the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account.

$$r_1 \leftarrow (\sigma_{branch\_name = \text{“Perryridge”}}(borrower \bowtie \text{loan}))$$

$$account \leftarrow account \cup \prod_{loan\_number,\ branch\_name,\ 200} (r_1)$$

$$\text{depositor} \leftarrow depositor \cup \prod_{customer\_name,\ loan\_number} (r_1)$$

Updating

- A mechanism to change a value in a tuple without charging *all* values in the tuple
- Use the generalized projection operator to do this task

$$r \leftarrow \prod_{F_1, F_2, \ldots, F_l,}(r)$$

- Each F_i is either
  - the I^{th} attribute of r, if the I^{th} attribute is not updated, or,
  - if the attribute is to be updated F_i is an expression, involving only constants and the attributes of r, which gives the new value for the attribute

Update Examples

- Make interest payments by increasing all balances by 5 percent.

$$account \leftarrow \Pi_{account\_number,\ branch\_name,\ balance\ *\ 1.05}\ (account)$$

- Pay all accounts with balances over $10,000 6 percent interest and pay all others 5 percent

$$account \leftarrow \Pi_{account\_number,\ branch\_name,\ balance\ *\ 1.06}\ (\sigma_{BAL\ >\ 10000}\ (account\ )) \cup \Pi_{account\_number,\ branch\_name,\ balance\ *\ 1.05}\ (\sigma_{BAL\ \leq\ 10000}\ (account))$$

THANK YOU

Advance Database Management System
UML Diagrams Tutorial

Learning Objectives

To know about:

- Class Diagram
- Use Case Diagram
- Activity Diagram

Class Diagram

What is Class Diagram?

- The Class diagram represents classes, their component parts, and the way in which classes of objects are related to one another.
- The Class diagram includes attributes, operations, stereotypes, properties, associations, and inheritance.

Components of Class Diagram

- **Attributes** describe the appearance and knowledge of a class of objects.
- **Operations** define the behavior that a class of objects can manifest.
- **Stereotypes** help you understand this type of object in the context of other classes of objects with similar roles within the system's design.
- **Properties** provide a way to track the maintenance and status of the class definition.
- **Association** is just a formal term for a type of relationship that this type of object may participate in. Associations may come in many variations, including simple, aggregate and composite, qualified, and reflexive.
- **Inheritance** allows you to organize the class definitions to simplify and facilitate their implementation.

Why Class Diagram is necessary?

- Although other diagrams are necessary, remember that their primary purpose is to support the construction and testing of the Class diagram.

- Whenever another diagram reveals new or modified information about a class, the Class diagram must be updated to include the new information. If this new information is not passed on to the Class diagram, it will not be reflected in your code.

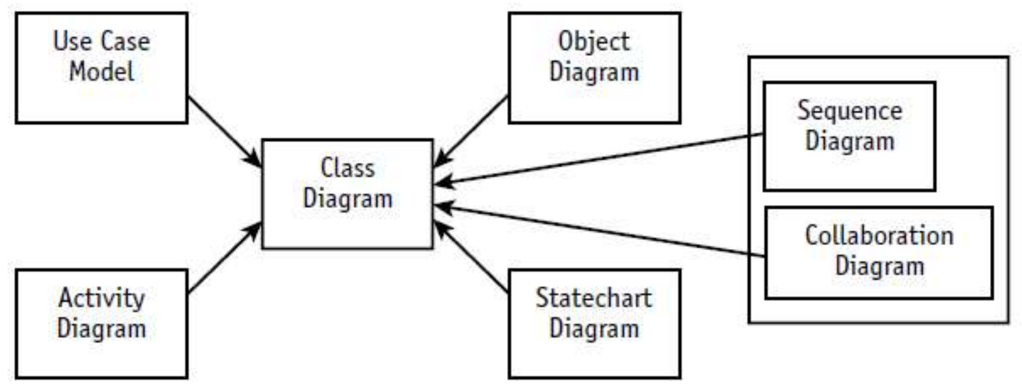


Figure 9-1 All diagrams support the Class diagram.

The Class Symbol

- The class symbol is comprised of three compartments (rectangular spaces) that contain distinct information needed to describe the properties of a single type of object.
  - The name compartment uniquely defines a class (a type of object) within a package. Consequently, classes may have the same name if they reside in different packages.
  - The attribute compartment contains all the data definitions.
  - The operations compartment contains a definition for each behavior supported by this type of object.

Attribute

- An attribute describes a piece of information that an object owns or knows about itself.
- Attribute visibility:
  - **Public (+)** visibility allows access to objects of all other classes.
  - **Private (-)** visibility limits access to within the class itself. For example, only operations of the class have access to a private attribute.
  - **Protected (#)** visibility allows access by subclasses. In the case of generalizations (inheritance), subclasses must have access to the attributes and operations of the superclass or they cannot be inherited.
  - **Package (~)** visibility allows access to other objects in the same package.

Attribute

visibility / attribute name : data type = default value {constraints}

- **Visibility (+, -, #, ~):** *Required before code generation*. The programming language will typically specify the valid options. The minus sign represents the visibility "private" meaning only members of the class that defines the attribute may see the attribute.

- **Slash (/):** The derived attribute indicator is *optional*. Derived values may be computed or figured out using other data and a set of rules or formulas. Consequently, there are more design decisions that need to be addressed regarding the handling of this data. Often this flag is used as a placeholder until the design decisions resolve the handling of the data.

- **Attribute name:** *Required*. Must be unique within the class.

- **Data type:** *Required*. This is a big subject. During analysis, the data type should reflect how the client sees the data. You could think of this as the external view. During design, the data type will need to represent the programming language data type for the environment in which the class will be coded. These two pieces of information can give the programmer some very specific insights for the coding of get and set methods to support access to the attribute value.

- **Assignment operator and default value:** *Optional*. Default values serve two valuable purposes. First, default values can provide significant ease-of-use improvements for the client. Second and more importantly, they protect the integrity of the system from being corrupted by missing or invalid values. A common example is the tendency to let numeric attributes default to zero. If the application ever attempts to divide using this value, you will have to handle resulting errors that could have been avoided easily with the use of a default.

- **Constraints:** Constraints express all the rules required to guarantee the integrity of this piece of information. Any time another object tries to alter the attribute value, it must pass the rules established in the constraints. The constraints are typically implemented/enforced in any method that attempts to set the attribute value.

- **Class level attribute (underlined attribute declaration):** *Optional*. Denotes that all objects of the class share a single value for the attribute. (This is called a *static* value in Java.)

Operation

- Objects have behaviors, things they can do and things that can be done to them. These behaviors are modeled as operations.

- Operations require a name, arguments, and sometimes a return.

- Arguments, or input parameters, are simply attributes, so they are specified using the attribute notation (name, data type, constraints, and default), although it is very common to use the abbreviated form of name and data type only.

Operation

visibility operationName (argname : data type {constraints}, …) :return data type {constraints}

- **Visibility (+, -, #, ~):** *Required before code generation*. The visibility values are defined by the programming language, but typically include public (+), private (-), protected (#), and package (~).
- **Operation name:** *Required*. Does not have to be unique, but the combination of name and parameters does need to be unique within a class.
- **Arguments/parameters:** Any number of arguments is allowed. Each argument requires an identifier and a data type. Constraints may be used to define the valid set of values that may be passed in the argument. But constraints are not supported in many tools and will not be reflected in the code for the operation, at least not at this point.
- **Argument name:** *Required for each parameter, but parameters are optional.* Any number of arguments is allowed.
- **Argument data type:** *Required for each parameter, but parameters are optional.*
- **Constraints: Optional. In general, constraints express rules that must be enforced** in the execution of the operation. In the case of parameters, they express criteria that the values must satisfy before they may be used by the operation. You can think of them as operation level pre-conditions.
- **Return data type:** *Required for a return value, but return values are optional*. The UML only allows for the type, not the name, which is consistent with most programming languages. There may only be one return data type, which again is consistent with most programming languages.
- **Class level operation (underlined operation declaration):** *Optional.* Denoted as an operation accessible at the class level; requires an instance (object) reference.

Class Notation

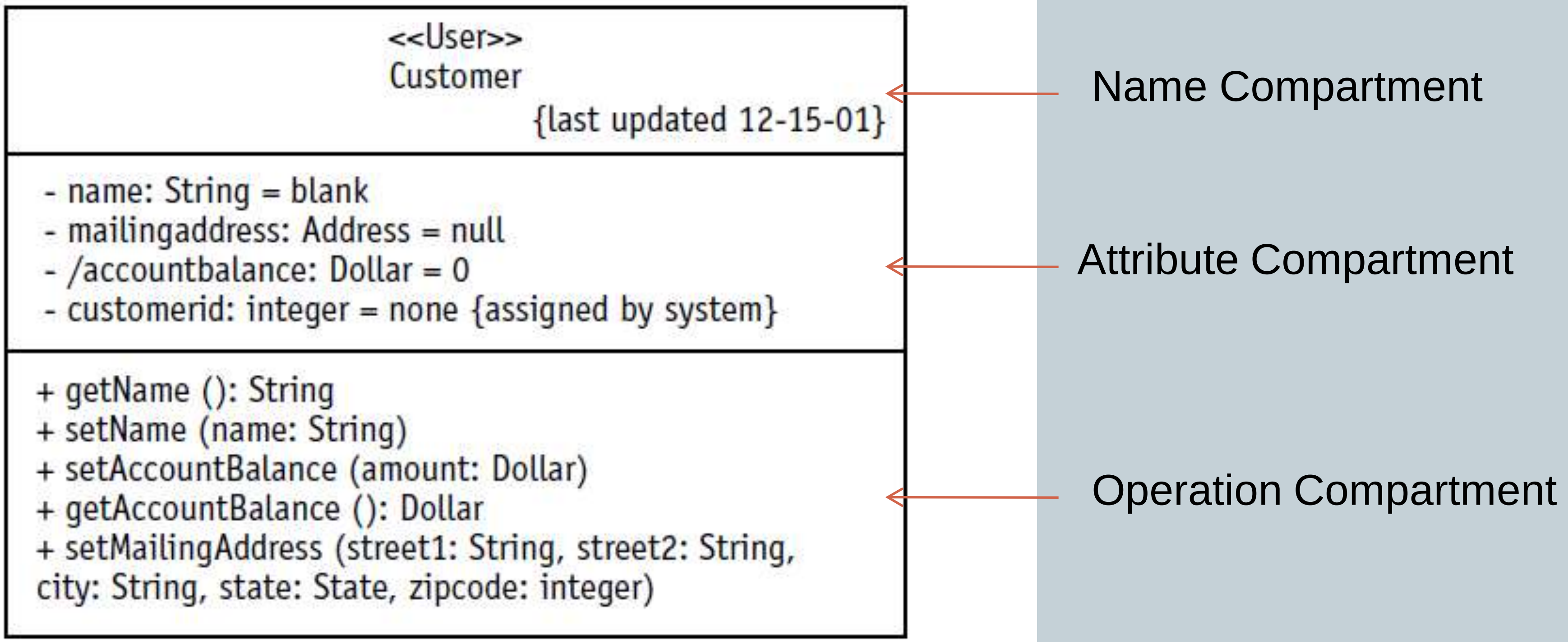
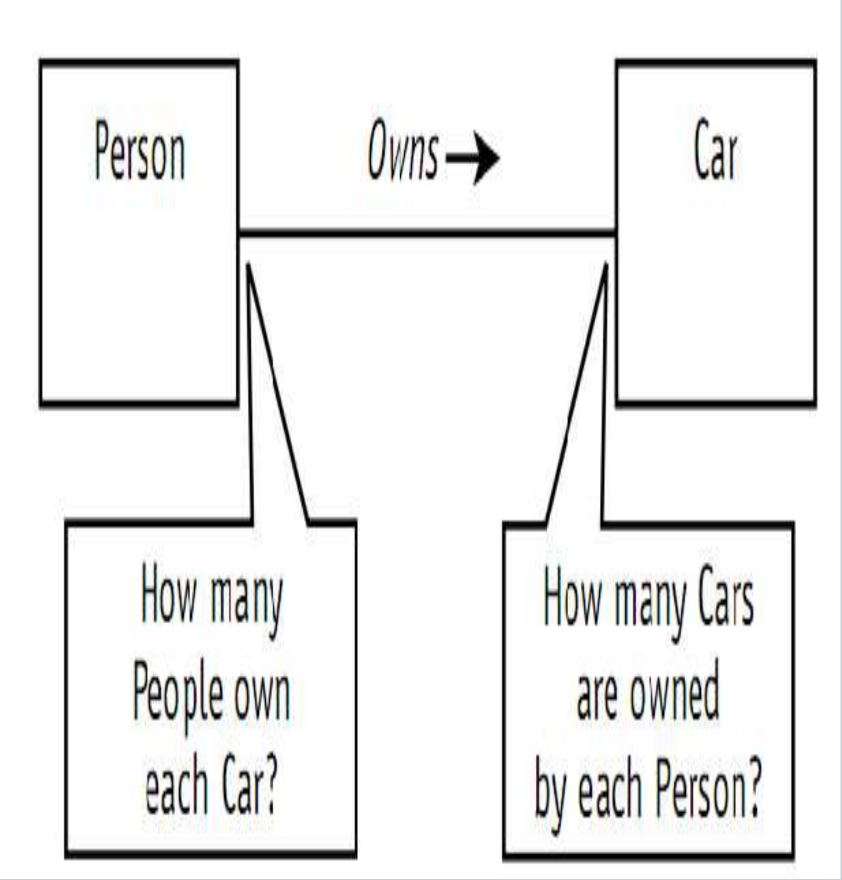


Figure 9-2 *Complete class specification with all three compartments*

Association multiplicity

- The UML allows you to handle some other important questions about associations: “How many Cars may a Person own?” “How many can they rent?” “How many people can drive a given Car?”
- Associations define the rules for how objects in each class may be related. So how do you specify exactly how many objects may participate in the relationship?



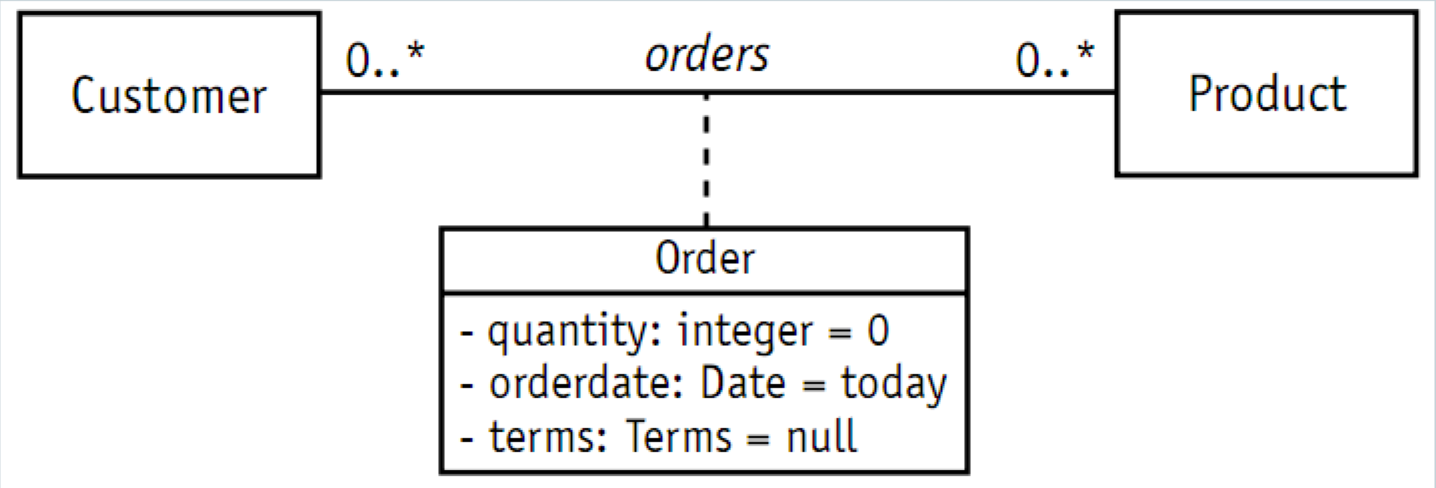
Association multiplicity

Summary list of the options for specifying multiplicity followed by some examples.

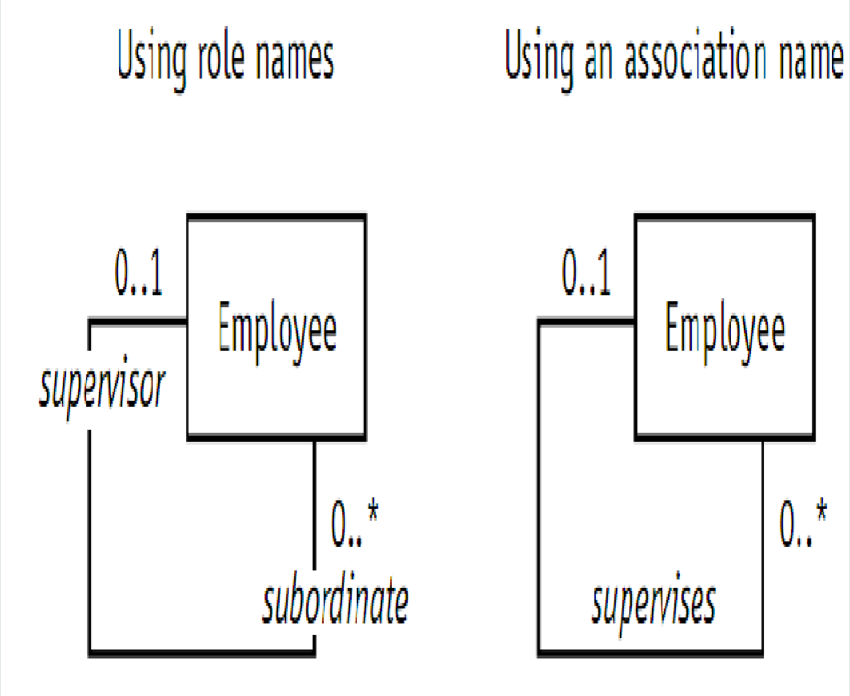
- Values separated by two periods (..) mean a range. For example, 1..3 means between 1 and 3 inclusively; 5..10 means between 5 and 10 inclusively.
- Values separated by commas mean an enumerated list of possibilities. For example, 4,6,8 means you may have 4 objects or 6 objects or 8 objects of this type in the association.
- Asterisk (*) when used alone means zero or more, no lower or upper limit.
- Asterisk (*) when used in a range (1..*) means no upper limit—you must have at least one but you can have as many more as you want.

Association Class

- Encapsulates information about an association.

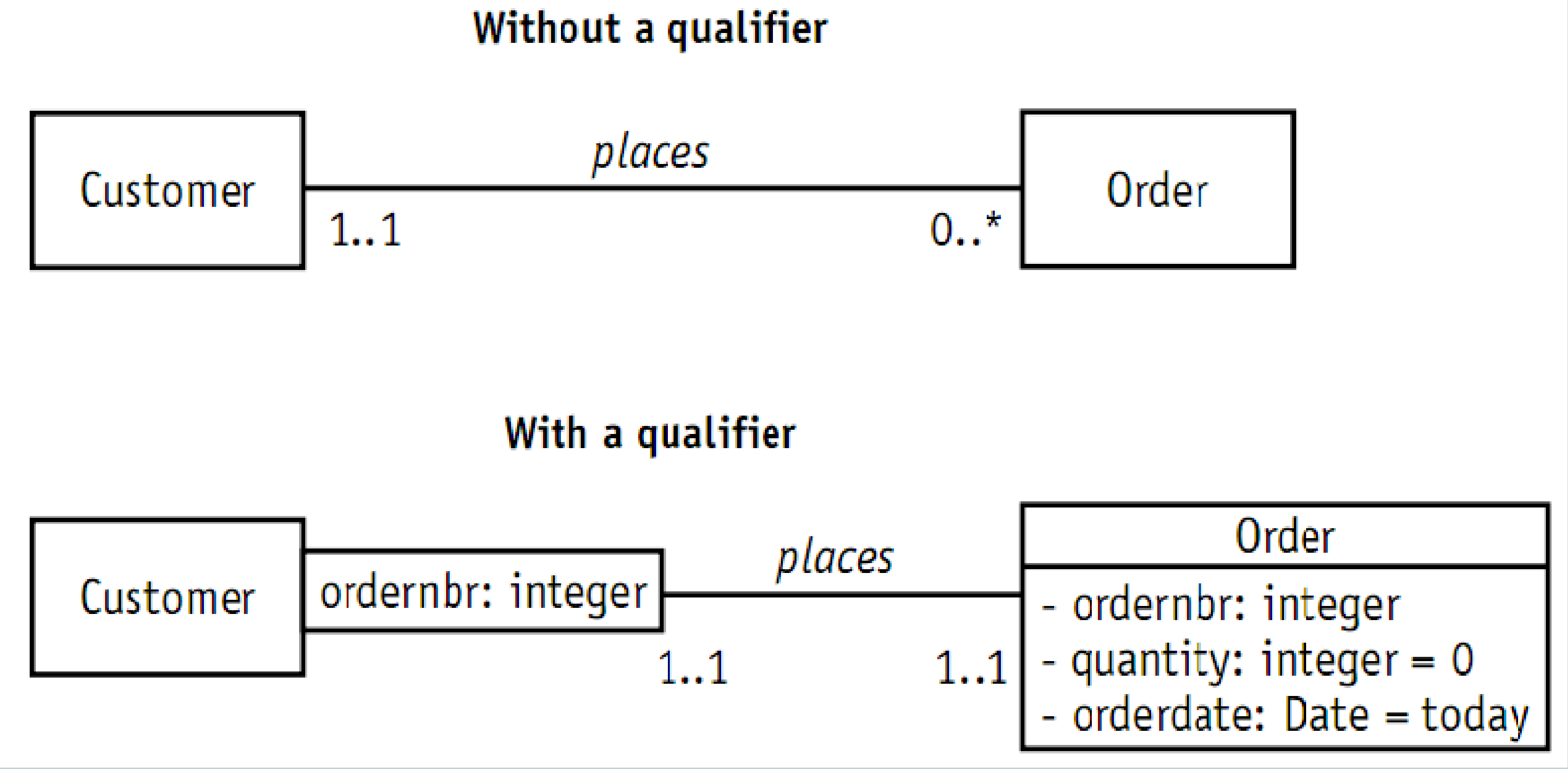


Reflexive Association



- Reflexive association is a fancy expression that says objects in the same class can be related to one another. The entire association notation you've learned so far remains exactly the same, except that both ends of the association line point to the same class.

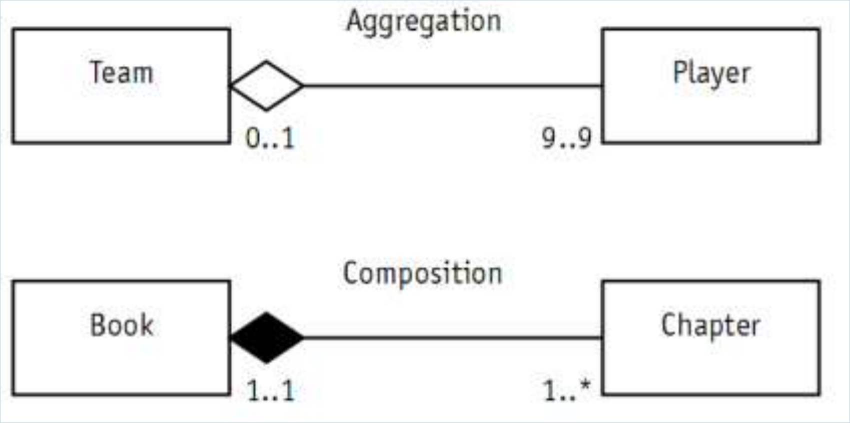
Qualified Association



Aggregation and Composition

- **Aggregation** is a special type of association used to indicate that the participating objects are not just independent objects that know about each other. Instead, they are assembled or configured together to create a new, more complex object.
  - For example, a number of different parts are assembled to create a car, a boat, or a plane.
- **Composition** is used for aggregations where the life span of the part depends on the life span of the aggregate.
- The aggregate has control over the creation and destruction of the part. In other words, the member object cannot exist apart from the aggregation.

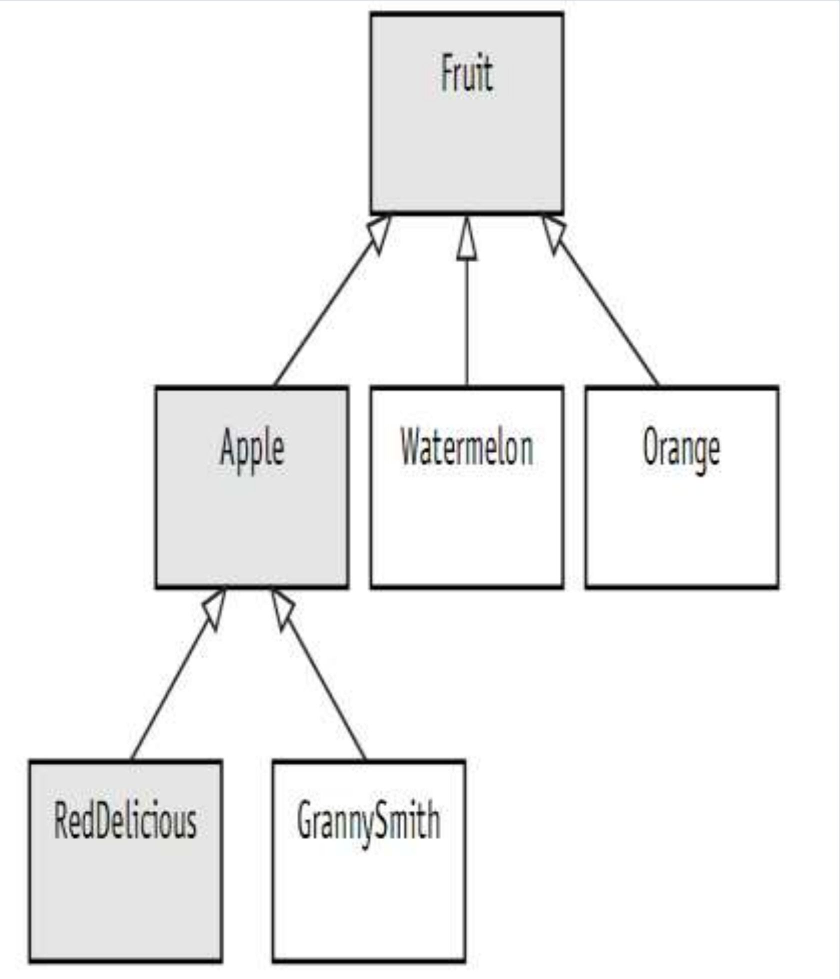
Aggregation and Composition



- Players are assembled into a team. But if the Team is disbanded, the players live on (depending of course on how well they performed).
- The Book example uses composition, the solid diamond. A Book is composed of Chapters. The Chapters would not continue to exist elsewhere on their own. They would cease to exist along with the Book.

Generalization

- Generalization is the process of organizing the properties of a set of objects that share the same purpose.
- A generalization is not an association.



Case Study: Class Diagram

Student may attend any number of courses
Every course may have any number of students
Instructors teach courses
For every course there is at least one instructor
Every instructor may teach zero or more courses
A school has zero or more students
Each student may be a registered member of one or more school
A school has one or more departments
Each department belongs to exactly one school
Every instructor is assigned to one or more departments
Each department has one or more instructors
For every department there is exactly one instructor acting as the department chair

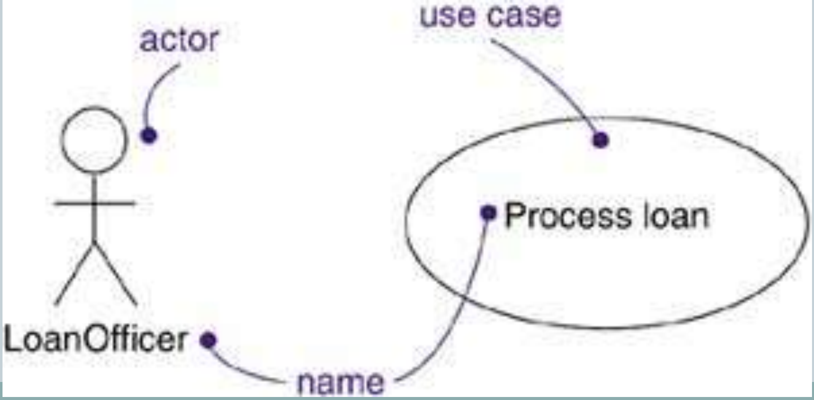
Use Case Diagram

What is Use Case?

- A use case is a description of a set of sequences of actions, including variants, that a subject performs to yield an observable result of value to an actor.
- For example, one central use case of a bank is to process loans for its loan applicants/ clients/ customers.

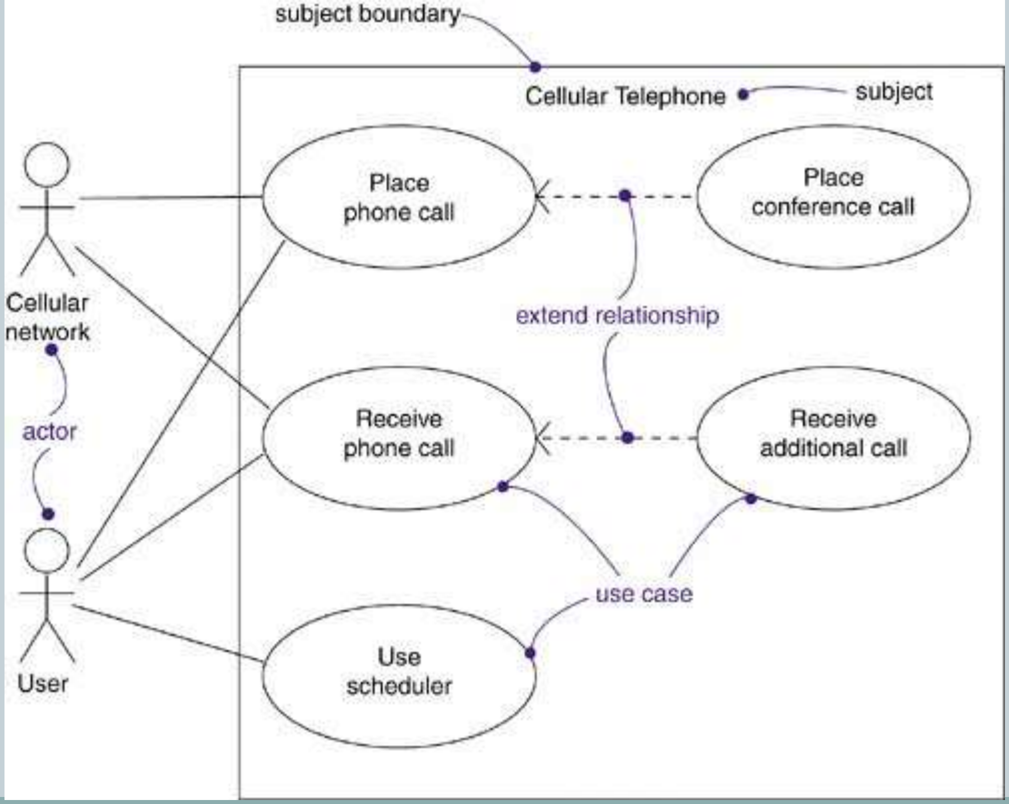
What is Actor?

- An actor represents a coherent set of roles that users of use cases play when interacting with these use cases.
- Actors can be human or they can be automated systems.
- For example, in modeling a bank, processing a loan involves, among other things, the interaction between a customer and a loan officer.



Use Case Diagram

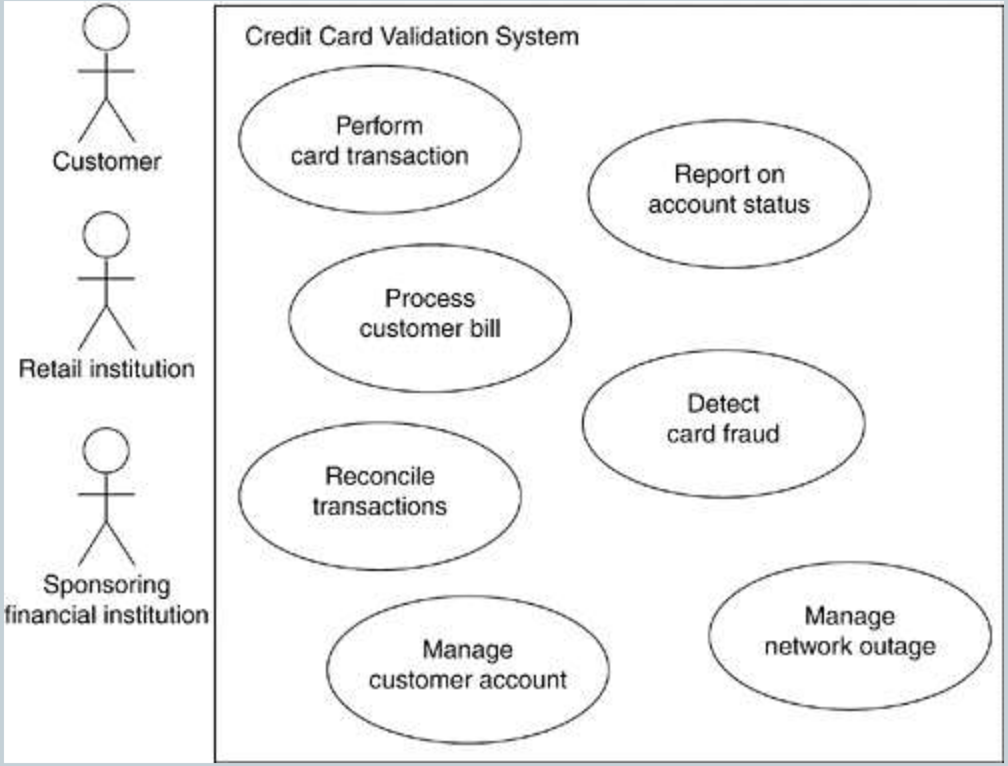
- Apply use case diagrams to visualize the behavior of a system, so that users can comprehend how to use that element, and so that developers can implement that element



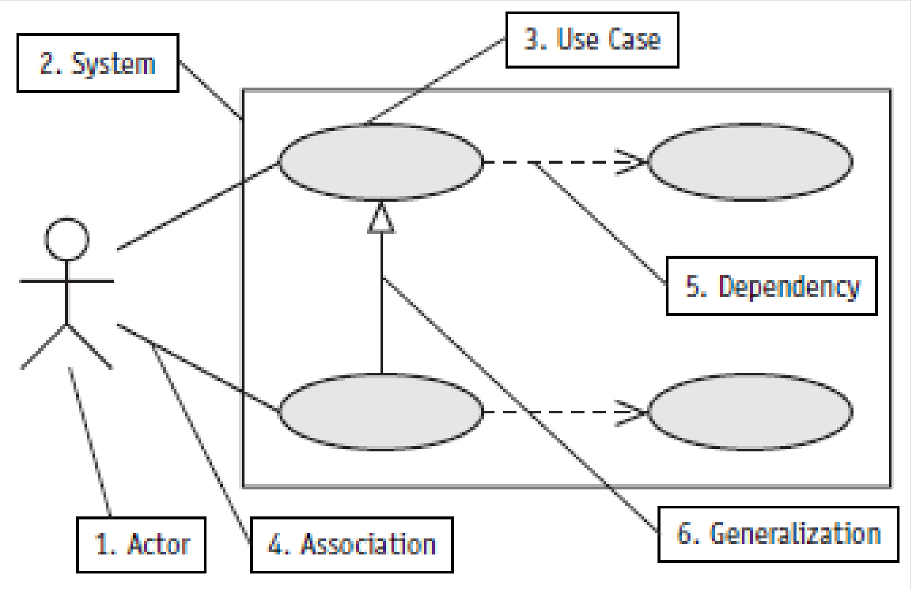
Modeling the Requirements of a System

- Establish the context of the system by identifying the actors that surround it
- For each actor, consider the behavior that each expects or requires the system to provide
- Name these common behaviors as use cases
- Model these use cases, actors, and their relationships in a use case diagram
- Enhance these use cases with notes or constraints

Modeling the Requirements of a System



Elements of Use Case Diagram



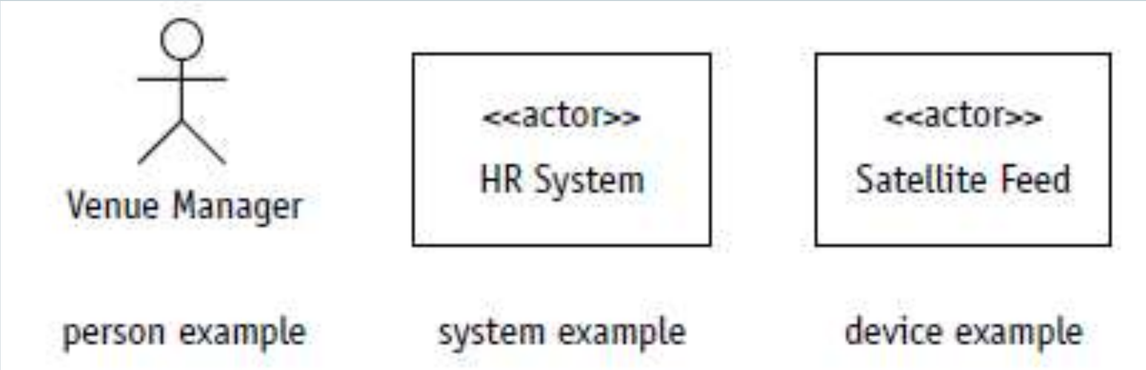
Relationships in Use Case Diagram

- Association
  - Between Actor and Use Case
  - **NOT** between Use Cases
- Generalization
  - Between Actors
  - Between Use Cases
- Dependency
  - Between Use Cases
  - <<include>> and <<extend>>

Actors

- Find the people, systems, or devices that communicate with the system
- The system-type actors are often easiest to spot

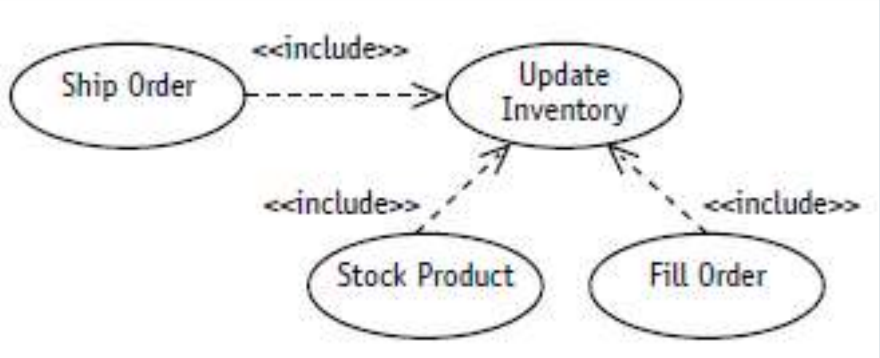
Notation of Actors

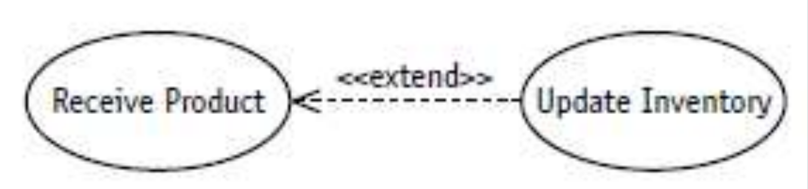


Dependency

- **<<include>>**
  - Mandatory dependency
  - << i >>
  - Arrow towards the use case dependent on
- **<<extend>>**
  - Optional Dependency
  - << e >>
  - Arrow from the use case dependent on
  - Often referred as options of the use case

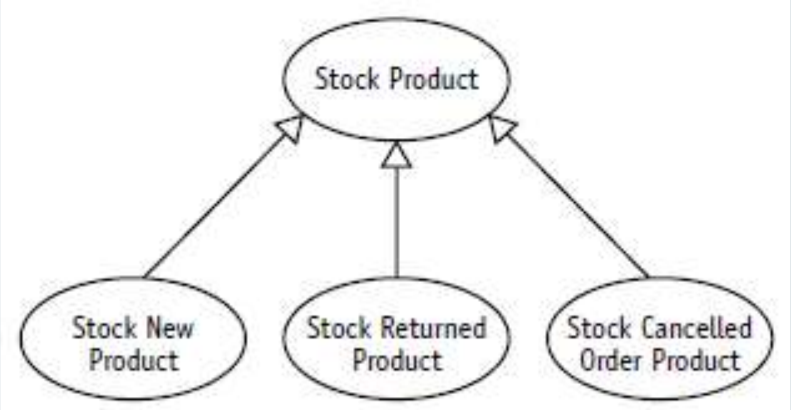
<<include>> & <<extend>>





Generalization

- The child use case inherits the behavior and meaning of the parent use case.
- The child may add to or override the behavior of its parent.



Case Study: Use Case Diagram

In a hotel management system a guest can rent rooms. Hotel receptionist uses the system to assist in the renting. A guest can also book a room for future renting with the help of the receptionist, but he has to check if the room has prior booking or not. A Guest pays for the rooms at the time they check out. He can pay by cash, check or credit card. Receptionists has to logon to the system before they can use it, but to logon he has go through username/password verification.

Activity Diagram

Learning Objectives

To know about:

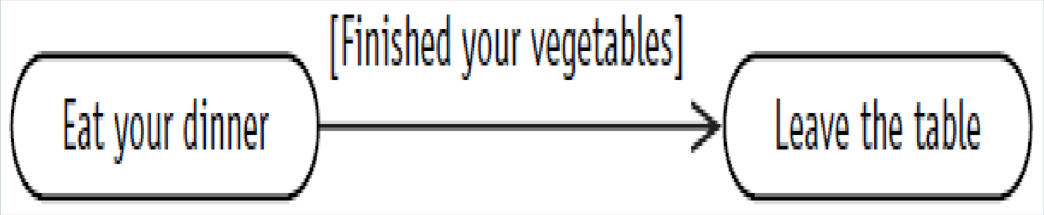
- Activity Diagram
- Components of Activity Diagram

Activity Diagram

- The Activity diagram is the UML version of the classic flowchart. It may be applied to any process, large or small.
- An Activity diagram is a series of activities linked by transitions, arrows connecting each activity. Typically, the transition takes place because the activity is completed. For example, you're currently in the activity "reading a page." When you finish this activity, you switch to the activity "turning page."

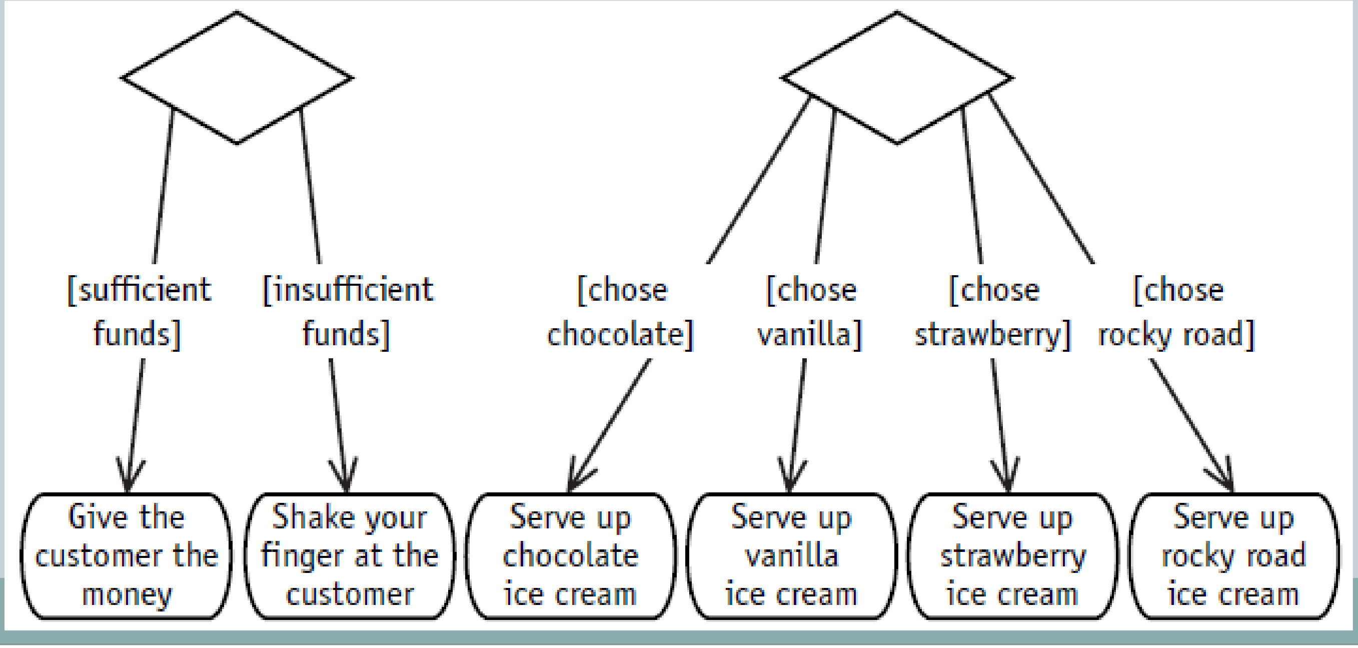
Guard Condition

- Sometimes the transition should only be used when certain things have happened. A guard condition can be assigned to a transition to restrict use of the transition. Place the condition within square brackets somewhere near the transition arrow. The condition must test true before you may follow the associated transition to the next activity.



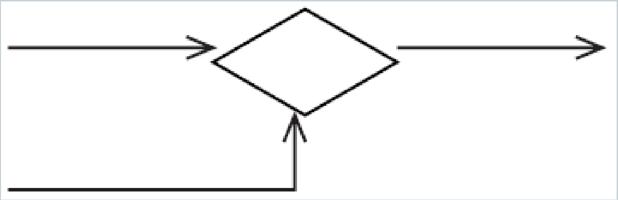
Decision

- The Activity diagram diamond is a decision icon, just as it is in flowcharts. In either diagram, one arrow exits the diamond for each possible value of the tested condition. The decision may be as simple as a true/false test The decision may involve a choice between a set of options.



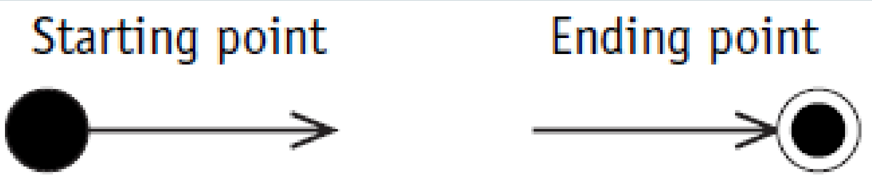
Merge Point

- The diamond icon is also used to model a merge point, the place where two alternative paths come together and continue as one. The two paths in this case are mutually exclusive routes. For example, you and I might each walk from your house to the store. I choose to walk down the left side of the street while you walk down the right. But two blocks before the store we both have to turn right and walk on the right side of that street to the store’s front door.



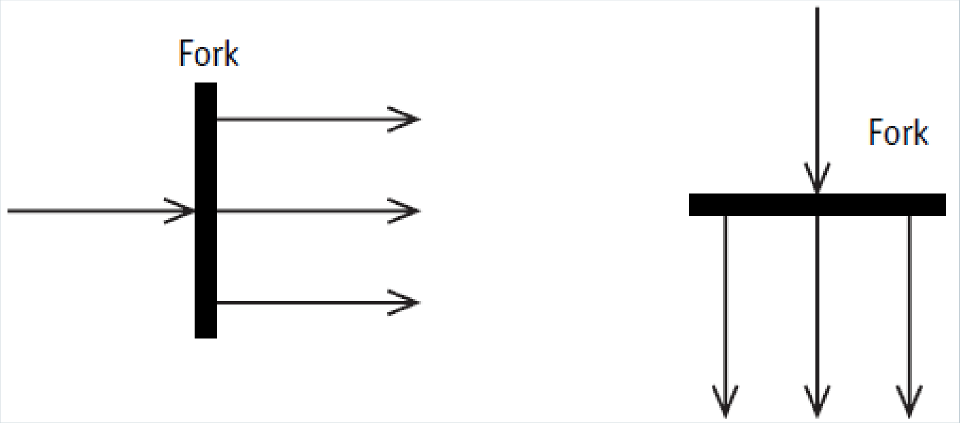
Start and End

- The UML also provides icons to begin and end an Activity diagram. A solid dot indicates the beginning of the flow of activity. A bull's-eye indicates the end point. There may be more than one end point in an Activity diagram. Even the simplest Activity diagram typically has some decision logic that would result in alternative paths, each with its own unique outcome. If you really want to, you can draw all your arrows to the sameend point, but there is no need to do so. Every end point means the same thing: Stop here.



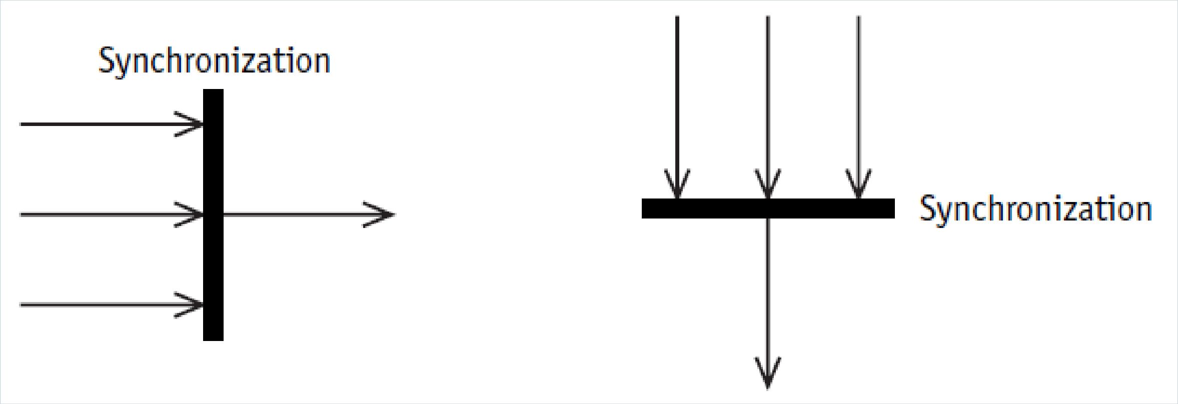
Concurrency

- The UML notation for the Activity diagram also supports concurrency. To show that a single process starts multiple concurrent threads or processes, the UML uses a simple bar called a fork.



Concurrency

- The Synchronization or merging of the concurrent threads or processes is shown in much the same way.



Case Study: Activity Diagram

In a computer shop, customers choose a laptop from the display shelf. The sales person then explains the features to the customer. If the customer does not like it, he leaves the shop and visits the next one. Otherwise the customer enquiries if there are any other color available. The sales person searches the system for available colors. Then the customer chooses the color and confirms the specific laptop that he wants to buy. The laptop is then sent to the service department for software installation and at the same time the bill is generated. While software are being installed, the customer collects a wireless mouse and a bag which comes free with the laptop. When software installation is finished, the customer collects the laptop from service department. Then he pays the bill in the bill paying counter.

THANK YOU

Advance Database Management System
Exception Handling Tutorial

Learning Objectives

To know about:

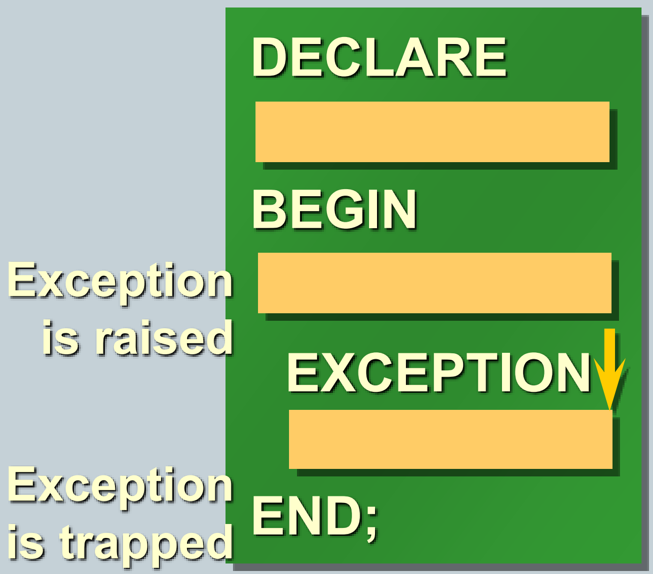
- PL/SQL Exception Handling

Handling Exceptions with PL/SQL

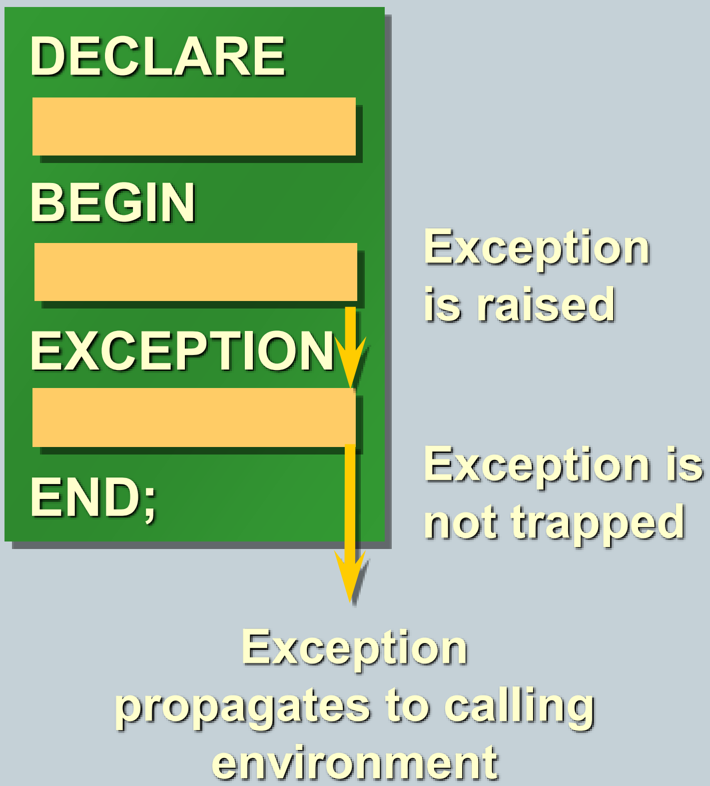
- What is an exception?
  - Identifier in PL/SQL that is raised during execution
- How is it raised?
  - An Oracle error occurs.
  - You raise it explicitly.
- How do you handle it?
  - Trap it with a handler.
  - Propagate it to the calling environment.

Handling Exceptions

Trap the exception



Propagate the exception



Exception Types

- Predefined Oracle Server
- Non-predefined Oracle Server
- User-defined

Implicitly raised

Explicitly raised

Trapping Exceptions

- Syntax

```
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;
    statement2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THEN
    statement1;
    statement2;
    . . .]
  [WHEN OTHERS THEN
    statement1;
    statement2;
    . . .]
```

Trapping Exceptions Guidelines

- WHEN OTHERS is the last clause.
- EXCEPTION keyword starts exception-handling section.
- Several exception handlers are allowed.
- Only one handler is processed before leaving the block.

Trapping Predefined Oracle Server Errors

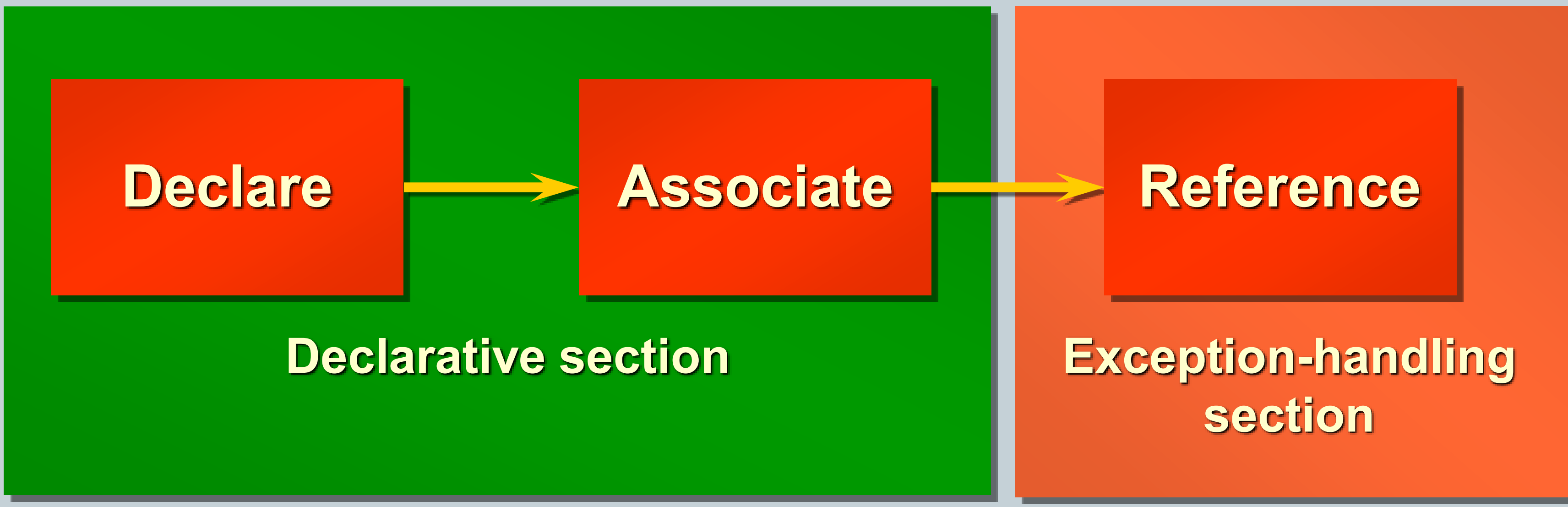
- Reference the standard name in the exception-handling routine.
- Sample predefined exceptions:
  - NO_DATA_FOUND
  - TOO_MANY_ROWS
  - INVALID_CURSOR
  - ZERO_DIVIDE
  - DUP_VAL_ON_INDEX

Predefined Exception

- Syntax

```
BEGIN  SELECT ... COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1;
    statement2;
  WHEN TOO_MANY_ROWS THEN
    statement1;
  WHEN OTHERS THEN
    statement1;
    statement2;
    statement3;
END;
```

Trapping Non-Predefined Oracle Server Errors



- Name the exception
- Code the PRAGMA EXCEPTION_INIT
- Handle the raised exception

Non-Predefined Error

- Trap for Oracle Server error number –2292, an integrity constraint violation.

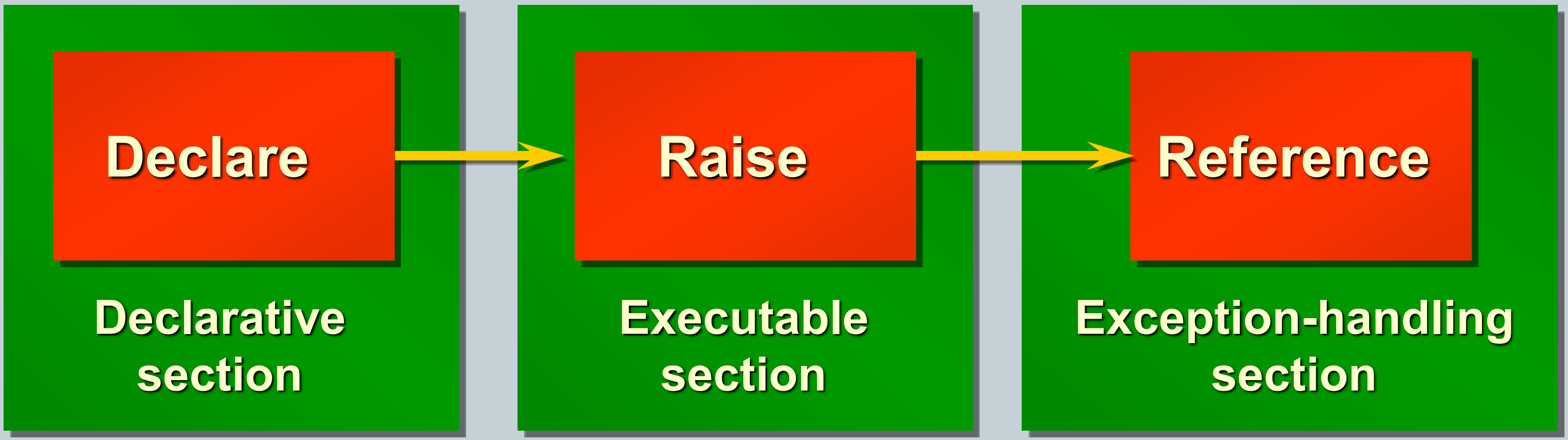
```
DECLARE
  e_emps_remaining    EXCEPTION;
  PRAGMA EXCEPTION_INIT (
            e_emps_remaining, -2292);
  v_deptno     dept.deptno%TYPE := &p_deptno;
BEGIN
  DELETE FROM dept
  WHERE       deptno = v_deptno;
  COMMIT;
EXCEPTION
  WHEN e_emps_remaining THEN
   DBMS_OUTPUT.PUT_LINE ('Cannot remove dept ' ||
   TO_CHAR(v_deptno) || '.  Employees exist. ');
END;
```

1

2

3

Trapping User-Defined Exceptions



- Name the exception
- Explicitly raise the exception by using the RAISE statement
- Handle the raised exception

User-Defined Exception

Example

```
DECLARE
  e_invalid_product  EXCEPTION;                              1
BEGIN
  UPDATE      product
  SET         descrip = '&product_description'
  WHERE       prodid = &product_number;
  IF SQL%NOTFOUND THEN
    RAISE e_invalid_product;                                 2
  END IF;
  COMMIT;
EXCEPTION
  WHEN  e_invalid_product  THEN                              3
    DBMS_OUTPUT.PUT_LINE('Invalid product number.');
END;
```

Functions for Trapping Exceptions

- SQLCODE

  Returns the numeric value for the error code
- SQLERRM

  Returns the message associated with the error number

Functions for Trapping Exceptions

- Example

```
DECLARE
  v_error_code      NUMBER;
  v_error_message   VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN
    ROLLBACK;
    v_error_code := SQLCODE ;
    v_error_message := SQLERRM ;
    INSERT INTO errors VALUES(v_error_code,
                              v_error_message);
END;
```

Calling Environments

SQL*Plus	Displays error number and message to screen
Procedure Builder	Displays error number and message to screen
Oracle Developer Forms	Accesses error number and message in a trigger by means of the ERROR_CODE and ERROR_TEXT packaged functions
Precompiler application	Accesses exception number through the SQLCA data structure
An enclosing PL/SQL block	Traps exception in exception-handling routine of enclosing block

Propagating Exceptions

Subblocks can handle an exception or pass the exception to the enclosing block.

```
DECLARE
  . . .
  e_no_rows       exception;
  e_integrity     exception;
  PRAGMA EXCEPTION_INIT (e_integrity, -2292);
BEGIN
  FOR c_record IN emp_cursor LOOP
    BEGIN
      SELECT ...
      UPDATE ...
      IF SQL%NOTFOUND THEN
        RAISE e_no_rows;
      END IF;
    EXCEPTION
      WHEN e_integrity THEN ...
      WHEN e_no_rows THEN ...
    END;
END LOOP;
EXCEPTION
  WHEN NO_DATA_FOUND THEN . . .
  WHEN TOO_MANY_ROWS THEN . . .
END;
```

RAISE_APPLICATION_ERROR Procedure

- Syntax

```
raise_application_error (error_number,
                         message[, {TRUE | FALSE}]);
```

  - A procedure that lets you issue user-defined error messages from stored subprograms
  - Called only from an executing stored subprogram

RAISE_APPLICATION_ERROR Procedure

- Used in two different places:
  - Executable section
  - Exception section
- Returns error conditions to the user in a manner consistent with other Oracle Server errors

THANK YOU

Advance Database Management System

Lecture 11:

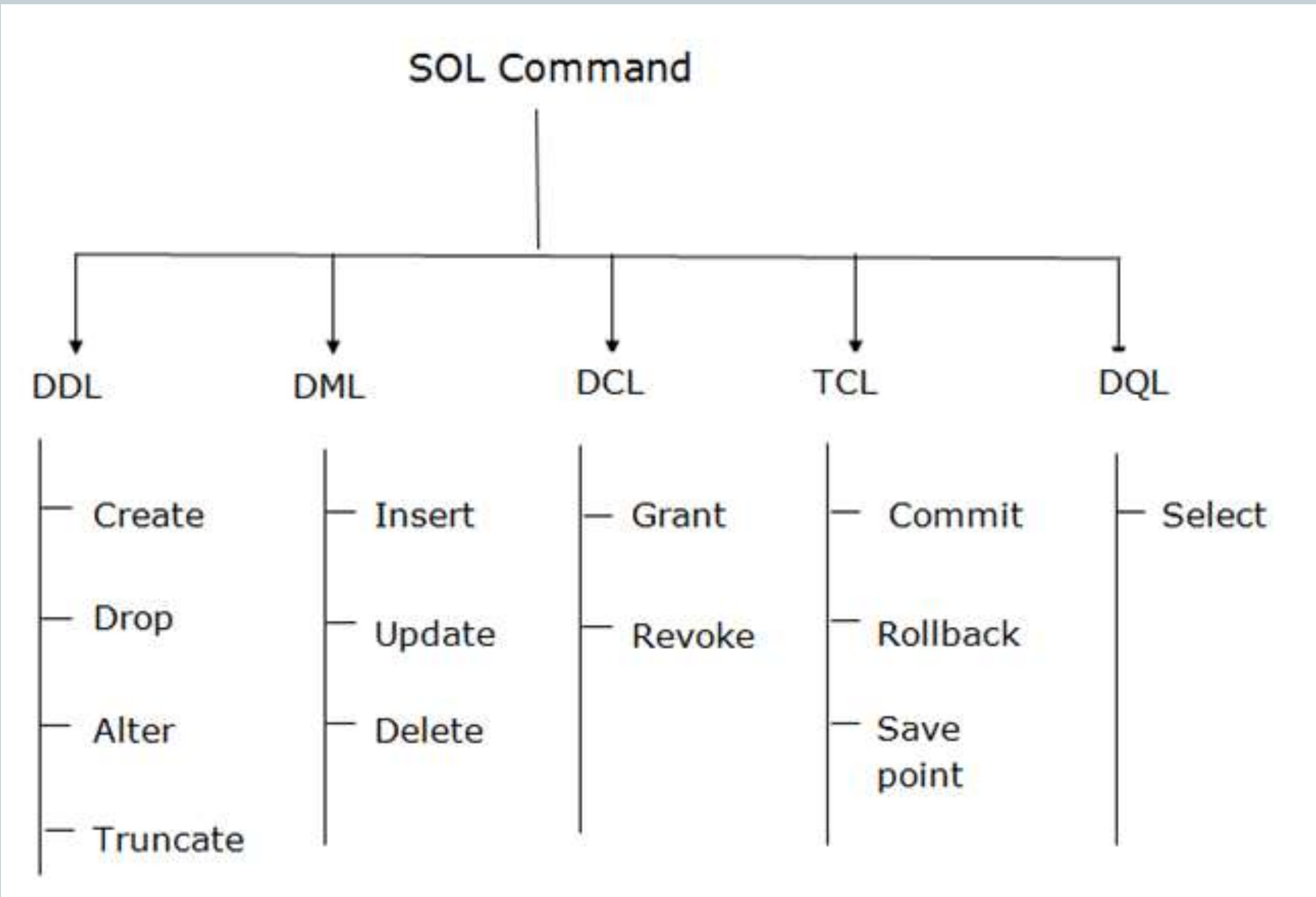
Database Transaction

Learning Objectives

To know about:

- Types of SQL Commands
- Database Transaction
- Transaction Control Language (TCL)
- COMMIT
- ROLLBACK
- SAVEPOINT

Types of SQL Commands



What is Transaction?

- A transaction is a set of SQL statements which Oracle treats as a Single Unit. i.e. all the statements should execute successfully or none of the statements should execute.
- To control transactions Oracle does not made permanent any DML statements unless you commit it. If you don't commit the transaction and power goes off or system crashes then the transaction is roll backed.

Transaction Control Language (TCL)

- Transaction control statements manage changes made by DML statements.
- TCL Statements available in Oracle are

  COMMIT : Make changes done in transaction permanent.
  ROLLBACK : Rollbacks the state of database to the last commit point.
  SAVEPOINT : Use to specify a point in transaction to which later you can rollback.

COMMIT

- To make the changes done in a transaction permanent issue the COMMIT statement

Example:

insert into emp (empno,ename,sal) values (101,’Abid’,2300);

commit;

ROLLBACK

- To rollback the changes done in a transaction give rollback statement. Rollback restore the state of the database to the last commit point.

  Example :

  delete from emp;

  rollback;

SAVEPOINT

- Specify a point in a transaction to which later you can roll back.

  Example

  insert into emp (empno,ename,sal) values (109,'Sami',3000);
  savepoint a;
  insert into dept values (10,'Sales','Hyd');
  savepoint b;
  insert into salgrade values('III',9000,12000);

THANK YOU

Advance Database Management System
Lecture 12:
Database Locking

Learning Objectives

To know about:

- Locking
- Types of Locking
- Example SQL
- Deadlock

Locking

In databases, locking is a mechanism used to manage concurrent access to data. It ensures data integrity and prevents conflicts when multiple users try to read or write the same data simultaneously.

Two ways of Locking:

1. Implicit Locking: The database automatically locks the necessary rows or tables during operations like UPDATE, DELETE, or INSERT.
2. Explicit Locking: Here, you manually request a lock on a row or table using locking clauses like FOR UPDATE or LOCK TABLE.

Locks Releasing:

1. Commit
2. Rollback
3. Rollback to Savepoint.

Implicit Locking

UPDATE accounts
SET balance = balance - 100
WHERE id = 1;

Here, the row with id = 1 is implicitly locked.
Other transactions cannot read or write to that row until this transaction commit or rollback.

Explicit Locking

*******************For update*******EXPLICIT LOCKING***********

SELECT * FROM TEMP_EMP2 WHERE COL1=1 FOR UPDATE;
SELECT * FROM TEMP_EMP2 FOR UPDATE;
SELECT * FROM TEMP_EMP2 FOR UPDATE NOWAIT;

THIS TABLE IS LOCKED AND CAN'T BE USED BY ANY OTHER USER UNTIL COMMIT
OR ROLLBACK IS IMPLEMENTED.

NOWAIT WILL TELL U WHETHER IT IS LOCKED BY OTHERS OR NOT.IF IT IS NOT
USED THEN IT WILL WAIT UNTIL LOCKS ARE RELEASED BY COMMIT OR ROLLBACK
STATEMENT.

EXAMPLE OF TESTING1:

SCOTT1:

```
SQL> SELECT * FROM TEMP_EMP2 FOR UPDATE;

     COL1       COL2
---------- ----------
        1          3
        1          3
        1          3
        1          3
        1          3
        1          3

6 rows selected.
```

ANOTHER SCOTT2:

```
SQL> UPDATE TEMP_EMP2 SET COL2=3
  2  WHERE COL1=1;
```

THE MACHINE WILL WAIT UNTIL LOCKS ARE RELEASED BY SCOTT1.fROM SCOTT1
WRITE COMMIT/ROLLBACK THEN SCOTT2 WILL EXECUTE THIS COMMAND.

EXAMPLE OF TESTING2:

SCOTT2:

```
SQL> UPDATE TEMP_EMP2 SET COL2=4
  2  WHERE COL1=1;

6 rows updated.
```

SCOTT1:

```
SQL> SELECT * FROM TEMP_EMP2 FOR UPDATE NOWAIT;
SELECT * FROM TEMP_EMP2 FOR UPDATE NOWAIT
              *
ERROR at line 1:
ORA-00054: resource busy and acquire with NOWAIT specified
```

THAT IS IT SPECIFIES THAT IT IS LOCKED BY OTHER AND CAN'T BE SELECTTED.
TO DO THIS FROM SCOTT2 EXECUTE COMMIT AND IT WILL WORK.

Explicit Locking


```
*****************EXPLICIT LOCKING**************************

LOCK TABLE TAB1,[TAB2],.....
                 IN [ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE |
                                  SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE ]
                 MODE [NOWAIT]

EXCLUSIVE :     ALLOW ONLY QUERY AND PROHIBIT ANY OTHER ACTIVITY.

SHARE:                            ALLOW ONLY QUERY AND PROHIBIT UPDATES.

ROW SHARE
SHARE UPDATE :                    BOTH ALLOWS CONCURRENT ACCESS TO TABLE.AND PROHIBIT
                                  OTHER USER TO LOCK ENTIRE TABLE EXCLUSIVELY.

ROW EXCLUSIVE:                   SAME AS SHARE UPDATE ALSO PROHIBIT LOCKING IN SHARED
                                 MODE.THESE LOCKS ARE ACQUIRED WHEN UPDATING,INSERTING
                                 OR DELETING.

SHARE ROW EXCLUSIVE: USED TO LOOK AT A WHOLE TABLE, TO SELECTIVE
                                    UPDATES AND TO ALLOW OTHER USERS TO LOOK AT ROWS
                                    IN THE TABLE BUT NOT LOCK THE TABLE IN SHARE MODE
                                    OR TO UPDATE ROWS.

NOWAIT:                           INDICATES THAT YOU DON'T WISH TO WAIT, IF RESOURCES
                                   ARE UNAVAILABLE.IF OMMITED, THE DBA WILL WAIT TILL
                                  RESOURCES ARE AVAILABLE.

LOCKS ARE RELEASED WHEN:
                 COMMIT, ROLLBACK
```

Explicit Locking

EXAMPLES:

1. SQL> LOCK TABLE TEMP_EMP2 IN EXCLUSIVE MODE NOWAIT;

Table(s) Locked.

2.SQL> LOCK TABLE TEMP_EMP2 IN SHARE MODE NOWAIT;

Table(s) Locked.

ALLOWS ONLY SELECT STATEMENT.

3.
SQL> LOCK TABLE TEMP_EMP2 IN ROW SHARE MODE NOWAIT;

Table(s) Locked.
ALLOWS INSERT/UPDATE/DELETE. DON'T ALLOWS OTHER USER TO PERFORM EXCLUSIVE LOCK ON THE TABLE.

Deadlock

A deadlock in a database occurs when two or more transactions are waiting for each other to release locks, but none of them can proceed because they're all stuck in a cycle of waiting.

Example:

Transaction A locks Row 1 and wants to update Row 2.
Transaction B locks Row 2 and wants to update Row 1.
Now:

- A is waiting for B to release Row 2.
- B is waiting for A to release Row 1.

Neither can proceed — this is a deadlock.

THANK YOU