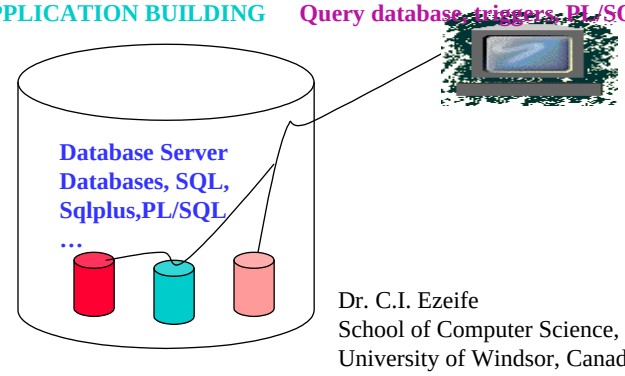
60-415: Advanced and Practical Database Systems (with Oracle PL/SQL and Form)

GETTING OUR HANDS DIRTY
:DB APPLICATION BUILDING

Database client, Forms,
Query database, triggers, PL/SQL,..



Dr. C.I. Ezeife
School of Computer Science,
University of Windsor, Canada.
Email: cezeife@uwindsor.ca

Course Objectives

- Broad Course Objective
- - Components of a database management system
- - Acquire database development skills necessary for building real life database applications with Oracle DBMS.
- Reference Materials
  - C.I Ezeife, *Custom Course Ware, Course Notes for 60-415*, Project Using Selected Tools: Advanced and Practical Database Systems (with Oracle PL/SQL and Form), University of Windsor, Fall 2006.
  - Benjamin Resenzweig and Elena Silverstrova, "Oracle PL/SL Interactive Workbook", The Prentice Hall PTR Oracle Series, 2003 edition 2, ISBN 0-13-047320-0
  - Baman Motivala, "Oracle Forms Interactive Workbook", The PTR Oracle Series, 2000 edition 1, ISBN 0-13-015808-9
  - Alex Morrison & Alice Rischest, "Oracle SQL Interactive Workbook", The PTR Oracle Series, 2003 edition 3, ISBN 0-13-145131-6

Course Objectives

- **Companion web site:**
  http://www.phptr.com/Rosenzweig

  http://www.phptr.com/phptrinteractive

  http://www.phptr.com/motivala

Course Objectives

- **Detailed Course Objective**
- **Part A: DBMS Components (Review)(slides 11 – 68)**
  - **DB design and Normal Forms**
  - **SQL DDL & DML**
  - **Oracle SqlPlus (slides 7 – 10; slide 26)**
  - **File Organizations and Indexing**
  - **Query Optimization**
  - **Transaction Processing**
- **Part B: Oracle Database Development (slides 69 – 204)**
  - **Oracle PL/SQL**
    - **Oracle PL/SQL summary (slides 205 – 377)**
- **Part C: Database Development (GUI) (slides 205 – 377)**
  - **Oracle Forms**
    - **Oracle Forms Summary (slides 207 – 219)**

Hardware and Software Requirements

- **Software Requirements**
  1. **Oracle DB Server (e.g., Oracle 10g R2 or a slightly lower version)**
  2. **Oracle DB Client (e.g., Oracle 10g)**
  3. **Sqlplus**
  4. **Access to WWW**
  5. **Windows OS (e.g., XP) and / or Unix OS (e.g., Solaris)**
- **Hardware Requirements**
  1. **A Personal Computer (e.g., 1 GHz processor, --Memory)**
  2. **A Unix Multiprocessor System (e.g., sol or luna systems)**

Hardware and Software Requirements

- **Note that both the software 1 & 2 can reside on the same computer or on two separate computers. Also, while the Oracle client software [e.g., Oracle 10g client] is most suitable on a Windows based PC, the Oracle Server software can reside on a Unix machine (like CS sol / luna)**

SQLPLUS

- Sqlplus is the software for executing SQL stmts (Sqlplus is to SQL stmts what C compiler is to C programs)
- How to end an SQL command in Sqlplus

SQL command can be ended in Sqlplus in one of the following 3 ways:

- with a semicolon (;)
- with a forward (/) on a line
- with a blank line

SQLPLUS

- The SQL Buffer
  - Sqlplus stores recently typed SQL command or PL/SQL block in an area of memory called SQL buffer.
  - The SQL buffer remains unchanged until a new command is entered or you exit sqlplus.
  - The SQL buffer can be edited by typing EDIT at SQL prompt.
  - While SQL and PL/SQL stmts are captured in the SQL buffer, Sqlplus commands (e.g., SET LINE…) are not captured in the buffer.

SQLPLUS

- When you create stored procedures, functions or packages, you begin with CREATE command.
- When you begin a PL/SQL block, you start by entering the word DECLARE or BEGIN
- Typing either BEGIN, DECLARE or CREATE puts the Sqlplus into PL/SQL mode.

- Running PL/SQL Blocks in Sqlplus
- How to End a PL/SQL block in Sqlplus
  - A PL/SQL block is ended with a period (.).

SQLPLUS

- How to Execute a PL/SQL Block in Sqlplus
  - A PL/SQL block is executed with a forward slash (/) or RUN
  - A PL/SQL program can be edited in sqlplus using EDIT
  - A PL/SQL program can be saved as a script file with a .sql extension. In that case, the file should be ended with a period to mark end of program, and followed with a forward slash (/) to execute the program when loaded.
  - To execute a script file in PL/SQL, use @filename.sql
- E.g., sql>@scriptfile.sql

Part A: DBMS Components (Review)
DBMS OVERVIEW(What are?)

- **What is a database? : It is a collection of data, typically describing the activities of one or more related organizations, e.g., a University, an airline reservation or a banking database.**
- **What is a DBMS?: A DBMS is a set of software for creating, querying, managing and keeping databases. Examples of DBMS's are DB2, Informix, Sybase, Oracle, Microsoft Access (relational).**
- **Alternative to Databases: Storing all data for university, airline and banking information in separate files and writing separate program for each data file.**

Components of a DBMS

- **A DBMS has the following basic components**
  - **1. A specific data model: the data structure for logically representing data by the DBMS (e.g., relational, object-oriented, hierarchical etc.).**
  - **2. Database Design and Tuning: Allows schema design at the conceptual level (e.g., normalization, fragmentation of data and performance tuning)**
  - **3. A Data definition and data manipulation language for creating files in the database and querying the database (e.g., SQL, QBE)**
  - **4. File Organization techniques for storing data physically on disk efficiently (e.g., B+-tree indexing or ISAM indexing).**

Components of a DBMS

- 5. Query Optimization and Evaluation facility: helps to generate the best query plan for executing a query efficiently.
- 6. Transaction Processing
  - Concurrency Control and recovery: Allowing more than one user access data concurrently and maintaining a consistent and correct data even after hardware or software failure.
  - Database Security and Integrity Issues: Protecting data from inconsistent changes made by different concurrent users.

1. DBMS Data model

- Data model provides the data structure that the database is stored in, and the operations allowed on this data structure.
- Some existing DBMS data models are relational, entity-relationship model, object-oriented and hierarchical data model.
- Schema in the relational model is used to describe the data in the database.

1. DBMS Data model

nstudent

Studid	Name	gpa
53666	Jones S	3.4
53688	Smith M	3.2
53650	Smith J	3.8
53831	Madayan A	1.8
53832	Guldin	2.0

Example of a relational instance of the table student is given above. Example of integrity constraint that can be defined on this table is “Every student has a unique id”.

1. DBMS Data model

- **Data in DBMS are described in 3 levels of abstraction namely:**
  - **External schema (representing how different users view the data). E.g., view for students with gpa > 3.2**
  - **Conceptual Schema (logical schema) - data described in terms of data model (e.g.)**
    - **student (stuid:string, name:string, gpa:real)**
    - **faculty (fid:string, fname:string, salary:real)**
    - **courses (cid:string, cname:string, credits:integer)**
    - **Rooms (rno: integer, address:string, capacity:integer)**
    - **Enrolled (stuid:string, cid:string, grade: string)**

1. DBMS Data model

- Teaches (fid:string, cid:string)
- Meets_In (cid:string, rno:integer, time:string)

- **Physical Schema: describes how data are actually stored on disks and tapes including indexes. Example physical design are:**
  - **store all relations as unsorted files of records**
  - **create indexes on the first column of student, faculty and course relations, the salary column of faculty and capacity column of rooms.**

2. Database Design and Tuning model

- **The steps in database design are:**
- **1. Requirements analysis: information about environment gathered**
- **2. Conceptual & Logical Design: Presents a high-level description of data and relationship between data entities (e.g., ER model). The second part is the logical design, which converts the ER model to relational database schema and applies refinement guided through the powerful theory of normalization.**
- **3. Physical Database Design: Here indexes are built on relations, tables are clustered or re-designed using information about work load to improve performance.**
- **4. Database Tuning: Uses interaction between 3 steps above to achieve better performance.**
- **5. Security Design: Identify user groups and roles (of privileges) are assigned to appropriate user groups. Example user groups are DBA. PUBLIC.**

2. Quick Review of ER Model

- The Entity-Relationship (ER) data model allows us to describe the data involved in a real world enterprise in terms of objects and their relationships.
- An entity is an object in the real world (e.g., Student, Faculty, Courses, Rooms)
- A relationship is an association among two or more entities (e.g., Enrolled, Teaches, Meets_In).
- An entity set (e.g., student), has a collection of similar entities described by the set of attributes (e.g., stuid, name, gpa).
- Each attribute has a domain of possible values (e.g., domain of gpa is 0 to 13)
- An entity set (e.g., student) is represented by a rectangle

2. Database Design and Tuning model (ER model)

- An entity set is represented by a rectangle, an attribute by an oval with each primary key attribute underlined.
- A relationship is represented by a diamond box and a relationship is uniquely identified by the participating entities.
- An arrow from an entity to a relationship places a key constraint requiring that each entity value has only one such relationship.
- A relationship set can be one-to-many (eg. Meets-in: A room is used for teaching many courses but no two courses are meeting in same room at the same time).
- A relationship can also be many-to-many (e.g. Enrolled: a student can enroll in several courses and a course can have several students enrolled in it.
- A relationship can as well be one-to-one (e.g. Teaches: if a faculty is allowed to teach only one course and a course is taught by only one faculty)

2. Database Design and Tuning model

- **Schema Refinement and Normal Forms**
- **Schema refinement in relations is an approach based on decomposition of relations.**
- **This is intended to address problems caused by redundant storage of information which are: wasting storage, update anomalies, insertion anomalies and deletion anomalies.**

Ssn	name	lot	rating	wage	hours
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Hourly-Emps Relation

2. Database Design and Tuning model

- **Assume that wage attribute is determined by rating attribute. And if same rating appears in the rating column of two tuples, then same value must appear in the wage column.**
- **Since rating 8 corresponds to wage 10, this information is repeated. If we change wage for tuple 1 to 9, this is update anomaly. We can not insert a tuple for an employee unless we know her hourly rating, which is insertion anomaly.**
- **If we delete all tuples with given rating, we lose the association between rating value and wage (deletion anomaly)**

2. Database Design and Tuning model

- **Functional dependencies and other integrity constraints force an association between attributes that may lead to redundancy.**
- **For a non-empty set of attributes X, Y in relation R, functional dependency FD X → Y is read as X functionally determines Y and is true if the following holds for every pair of tuples t_1 and t_2 in R.**

  t_1. X = t_2. X, then, t_1. Y = t_2 . Y
- **Note that both sides of an FD contain sets of attributes**
- **Many of the problems of redundancy can be eliminated through decomposition of relations guided by the theories of normal forms summarized as:**

2. Database Design and Tuning model

- **The normal forms based on FDs are first normal form (1NF), second (2NF), third (3NF) and Boyce-Codd normal form (BCNF)**
- **A relation is in 1NF if every field contains only atomic values.**
- **Second normal form (2NF) is historical and its concerns are better taken care of by 3NF. A relation is in 2NF if every non-key attribute is fully functionally dependent on the primary key.**
- **R is in 3NF if every non-key attribute is non-transitively dependent on the primary key.**
- **R is in BCNF if every determinant is a candidate key and prevents forming relations with multiple composite keys that overlap.**
- **A relation R is in 5NF if for every JD (join dependency) that holds over R one of the following is true: join of Ri = R for all i, or That is, if a join of relations Ri gives back R without any loss of a tuple.**

3. Data Definition and Manipulation Languages (DDL & DML): Overview of Operations

- **Basic DDL and DML operations for SQL (structured query lang):**
  - **1. Create tables [Create Table …]**
  - **2. Destroy tables [Drop Table …]**
  - **3. Change tables [Alter Table ….]**
  - **4. Insert Data into Tables [Insert Into ….]**
  - **5. Delete Data from Tables [Delete from …]**
  - **6. Update Data in Tables [Update ….]**
  - **7. Query Tables [Select … from … where …]**
  - **8. Find the structure of DB, relation, view, index, etc.[querying catalogue with select * from tab; select * from cat; Desc Table; etc.]**

3. (DDL & DML): How To Run SQL

- **In order to create and query database tables, user needs to connect to SQL interpreter called Sqlplus as follows:**
- **1. From Unix account, type:**
  >Sqlplus
  >username@cs01
  >password
  2. To quit sqlplus, type:
  > exit
  3. To load and run a .sql file, type:
  > @filename
  4. To execute a SQL command like "Create Table …", type:
  > Create Table student(stuid VARCHAR(20), ….;

**** A Summary Handout of other Sqlplus commands for doing various things is handed out in class and posted on the course web site.**

3. DDL & DML: Create Tables

- Example of a DDL and DML language is structured query language (SQL).
- DDL is used to create and delete tables and views, and to modify table structures. E.g. of an SQL instruction for creating the table student is:

```
CREATE TABLE student (stuid VARCHAR2(20),
        name VARCHAR2 (20),
        gpa   NUMBER(5,2),
        PRIMARY KEY (stuid));
```


3. DDL & DML: Destroy and Alter Tables

- To destroy a table or view, e.g., student, use:
  - DROP TABLE student RESTRICT;
  - or
  - DROP TABLE student CASCADE;
- The RESTRICT keyword prevents the table from being destroyed if some integrity constraints are defined on it. The CASCADE keyword destroys both.
- To modify table structure, use:
  - ALTER TABLE student
    ADD COLUMN major CHAR (20);

DDL & DML: Insert, Delete, Update Data into/from Tables

- An example instruction for inserting a record into the created student table is:
- Insert into student(stuid, name, gpa) values ('53666', 'Jone S', 3.4);
- To delete the inserted tuple, use:
- Delete from student s
  where s.name = 'Jone S';
- To Update the student table, use an instruction like:
- Update student s
  set s.gpa = s.gpa + 10
  where s.gpa >= 3.5;

3. DDL & DML: Querying Tables

- The DML subset of SQL is used to pose queries and to insert, delete and modify rows of tables.
- For example, to query the student table in order to print the ids, names and gpas of all students with gpa > 3.2, we use:
  - SELECT s.stuid, s.name, s.gpa
    FROM student s
    WHERE s.gpa > 3.2;
- The basic form of an SQL query is as follows:

3. DDL & DML: Querying Tables

- **SELECT [DISTINCT] select-list**
  FROM from-list
  WHERE qualification;
- **from-list is a list of table names possibly followed by a range variable.**
- **Select-list is a list of (expressions) column names of tables from the from-list.**
- **The qualification is a Boolean combination in the form *expression op expression* with possible connectives (AND, OR, NOT)**

3. DDL & DML: Querying Tables

- **These tables model sailors reserve boats world**

Sailors: instance S1

Sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Sid	bid	day
22	101	10/10/96
58	103	11/12/96

Reserves: instance R1

Sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.0
44	guppy	5	35.0
58	rusty	10	35.0

Sailors: instance S2

3. DDL & DML: Querying Tables

- DISTINCT keyword is optional and eliminates duplicate tuples.
- E.g., find the names of sailors who have reserved boat number 103.
  - Select S.sname
    from sailors S, Reserves R
    where S.sid = R.sid
    And R.bid = 103;

3. DDL & DML: Querying with Aggregate, Set and other Operators

- Aggregate Operators
- SQL supports a more general version of column list
- Each item in a column list can be of the form expression AS column-name, where expression is any arithmetic or string expression over column names and constants.
- A column-list can also contain aggregates (sum, count, avg, min, max).
- It supports pattern matching through the LIKE operator, along with the use of wild-card symbols

3. DDL & DML: Querying with Aggregate, Set and other Operators

- % (zero or more characters) and - (exactly one arbitrary character).
- E.g., Find the ages of sailors whose name begins and ends with B and has at least three characters.
  - SELECT S.age
    from sailors
    where s.name LIKE 'B - %B';
- SQL supports the set operations unions, intersect and difference under names UNION, INTERSECT, and EXCEPT.

3. DDL & DML: Nested Queries

- Nested Queries
- A nested query is a query that has another query embedded within it. The embedded query is called subquery.
- E.g., Find the names of sailors who have reserved boat number 103.
  - Select s.same
    from sailors S
    where s.sid IN (Select R.sid
    from Reserves R
    where R.bid = 103);

3. DDL & DML: Nested Queries

- Can also be expressed as:
  - Select S.sname
    from sailors S
    where Exists (Select *
    from Reserves R
    where R.bid = 103
    And S.sid = R.sid);
- The latter version is a correlated nested query
- Other set comparison operators are UNIQUE, ANY, ALL.
- IN and NOT IN are equivalent to =ANY and <>ALL respectively.

3. DDL & DML: Querying with Group By and Having clauses

- General SQL form is
  - Select [Distinct] select-list
    from from-list
    where qualification
    Group by grouping-list
    Having group-qualification;
  - Example query is: Find the age of the youngest adult sailor for each rating level with at least 2 such sailors.

3. DDL & DML: Querying with Group By and Having clauses

- **Solution is:**
- **Select s.rating, Min(s.age) As minage**
  from sailors s
  where s.age >= 18
  group by s.rating
  Having count(*) > 1;
- **Result is:**

Rating	minage
3	25.5
7	35.0
8	25.5

3. DDL & DML: Null Values

- **Null Values**
- **A new sailor, Bob, may not have a rating assigned, leaving the data value for this column unknown.**
- **Some columns may be inapplicable to some sailors, e.g., column maiden-name is inapplicable to men and single women sailors.**
- **SQL provides a column value for these kinds of situations.**
- **SQL provides a comparison operator to test if a column value is null (IS NULL) and (IS NOT NULL).**

3. Querying System Catalogs

- **A DBMS maintains information about every relation, index, views that it contains which are stored in a collection of relations called system catalog.**
- **System catalog has information about each relation**
  - **its name, filename, file structure**
  - **name and type of each of its attributes**
  - **index name of each index on the table**
  - **integrity constraints, number of tuples**
  - **name and structure of the index**
  - **for each user, accounting and authorization information. Etc.**
  - **Select * from cat; select * from tab; Desc tablename; are some ways to query the catalog.**

3. DDL & DML- Embedded SQL [Optional Part]

- **Building db applications with nice graphical user interface would require facilities provided by general purpose langs in addition to SQL. The use of SQL commands within a host lang program is called embedded SQL.**
- **In embedded SQL, SQL statements are used wherever a stmt in the host lang is allowed and SQL stmts are clearly marked. Eg., in C (e.g., Oracle Pro*C), SQL stmts must be prefixed by EXEC SQL**
- **Any host lang variable for passing arguments into an SQL command must be declared in SQL. Such host lang variables must be prefixed by (:) in SQL stmts and be declared between the commands EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION.**

3. DDL & DML- Embedded SQL [Optional Part]

- E.g., in embedded C, we can declare variables c-sname, c-sid, c-rating and c-age as follows:

```
EXEC SQL BEGIN DECLARE SECTION
        char c-sname[20];
        long c-sid;
        short c-rating;
        float c-age;
EXEC SQL END DECLARE SECTION
```

The above are C variables in C data types to be read and set in an SQL run time environment.

The SQL data types corresponding to the various C types are SQL CHAR(20) for C's char[20], SQL's INTEGER for C's long, SQL's SMALLINT for C's short, SQL's NUMBER(N, D) for C's float.

3. DDL & DML- JDBC/ODBC [Optional Part]

- ODBC (open database connectivity) and JDBC (Java database connectivity) also allow integration of SQL with a general purpose programming lang.
- ODBC and JDBC connect to databases through application programming interface (API).
- ODBC and JDBC connectivity provide more portable access to different database management systems than embedded SQL.
- With ODBC and JDBC, all interactions with a specific DBMS occurs through a DBMS specific driver.
- The driver is responsible for translating ODBC or JDBC calls into DBMS-specific calls.
- Available drivers are registered with a driver manager.

3. DDL & DML- JDBC/ODBC [Optional Part]

- JDBC is a collection of Java classes and interfaces for enabling database access from programs written in Java lang.
- The JDBC classes and interfaces are part of the java.sql package. Thus, all Java database applications should include at the beginning

  import java.sql.*

3. DDL & DML- Stored Procedures like PL/SQL

- A stored procedure is a program executed through a single SQL stmt locally executed and completed within the process space of the database server.
- Once a stored procedure is registered with the db server, different users can re-use it.
- All major db systems provide ways for users to write stored procedures in a simple general purpose lang close to SQL – e.g., Oracle PL/SQL.
- Part B of course teaches Oracle PL/SQL in detail.

3. Relational Algebra and Calculus

- Two formal query langs. associated with the relational model are relational algebra and calculus.
- A relational algebra operator accepts one or 2 relation instances as arguments and returns a relation instance as output
- Basic algebra operators are for selection, projection, union, cross-product and difference.
- There are some additional operators defined in terms of basic operators (e.g., Joins – conditional, equijoin, natural, outer (theta, left and right outer) joins).

3. Views

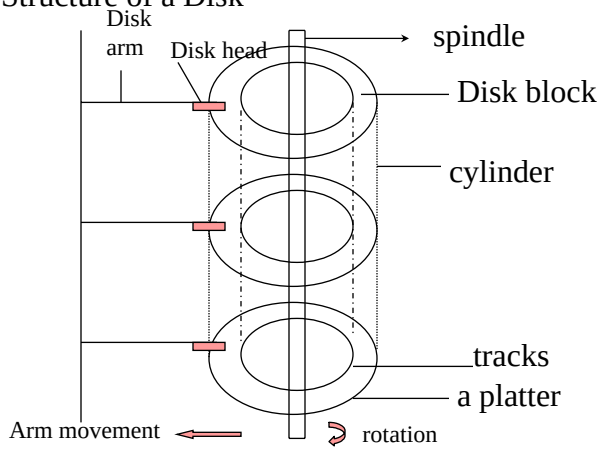
- Views are tables that are defined in terms of queries over other tables and its rows are not generally stored explicitly in the database but computed from definition.
- The view mechanism can be used to create a window on a collection of data that are of interest to a group of users, and it provides logical data independence since changes in the base tables do not affect the view design.
- The following query creates a view to find the names and ages of sailors with a rating > 6, and include the dates.
  - CREATE VIEW ActiveSailors(name, age, day)
    AS SELECT S.name, S.age, R.day
    FROM Sailors S, Reserves R
    WHERE S.sname = R.sname AND S.rating > 6;

4. File Organization techniques

- **Data in a DBMS are stored on storage devices such as disks and tapes.**
- **The file manager issues requests to disk manager to allocate or free space for storing records in units of a page (4KB or 8KB).**
- **The file manager determines the page of a requested record and requests that this page be brought to the buffer pool (part of memory) by the buffer manager.**
- **The disk composition is shown in the following figure.**

4. File Organization techniques

Structure of a Disk



4. File Organization techniques

- The time to access a disk block is:
  Seek time + Rotational delay time + Transfer time
- Seek time is the time to move disk heads to the track on which a desired block is located.
- Rotational delay is the waiting time for the desired block to rotate under the disk head.
- Transfer time is the time to actually read or write the data in the block once the head is positioned.
- To minimize disk I/O time, records should be stored such that frequently used records be placed close together.

4. File Organization techniques

- The closest we can place two records on disk is on the same block, or then on the same track, same cylinder or adjacent cylinder in decreasing order of closeness.
- Pages of records are stored on disk and brought up to memory when any record in them are requested by a database transaction.
- Thus, the disk manager organizes a collection of sequential records into a page.
- Higher levels of DBMS code treat a page as a collection of records and a file of records may reside on several pages. How can pages be organized as a file?

4. File Organization techniques

- **The possible file structures are:**
  - **1. Heap files: keep unordered data in pages in a file (called heap file). To support inserting, deleting a record, creating and destroying files, there is need to keep track of pages in a heap file using doubly linked list of pages or a directory of pages.**
  - **2. Ordered files: records are stored in an order in data pages of the file.**
  - **3. Indexes: a file of ordered records for quickly retrieving records of the original data file.**

4. Indexes

- **Assume we have a database file of 1 million records with structure (student id, name, gpa), to get the students with gpa > 4.0, we need to scan the 1 million records. Slow approach.**
  - **A way to speed up processing of queries is build an index on the gpa attribute and store as an index file, which stores the records in gpa order.**
  - **An index is an auxilliary data structure that helps to find records meeting a selection condition.**
  - **Every index has an associated search key, a collection of one or more fields of the file we are building the index; any subset of the field can be a search key.**
  - **Indexed file speeds up equality or range selections on the search key and quick retrieval of records in index file is done through access methods.**

4. Indexes

- Examples of access methods (organization techniques for index files) are B+ trees, hash-based structures
- A database table may have more than one index file.
- A clustered index has its ordering the same or close to the ordering of its data records in the main database table. E.g., index on student id is clustered while that on gpa is unclustered.
- A dense index contains at least one data entry for every search key value that appears in a record in the table.
- A non-dense or sparse index contains one entry for each page of records in the data file.
- A primary index includes the primary key as its search key while a secondary index is an index defined on a field other than the primary key.

4. Indexes

- Tree-Structured Indexing
- Assume we have the students file sorted on gpa,
- To answer the range query "Find all students with gpa higher than 3.0, we identify the first such student by doing a binary search of the file and then scan the file from that point on.
- An ISAM tree is a static structure which is effective when the file is not updated frequently.
- B+ tree is a dynamic structure that adjusts to changes (addition and deletion) in the file gracefully.

4. Indexes – Creating in Oracle

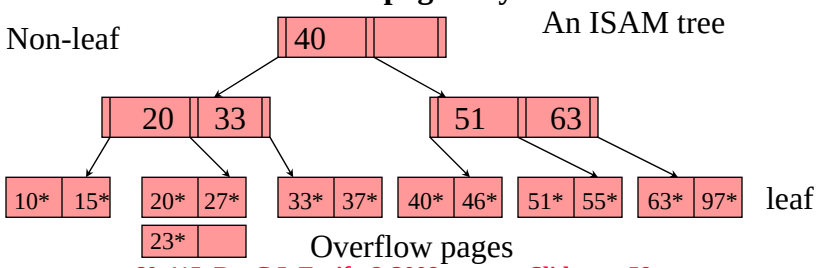
- In Oracle, you can create an index with the general syntax:
  Create [Unique|Bitmapped] index indexname
  ON tablename
  (Column|Col_expression[,column|col_expression …);
- Example:
- Create INDEX sect-location_i
  ON section(location);
- A subsequent like this below will take advantage of this index by retrieving rows faster than sequentially.
- Select course-no, sect-no
  from section
  where location = 'L206';

4. Indexes

- B+ tree supports equality and range queries well
- In ISAM index structure there are data pages, index pages and overflow pages.
- Each tree node is a disk page and all the data reside in the leaf pages.
- At file creation, leaf pages are allocated sequentially and sorted on key value. Then the non-leaf pages are allocated.
- Additional pages needed because of several inserts are allocated from an overflow area.

4. Indexes (ISAM)

- **The basic operations of insert, delete and search are accomplished by searching for the non-leaf node less or equal to the search key and following that path to a leaf page where data is inserted, deleted or retrieved. An overflow page may need to be checked.**



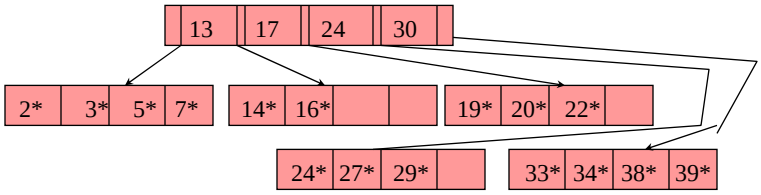
4. Indexes (ISAM)

- **An insert operation of record 23 causes an overflow page since each leaf page holds only 2 records. Inserts and deletes affect only leaf pages**
- **Number of disk I/O is equal to the number of levels of the tree and is $\log_F P$ where P is the number of primary leaf pages and F is the fan out or number of entries per index page. N is P * F.**
- **This is less than number of disk I/O for binary search, which is $\log_2 N$ or $\log_2 (P * F)$. E.g., with 64 entries, 32 pages and 2 entries per page, ISAM's disk I/O is 5 while binary search disk I/O is 6.**

4. Indexes (B+ trees)

- **B+ tree search structure is a balanced tree in which the internal nodes direct the search and the leaf nodes contain the data entries.**
- **Leaf pages are linked using page pointers since they are not allocated sequentially.**
- **Sequence of leaf pages is called sequence set.**
- **It requires a minimum occupancy of 50% at each node except the root.**
- **If every node contains m entries and the order of the tree (a given parameter of tree) is d, the relationship $d \leq m \leq 2d$ is true for every node except the root where it is $1 \leq m \leq 2d$.**
- **Non-leaf nodes with m index entries contain m+1 pointers to children.**
- **Leaf nodes contain data entries.**

4. Indexes (B+ trees)



A b+ tree of height 1, order d=2

- **Insertion of 8 into the tree leads to a split of leftmost leaf node as well as the split of the index page to increase the height of the tree.**
- **Deletion of a record may cause a node to be at minimum occupancy and entries from an adjacent sibling are then redistributed or two nodes may need to be merged.**

5. Query Optimization and Evaluation

- Queries are parsed and then presented to a query optimizer which is responsible for identifying an efficient execution plan for evaluating the query.
- The goal of a query optimizer is to find a good evaluation plan for a given query.
- A query evaluation plan consists of an extended relational algebra tree with annotations indicating the access methods to use for each relation and the implementation method to use for each relational operator
- Result sizes may need to be estimated and the cost of the plans estimated.
- The goal of a query optimizer is to find a good evaluation plan for a given query.

6. Transaction Processing

- Concurrency control and Recovery
- A transaction is a DML statement or group of statements that logically belong together.
- The group of statements is defined by two commands:

  COMMIT and ROLLBACK in conjunction with the SAVEPOINT command.
- An interleaved execution of several transactions is called a schedule.
- An execution of a user program or transaction is regarded as a series of reads and writes of database objects.
- The important properties of database transactions are ACID for atomicity, consistency, isolation and durability.

6. Transaction Processing

- Assume we have two transactions T1 and T2, defined as follows:
  - T1: $R_1(A)$, $W_1(A)$, $R_1(C)$, $W_1(C)$
  - T2: $R_2(B)$, $W_2(B)$
- A schedule for running T1 and T2 concurrently should produce the same effect as running T1, T2.
- One such schedule is:
  - $R_1(A)$, $W_1(A)$, $R_2(B)$, $W_2(B)$, commit(T2), $R_1(C)$, $W_1(C)$, commit(T1)
- Approaches for concurrency control include (1)strict two-phase locking (strict 2PL), (2) 2 Phase locking, serializability and Recoverability, (3) View Serializability, (4) Optimistic concurrency control and (5) Timestamp-based concurrency control.

6. Crash Recovery

- The recovery manager is responsible for atomicity (ensuring that actions of uncommitted transactions are undone) and durability (ensuring that actions of committed transactions survive system crashes and media failures).
- It keeps a log of all modifications on stable storage. The log is used to undo the actions of aborted and incomplete transactions and to redo the actions of committed transactions.

- DATABASE SECURITY
- Issues of interest in a secure database are secrecy, integrity and availability
- Secure policy and mechanisms are needed to enforce this

6. Database Administrator

- **Role of the Database Administrator (DBA) are:**
  - **1. Creating new accounts – granting privileges to database users as follows:**

    Create User Music
    identified by listen;

    Grant All to Music;
  - **2. Mandatory control issues: must assign security classes to each database object and security clearance to each authorization id in accordance with the chosen security policy.**
  - **3. Maintaining Audit trail: log of updates or all actions, etc.**

6. Database Administrator

- **A privilege is a right to execute a particular type of SQL statements. Two types exist – system and object privileges.**
- **A system privilege or an object privilege is granted to a user with GRANT command.**
- **A role is a collection of privileges.**
- **Example system privilege is right to create a table. E.g. object privilege is that to select from an Instructor table. Object privileges are granted for a particular object.**
- **To extend an object (table, index, views) privilege to another user, you must be the object owner and should have been given this GRANT privilege with the GRANT OPTION.**

Part B: Oracle Database Development (Oracle PL/SQL)

- **PL/SQL in Client/Server Architecture**
  - **Oracle applications can be built using client-server architecture where the Oracle database resides on the server and the program that requests data and changes on the database resides on a client machine.**
  - **The client program can be written in C, Java or PL/SQL**
  - **PL/SQL is not a stand-alone programming language like C or Java, but is part of the Oracle RDBMS.**
  - **PL/SQL can reside in two environments – client side and server side.**
  - **PL/SQL blocks are processed by PL/SQL engine, a special component of such Oracle products as Oracle server, Oracle Forms, Oracle Reports.**
  - **The SQL processor resides only on the Oracle server.**

PL/SQL Formatting Guide

- **PL/SQL Formatting Guide**
- **CASE**
- **PL/SQL is case-insensitive [use upper case for Reserved keywords and lower case for others].**
- **WHITE SPACE**
- **Use proper indentation for readability.**
- **NAMING CONVENTIONS**
- **Use appropriate prefixes to distinguish identifiers standing for variables (eg, v_studentid), cursor (c_studentid), record (r_studentid), table (t_studentid), exception(e_studentid), etc.**

Oracle PL/SQL

- PL/SQL processor sends SQL statements to the SQL processor to process when encountered.

- The PL/SQL Block Structure
  - The most basic unit in PL/SQL is a block
  - All PL/SQL programs are combined into blocks that are nested within each other.
  - PL/SQL blocks can be named or anonymous.
  - Named blocks are used for subroutines (which are procedures, functions and packages)
  - PL/SQL block has 3 sections: declaration section (optional), executable section (mandatory) and exception – handling section (optional).

Part B: PL/SQL IN A WRAP (slide 1 of 6)

- PL/SQL Program or block has a type and a structure as:
  - T: PL/SQL block Type
  - S: PL/SQL block Structure
- T: PL/SQL block Type
  - T1: Anonymous block (e.g., sl 78)
  - T2: Named block
    - T2.1: Procedure (e.g., sl 156 - 165)
    - T2.2: Function (e.g., sl 166 - 168)
    - T2.3: Package (e.g., sl 169 – 177)

Part B: PL/SQL IN A WRAP (slide 2 of 6)

- S: PL/SQL block Structure
  - S1: Declaration section (optional)(e.g. sl 79, 97)
    - S1.1: Data types and rules (e.g., 102-103;)
    - (Varchar2, char, Number, binary_integer, Date, BOOLEAN, Long or CLOB, Rowid, %TYPE, Exception, %ROWTYPE, CURSOR, Type Record, Type Table, and Bfile or BLOB.
    - S1.2: Substitution variable for reading from the keyboard (e.g., sl 84 - 92)
    - S1.3: Declaring Anchored Types (sl 94 - 95)
    - S1.4: Declaring Record Types:
      - S1.4.1. Cursors (sl 133 - 143)
      - S1.4.2. Using %ROWTYPE
      - S1.4.3. Using TYPE (like struct)

Part B: PL/SQL IN A WRAP (slide 3 of 6)

  - S1.5: Declaring Exceptions (sl 131-132; 144-153)
  - S1.6: Pl/SQL Table (arrays) (sl 204)
- S2. Executable Section
  - S2.1: SQL statements (sl 105 - 110)
  - S2.2: Printing instruction (sl 90-92)
    - DBMS_OUTPUT.PUTLINE(parameter);
  - S2.3: Assignment instructions (sl 102, 116-119)
  - S2.4: Decision instructions (sl 127-128; ..)
    - S2.4.1: IF-THEN-ENDIF statement
    - S2.4.2: IF-THEN-ELSE-ENDIF statement
    - S2.4.3: IF-THEN-ELSIF----ELSE-ENDIF statement

Part B: PL/SQL IN A WRAP (slide 4 of 6)

- S2.5: Repetition instructions (sl 33-143)
  - S2.5.1: LOOP……END LOOP; statement
  - S2.5.2: FOR loop_counter IN [REVERSE] lower_limit .. Upper_limit LOOP ……. END LOOP; statement
  - S2.5.3: CURSOR FOR LOOP statement
  - S2.5.4: FOR UPDATE CURSOR statement
  - S2.5.5: WHILE condition LOOP …. END LOOP;
- S2.6: Declaraing and Calling a function, procedure or package (sl 156 - 177)
- S2.7: Declaring and calling a trigger (sl. 183 – 197)

Part B: PL/SQL IN A WRAP (slide 5 of 6)

- (Note1: expressions are important parts of all these instructions and substitution variables can be used in expressions).
- Note2: A function, procedure, or package must be declared, compiled successfully into p-code and stored in the database server as database object to be called by other program units.

Part B: PL/SQL IN A WRAP (slide 6 of 6)

- **S3: Exception Handling Section (sl 131- 132, 144 - 153)**
  - **S3.1: Builtin exceptions**
    - **(VALUE_ERROR, NO_DATA_FOUND, TOO_MANY_ROW, ZERO_DIVIDE, LOGIN_DEFINED, PROGRAM_ERROR, DUP_VALUE_ON_INDEX)**
  - **S3.2: User Defined exceptions (e.g., sl 144)**
    - **These must be declared in the declaration part, condition to raise them specified in the executable section and action to take when they occur specified in the exception handling section.**

Oracle PL/SQL: Structure of a block

Structure of an anonymous PL/SQL block is:

```
DECLARE
   Declaration statements
BEGIN
   Executable statements
EXCEPTION
   Exception-handling statements
END;
```

PL/SQL: Declaration Section

- Declaration section is for definitions of PL/SQL identifiers (variables, constants, cursors, etc)
- E.g.,

```
DECLARE
        v_first_name    VARCHAR2(35);
        v_last_name     VARCHAR2(35);
        v_counter       NUMBER:=0;
```

- A semicolon ends each declaration
- A variable declaration has the format
  identifier-name identifier-type (size);
- A constant CONSTANT declaration has the format
- constant-name CONSTANT -type := initial value;

PL/SQL: Executable Section

- Executable section starts with BEGIN statement and ends with END statement as in:

```
BEGIN
        SELECT first_name, last_name
        INTO v_first_name, v_last_name
        FROM student
        WHERE student_id = 123;
        DBMS_OUTPUT.PUT_LINE
('Student name:' || v_first_name || ' ' || v_last_name);
END;
```

PL/SQL: Executable Section

- Above selects first and last names of student with id 123 from db student table into PL/SQL variables v_first_name and v_last_name so that they can be printed using DBMS_OUTPUT.PUT_LINE statement.
  - An example Exception handling section for the above block is:

```
EXCEPTION
    WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE
    ('There is no student with id 123');
```


PL/SQL: Reading Data with Substitution variables

- Reading Data with Substitution variables
  - PL/SQL cannot accept input from a user directly.
  - However, sqlplus enables PL/SQL to receive input information with substitution variables.
  - Substitution variables are usually prefixed by the ampersand (&) or double ampersand (&&) character.
  - Substitution variables cannot be used to output values since no memory is allocated for them

PL/SQL: Reading Data with Substitution variables

- **E.g., The following block prompts user for v_student_id (the substitution variable), which it stores as PL/SQL variable v_student_id. Then, it stores the first and last names of the student with this student id from student table in the database and displays the student names as output.**

PL/SQL: Reading Data with Substitution variables

```
DECLARE
    v_student_id NUMBER := &sv_studentid;
    v_first_name VARCHAR2(35);
    v_last_name  VARCHAR2(35);
BEGIN
    SELECT first_name, last_name
        INTO v_first_name, v_last_name
      FROM student
    WHERE student_id = v_student_id;
    DBMS_OUTPUT.PUTLINE
   ('Student Name: ' || v_first_name || ' ' || v_last_name);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUTLINE ( 'No such student');
END;
```

PL/SQL: Reading Data with Substitution variables

- When a single ampersand is used in a substitution variable, the user is prompted to enter a new value for each occurrence of the variable.
- E.g., on use of single substitution (&) variable

```
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Today is ' || ' &sv_day');
    DBMS_OUTPUT.PUT_LINE ('Tomorrow is '|| '&sv_day');
END;
```

- The above block produces the following output
  - Enter value for sv_day: Monday
- Old 2: DBMS_OUTPUT.PUT_LINE ('Today is' || '&sv_day');
- New 2: DBMS_OUTPUT.PUT_LINE ('Today is' || 'Monday');

PL/SQL: Reading Data with Substitution variables

  - Enter value for sv_day: Tuesday
- Old 3: DBMS_OUTPUT.PUT_LINE ('Tomorrow is' || '&sv_day');
- New 3: DBMS_OUTPUT.PUT_LINE ('Tomorrow is' || 'Tuesday');
- Today is Monday
- Tomorrow is Tuesday
- PL/SQL procedure successfully completed.
  - The program output contains statements showing how the substitution for the substitution variables are done. (e.g., statements beginning with old 2, new 2, old 3, new 3)

PL/SQL: Reading Data with Substitution variables

- To block the display of substitution statements, use the SET command option before running the script as in:

SET VERIFY OFF;

- This gives the output that excludes the 4 statements beginning with old and new.
- When we use a substitution variable that is preceded by a double (&&), PL/SQL processor prompts the user to enter the value of this variable once first time used. Then, it substitutes this value for other uses of this variable (which should be single (&)) in the block.

PL/SQL: Reading Data with Substitution variables

- E.g., Use of Double (&&) substitution variable.

```
BEGIN
   DBMS_OUTPUT.PUT_LINE ('Today is' || '&&sv_day');
   DBMS_OUTPUT.PUT_LINE ('Tomorrow is' || '&&sv_day');
END;
```

- Here, user is prompted only once and if entered day is 'Monday', both output lines use Monday and result is like:

Today is Monday

Tomorrow is Monday

PL/SQL procedure successfully completed.

PL/SQL: Reading Data with Substitution variables

- It is a good practice to enclose a substitution variable in single quotes if it is assigned to string (text) datatype as follows.
- E.g., Use of string substitution variable

  v_course_no VARCHAR2(5) := '&sv_course_no';
- Sqlplus allows changing the substitution variable character from (&) to a non-alphanumeric character specified using the following SET option

  SET DEFINE character

  SET DEFINE *
- To disable substitution variable feature, use:

  SET DEFINE OFF
- To enable substitution variable feature, use:

  SET DEFINE ON

PL/SQL: Displaying Output

- DISPLAYING OUTPUT with DBMS_OUTPUT.PUT_LINE
- The DBMS_OUTPUT.PUT_LINE is a call to procedure PUT_LINE in the DBMS_OUTPUT package of the Oracle user SYS
- This procedure DBMS_OUTPUT.PUT_LINE writes lines to buffer so that they can be displayed on the screen at the end of the program.
- The size of the buffer can be set to between 2000 and 1M bytes.
- Before output printed on the screen can be viewed, one of the following statements must be entered before the PL/SQL block.

  SET SERVEROUTPUT ON;

  or

  SET SERVEROUTPUT ON SIZE 5000;

PL/SQL: Displaying Output

- **Both statements enable the DBMS_OUTPUT_PUTLINE statements. And while the first statement uses default buffer size, the second uses buffer size of 5000 byte.**
- **To disable info from being displayed on the screen, use:**
  SET SERVEROUTPUT OFF;
- **E.g., PL/SQL code for Exercise 1 on page 48 for computing the area of a circle given the radius as substitution variable is next.**

PL/SQL: Displaying Output

- **Solution:**

```
DECLARE
   v_radius NUMBER := &sv_radius;
   v_area   NUMBER := v_radius * v_radius * 3014;
BEGIN
   DBMS_OUTPUT.PUT_LINE ( 'Area of Circle with
   radius' || v_radius || 'is'     || v_area);
END;
```

PL/SQL Programming Fundamentals

- PL/SQL Programming Fundamentals
- Character Types
  - PL/SQL engine accepts four types of characters (letters, digits, symbols (*, +, -, =, …) and white space.
  - Combinations of characters form one of the valid 5 lexical units (identifiers, reserved words, delimiters, literals, comments).
  - Identifiers begin with a letter and can be up to 30 characters long (avoid reserved words).
  - Reserved words like BEGIN, END etc are for use by PL/SQL
  - Delimiters are arithmetic, comparison and logical operators and quotation marks.

PL/SQL Programming Fundamentals

- Literals are values that are not identifiers, e.g., 150, 'Holiday', FALSE.
- Comments: lines beginning with (--) are single line comments while those lines between (/*) and (*/) are multiple line comments.

- Anchored Datatypes.
  - An anchored datatype is based on the datatype of a database object (like database attribute, e.g., student.firstname).
  - Giving a PL/SQL variable, an anchored datatype that is similar to the datatype of database attribute, Student.first_name can be done with the following instruction:

v_name student.first_name%TYPE;

  - General syntax for declaring variable of anchored type is:

<variable_name> <type attribute> % TYPE;

PL/SQL Programming Fundamentals

- **E.g.,**

```
DECLARE
   v_name student.first_name % TYPE;
   v_grade grade.grade_type_code % TYPE;
BEGIN
   DBMS_OUTPUT.PUT_LINE (NVL(v_name, 'No
   Name') || ' has grade of  ' ||  NVL(v_grade, ' no grade');
END;
```


PL/SQL Programming Fundamentals

- **DECLARING AND INITIALIZING VARIABLES**
  - **Each variable declared to be used by the program in the DECLARATION section should be terminated with a semicolon.**
  - **A numeric constant variable must be assigned a value with (:=) at declaration time and this value cannot be changed later in the program**
  - **A constant variable during declaration includes the keyword CONSTANT as in:**

```
v_cookies_calorie CONSTANT NUMBER := 300;
```

PL/SQL Programming Fundamentals

- Example declarations are:

```
DECLARE
   v_lname VARCHAR2(30)
   v_regdate DATE;
   v_pctincv CONSTANT NUMBER(4, 2) := 1.15;
   v_counter NUMBER := 0;
   v_new_cost course.crsecost % TYPE;
   v_yorn BOOLEAN := TRUE;
BEGIN
     NULL;
END;
```


EXPRESSIONS, OPERANDS AND OPERATORS

- An expression is a sequence of variables and literals, separated by operators, for performing calculatioins and comparing data.
- An expression is a combination of operands and operators.
- An operand is a variable, a constant or a function call.
- An operator is arithmetic (**, /, *, +, -), comparison (<, >, <>, =, >=, <=, !=, like, in, between, is null), logical (AND, OR, NOT), string (||, like)
- Parentheses can be used to enforce the order of execution of an expression.
- General operator precedence is
  - **, NOT,
  - +, -, arith identity and negation, *, /, +, -, ||, =, <>, <=, <, >, like, between, IN, IS NULL.
  - AND
  - OR

EXPRESSIONS, OPERANDS AND OPERATORS

- **E.g., expressions are:**

((v_counter + 5) * 2) / 2

(v_new_cost * v_counter) / 5

- **Expressions form the right sides of assignment instructions like:**

v_counter := ((v_counter +5) * 2) / 2;

v_new_cost := (v_new_cost * v_counter) / 4;

Use of Labels, Scope of Block & Variables

- **Use of Labels**
  - **Labels can be used for readability and label for a block must appear before the first line of executable code (BEGIN or DECLARE) as follows.**

```
<<find_stu_num>>
BEGIN
      DBMS_OUTPUT.PUT_LINE('procedure find_stu_num has been executed.');
END find_stu_num;
```

- **Scope of a Block & Variables**
  - **The scope or existence of variables defined in the declaration section of a block is the block.**
  - **A nested block is a block totally inside another block.**

Scope of Block & Variables; Common Data Types

- Visibility of a variable is the part of the program where this variable can be used or accessed.
- Scope of exception is also the block it is defined.

- Most Common Datatypes
- 1. VARCHAR2 (maximum_length): takes character variable specifying maximum length of up to 3276 bytes. Maximum width of a VARCHAR2 database column is 2000 bytes.
- 2. CHAR (maximum_length): stores fixed size character with specified MAX_length, that is possibly padded with blanks. Maximum length that can be specified is 32767 bytes although maximum length of a database column that can be stored with this type is 255 bytes. Default length is set to 1 if max_length is not specified.

Common Data Types

- 3. NUMBER [(precision, scale)]: stores fixed or floating-point number of any size where precision represents number of digits and scale determines number of digits following decimal point.
  - When scale is omitted, it represents integer number
  - Maximum precision is 38 decimal digits
  - A negative scale causes rounding to the left of the decimal point.
  - E.g., with the declarations
    v_num NUMBER (6, 2) := 3.456;
    v_num NUMBER (6, 3) := 3456;
    v_num has 3.46 and v_numl has 3000.
  - When scale is not specified, it defaults to 0 (rounding to nearest whole number).
- 4. BINARY INTEGER: stores signed integer variables in binary format for less space and more efficiency.

Common Data Types

- 5. DATE: stores fixed_length date values from January 1, 4712 BC to December 31, 4712 AD.
  - When stored in database column, date values include the time of day in seconds since midnight. The date portion defaults to midnight. Dates are displayed according to default format.
- 6. BOOLEAN: stores the values TRUE and FALSE and the non-value NULL. The values TRUE and FALSE cannot be inserted into a database column.
- 7. LONG: stores variable-length character strings of up to 32, 760 bytes, and can be inserted into a LONG database column, (which has a maximum width of 2, 147,483,647 bytes.
  - We cannot select a value longer than 32, 760 bytes from a LONG column into a LONG variable.
  - LONG columns can store text arrays of characters, or short documents, can be referenced in UPDATE, INSERT and (most) SELECT statements but not in expressions, SQL function calls, or certain SQL clauses such as WHERE, GROUP BY and CONNECT BY.
- 8. ROWID: stores rowids in a readable format. Internally, every Oracle database table has a ROWID pseudo column, which stores binary values called rowids.

Managing PL/SQL Code with SQL

- Managing PL/SQL Code with SQL
  - The changes to the database due to an application session are saved into the database after a COMMIT is executed.
  - Work within a transaction up to commit can be ROLLED BACK (that is undone).
  - A transaction is a series of SQL statements grouped together into a logical unit by the programmer.
  - A SAVEPOINT can be used to break down large SQL statements into individual units easier to manipulate.

Variable Initialization

- **Variable Initialization with SELECT INTO**
  - **In PL/SQL, variables can be assigned values in one of 2 ways:**
    - **During declaration with ':='**
    - **Assigning a value with SELECT INTO statement.**
- **SELECT INTO Statement: The Syntax of assignment with SELECT INTO is:**

```
SELECT item_name
INTO variable_name
FROM table_name;
```


Variable Initialization

- **E. g.,**

```
SET SERVEROUTPUT ON;
DECLARE
   v_average_cost VARCHAR2(10);
BEGIN
   SELECT  To_char (Avg(cost), '$9, 999. 99')
   INTO  v_average_cost
    FROM  course;
   DBMS_OUTPUT.PUT_LINE(' The average cost of a '||' course in
   the CTA program is '|| v_average_cost);
END;
```

Variable Initialization

- Variable v_average_cost is given the datatype VARCHAR2 because of the function used on the data.
- The TO_CHAR function formats the cost and the number datatype is converted to a character datatype.
- Another example of use of DML statement in PL/SQL block is:

```
DECLARE
      v_city zipcode.city % TYPE;
BEGIN
     SELECT 'COLUMBUS'
     INTO v_city
     FROM dual;
     UPDATE zipcode
            SET city = v_city
    WHERE zip = 43224;
END;
```


Inserting Data in PL/SQL

- DDL is not valid in a simple PL/SQL block.
- Data can be inserted as shown in the following example.

```
DECLARE
     v_zip zipcode.zip % TYPE;
     v_user zipcode.created_by % TYPE;
     v_date zipcode.created_date % TYPE;
BEGIN
    SELECT 43438, USER, SYSDATE
    INTO v_zip, v_user, v_date
     FROM dual;
    INSERT INTO zipcode
    (ZIP, CREATED_BY, CREATED_DATE, MODIFIED_BY,
    MODIFIED_DATE)
    VALUES (v_zip, v_user, v_date, v_user, v_date);
END;
```

Using an Oracle Sequence

- USING AN ORACLE SEQUENCE
  - An Oracle sequence is a database object used to generate unique numbers like primary keys.
  - Already created sequence values in SQL statements can be accessed with pseudo columns.
- CURRVAL (for returning the sequence current value)
- NEXTVAL (for incrementing the sequence and returning new value)
  - E.g., to create a sequence called ESEQ in sqlplus, we use:
- CREATE SEQUENCE eseq INCREMENT BY 10;
  - This sequence can be used to populate the column number attribute of a table called Teacher as follows:

Using an Oracle Sequence

```
CREATE SEQUENCE ESEQ
  INCREMENT BY 10;
CREATE TABLE TEACHER (col number);
BEGIN
      INSERT INTO TEACHER
       VALUES (ESEQ.NEXTVAL);
END;
```

Making Use of Savepoint

- Making Use of SavePoint
  - A transaction is a logical unit of work consisting of a set of SQL statements.
  - A transaction would either succeed (once a COMMIT is executed) or fail (if not successfully committed) as a unit.
  - The PL/SQL block for one transaction ends with COMMIT or ROLLBACK.
  - COMMIT makes events within a transaction permanent and releases all locks required by the transaction.
  - ROLLBACK erases (undoes) events within a transaction and releases all locks acquired by transaction.

Making Use of Savepoint

- SAVEPOINT can be used to control transaction such that SQL statements are split into transaction units that can be committed and rolled back as necessary.
- A COMMIT statement has the syntax:

COMMIT [WORK];

- The word WORK is optionally used for readability.
- A ROLLBACK statement has following syntax:

ROLLBACK [WORK];

- A SAVEPOINT command has the following syntax:

SAVEPOINT name;

The word name is the SAVEPOINT's name.

Making Use of Savepoint

- A program can be made to rollback to a SAVEPOINT using the more general form of ROLLBACK instruction below:
- ROLLBACK [WORK] to SAVEPOINT name;
- E.g., Page 81-82

```
BEGIN
    INSERT INTO student
        (student_id, Last_name, zip, registration_date, created_by,
        created_date,   modified_by, modified_date)
    VALUES (student_id_seq.nextval, 'Tashi', 10015, '01-JAN-99',
        'STUDENTA', '01-JAN-99', 'STUDENTA', '01-JAN-99'   );
    SAVEPOINT A;
```


Making Use of Savepoint

```
INSERT INTO student
  (student_id, Last_name, zip, registration_date, created_by, created_date
, modified_by, modified_date)
  VALUES (student_id_seq.nextval, 'Sonam', 10015, '01-JAN-99',
'STUDENTB',
 '01-JAN-99', 'STUDENTB', '01-JAN-99');
  SAVEPOINT B;
  INSERT INTO student
  (student_id, last_name, zip, registration_date, created_by, created_date
, modified_by, modified_date)
  VALUES (student_id_seq.nextval, 'Norbu', 10015, '01-JAN-99',
  'STUDENTA', '01-JAN-99', 'STUDENTB', '01-JAN-99');
  SAVEPOINT C;
   ROLLBACK TO B;
  END;
```

Making Use of Savepoint

- An example PL/SQL block that can contain multiple transactions

```
DECLARE
    v_counter NUMBER;
BEGIN
    v_counter := 0;
    FOR  i IN  1 .. 100
    LOOP
            v_counter := v_counter + 1;
            IF v_counter = 10
            THEN
                    COMMIT;
                    v_counter := 0;
            END IF;
    END LOOP;
END;
```

- Here, when v_counter hits 10, it commits keeping 10 transactions in one PL/SQL block.

Types of Instructions

- 1. Assignment Instructions
  - 1.1 Using assignment operator :=
    E.g., v_counter := ((v_counter + 5)) * 2)/2;
  - 1.2 Using SQL statements like:
    SELECT first_name, last_name
    INTO v_firstname, v_lastname
    FROM STUDENT
    WHERE stuid = v_stud_id;
- 2. Print and Read statements
  - 2.1 Print instructions with DBMS_OUTPUT.PUT_LINE as in:

Types of Instructions

- DBMS_OUTPUT.PUT_LINE('Area of Circle is' || v_area);
- 2.2. Read from the keyboard with substitution variables as in e.g.,
  NUMBER := &sv_radius;

- 3. Conditional Instructions (see slides 108 to 116 for IF statement examples)
  - 3.1 IF-THEN statement
  - 3.2 IF-THEN-ELSE statement
  - 3.3. IF-ELSIF. ….ELSE statement
  - 3.4 CASE statements: CASE form is given next.

Types of Instructions

- CASE condition
  WHEN expression 1 THEN statement 1;
  WHEN expression 2 THEN statement 2;
  ……
  WHEN expression N THEN statement N;
  ELSE statement N+1;
  END CASE;

Types of Instructions

- **4. Repetition Instructions**
  - **4.1 Simple Loop (LOOP …. END LOOP)**
  - **4.2 Numeric FOR LOOP (FOR loop_counter IN [REVERSE] lower_limit .. Upper_limit LOOP ….. END LOOP;]**
  - **4.3 Variations of FOR loop used for CURSOR (CURSOR FOR LOOP …)**
  - **4.4 WHILE condition LOOP (WHILE condition LOOP …. END LOOP)**

Conditional Control

- **Conditional Control**
- **IF Statements**
- **An IF-THEN statement has the following structure:**

```
IF CONDITION
    THEN
        STATEMENT 1;
        …
        STATEMENT N;
END IF;
```

Conditional Control

- If the CONDITION expression evaluates to TRUE, statements 1 to N are executed.
- E.g., write a PL/SQL block that compares two integer values in v_num1 and v_num2 and stores the smaller value always in v_num1.

```
DECLARE
      v_num1 NUMBER := 5;
      v_num2 NUMBER := 3;
      v_temp NUMBER;
BEGIN
 -- if v_num1 is greater than v_num2, then switch their values
         IF v_num1 > v_num2
```


Conditional Control

```
        THEN
          v_temp := v_num1;
          v_num1 := v_num2;
          v_num2 := v_temp;
        END IF;
          DBMS_OUTPUT.PUT_LINE('v_num1=' || v_num1);
          DBMS_OUTPUT.PUT_LINE('v_num2=' || v_num2);
END;
```

- The above produces the following output

  v_num1 = 3

  v_num2 = 5

- PL/SQL procedure successfully completed.

Conditional Control

- IF-THEN-ELSE STATEMENT
- The structure of the IF-THEN-ELSE statement is:

```
IF CONDITION
    THEN
      STATEMENT 1;
    ELSE
      STATEMENT 2;
END IF;
```

- STATEMENT 3;
- When CONDITOIN evaluates to TRUE, STATEMENT 1 is executed, but if it is FALSE, STATEMENT 2 is executed. The next statements in the program executed

Conditional Control

after the IF-THEN-ELSE statement is STATEMENT 3. E.g, Use of IF-THEN-ELSE statement is shown next.

```
DECLARE
        v_num NUMBER := &sv_user_num;
BEGIN
        -- test if provided number is even
IF MOD (v_num, 2) = 0
    THEN
        DBMS_OUTPUT.PUT_LINE (v_num || 'is even');
    ELSE
   DBMS_OUTPUT.PUT_LINE (v_num || 'is odd');
   END IF;
DBMS_OUTPUT.PUT_LINE ('Done');
END;
```

NULL Condition

- A NULL condition may arise if one of the compared variables has no value, for example:

```
DECLARE
   v_num1 NUMBER := 0
   v_num2 NUMBER;
BEGIN
  IF v_num1 = v_num2
  THEN
       DBMS_OUTPUT.PUT_LINE('They are equal');
  ELSE
       DBMS_OUTPUT.PUT_LINE('They are not equal');
  END IF;
END;
```


NULL Condition

- Note that v_num2 has no value leading to a NULL condition that evaluates to NULL and treated as false in this case.
- Use of Some Functions (Page 95), eg.
- TO_DATE, TO_CHAR, RTRIM

```
v_date DATE := TO_DATE('&sv_user_date', 'DD-MM-YY');
v_day := RTRIM(TO_CHAR(v_date, 'DAY');
```

- In the above instructions, the function TO_CHAR returns the day of the week with v_date padded with blanks since this function always returns 9 bytes.
- Next, the function RTRIM is used to remove trailing spaces.

ELSIF STATEMENT

ELSIF statements
An Elsif statement has the following structure

```
IF CONDITION 1
    THEN
   STATEMENT 1;
ELSIF CONDITION 2
    THEN
   STATEMENT2;
ELSIF CONDITION 3
    THEN
   STATEMENT 3;
…
ELSE
    STATEMENT N;
END IF;
```


Condition Control

- **Only one of statements 1 to N is executed depending on which of conditions 1 to N evaluates to TRUE. E.g.,**

```
DECLARE
    v_num NUMBER := &sv_num;
BEGIN
    IF v_num < 0
    THEN
       DBMS_OUTPUT.PUT_LINE ( v_num || 'is a negative number');
    ELSIF v_num = 0
  THEN
       DBMS_OUTPUT.PUT_LINE ( v_num || 'is equal to zero');
  ELSE
       DBMS_OUTPUT.PUT_LINE ( v_num || 'is a positive number');
    END IF;
END;
```

Exception Handling

- Exception Handling and Builtin Exception
  - Exception handling section in a PL/SQL block specifies what action to take when an exception error occurs.
  - Two types of exceptions exist – builtin and user-defined exceptions
  - Errors that occur in a program are either compilation or runtime errors. Exceptions are defined mostly for runtime errors.
  - Compilation errors are due to language syntax violation and are also called syntax errors.
- E.g., v_num1 = v_num1 / v_num2;

Exception Handling

- Will generate syntax error because assignment operator is (:=) and not (=). Statement should then be changed to following and re-compiled:
- v_num1 := v_num1 / v_num2;
  - Assume v_num1 has an initial value of 5 while v_num2 has a value of 0. Running this statement in a PL/SQL block leads to a runtime error because an illegal operation of dividing by zero has occurred.
  - Compilation and runtime errors may cause the program to not successfully complete.
  - Exception handling section is used to produce informative message when a runtime exception occurs. It also makes the program end successfully.

Exception Handling

- **Exceptions:**
  - **VALUE_ERROR: This is raised when there is a conversion or size mismatch error. Eg, v_num := SQRT(v_num1); if v_num1 has a negative value, the SQRT function cannot accept it, raising a VALUE_ERROR.**
- **Usage:**

```
EXCEPTION
    WHEN VALUE_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('Value Error Occurs');
```

- **NO_DATA_FOUND: raised when a select into statement, which makes no calls to group functions such as SUM or COUNT, does not return any rows. [Note that if the select makes a call to a group function like count, if nothing is found, it returns 0, and thus there is no need to raise a NO_DATA_FOUND exception in that case].**
- **TOO_MANY_ROWS: raised when a SELECT INTO statement returns more than one row [It normally should return only one row].**

Exception Handling

- **ZERO_DIVIDE: raised when a division by zero is performed.**
- **LOGIN_DEFINED: raised when a user is trying to log on to Oracle with invalid username and password.**
- **PROGRAM_ERROR: raised when the PL/SQL program has an internal problem.**
- **DUP_VALUE_ON_INDEX: raised when a program tries to store a duplicate value in the columns that have unique index defined on them. E.g., inserting values for course #, section # for course 60-415, section 1 that already exists and has a unique index defined on it.**

Cursors

- Example use of Cursor

```
DECLARE
v_sid student.student_id%TYPE
CURSOR c_student IS
SELECT student_id
FROM student
WHERE student_id < 110;
BEGIN
    OPEN c_student;
    LOOP
      FETCH c_student INTO v_sid;
      EXIT WHEN c_student % NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('STUDENT ID:' || v_sid);
```


Cursors

```
    END LOOP;
    CLOSE c_student;
    EXCEPTION
      WHEN OTHERS
      THEN
            IF c_student % ISOPEN
            THEN
                  CLOSE c_student;
            END IF;
END;
```

Cursors

- Using Cursor For LOOPS and Nesting Cursors
- Cursor FOR LOOP statement opens, fetches, and closes the cursor implicitly.
- The cursor FOR LOOP specifies a sequence of statements to be repeated once for each row returned by the cursor.
- Use the cursor FOR LOOP if you need to FETCH and PROCESS each and every record from a cursor.

Cursors

- For example, assume the existence of a table called table_log with one column.

```
DECLARE
    Cursor c_student IS
    SELECT student_id, last_name, first_name
    FROM student
    WHERE student_id < 110;
BEGIN
    FOR r_student IN c_student
    LOOP
      INSERT INTO table_log
      VALUES (r_student.last_name);
    END LOOP;
END;
```

Cursors

- Cursors can be nested inside each other
- Example nested cursor with a single child cursor.

```
DECLARE
  v_zip zipcode.zip % TYPE;
  CURSOR c_zip IS
        SELECT zip, city, state
        FROM zipcode
        WHERE state = 'CT';
  CURSOR c_student IS
        SELECT first_name, last_name
        FROM student
        WHERE zip = v_zip;
```


Cursors

```
BEGIN
        FOR r_zip IN c_zip
        LOOP
                v_zip := r_zip.zip;
                DBMS_OUTPUT.PUT_LINE(CHR(10));
                DBMS_OUTPUT.PUT_LINE('Students living in'
  ||r_zip.city);
                FOR r_student IN c_student
                LOOP
                DBMS_OUTPUT.PUT_LINE(r_student.first_name || ' ' ||
                        r_student.last_name);
                END LOOP;
        END LOOP;
  END;
```

Cursors

- **USING PARAMETERS WITH CURSORS AND FOR UPDATE CURSORS**
- **A cursor can be declared with parameters to enable it generate a more specific result set and make itself more reusable.**
- **E.g., create a cursor that works for only a set of values.**

```
CURSOR c_zip (p_state IN zipcode.state % TYPE)
IS
SELECT zip, city, state
FROM ZIPCODE
WHERE state = p_state;
```


Cursors

- **A cursor declared to take a parameter must be called with a value for that parameter.**
- **The c_zip cursor is called as follows:**

```
OPEN c_zip (parameter_value);
OPEN c_zip ('NY');
```

- **Using a FOR UPDATE CURSOR**
- **The cursor FOR UPDATE clause is only used with a cursor when you want to update tables in the database.**
- **This entails simply adding FOR UPDATE to the end of the cursor definition.**
- **Using the FOR UPDATE has the effect of locking the rows that have been identified in the active set.**

Cursors

- If we want to lock only one of multiple items being selected, add:
  FOR UPDATE OF <item_name>
- E.g.,

```
DECLARE
   CURSOR c_course IS
     SELECT course_no, cost
     FROM course FOR UPDATE;
BEGIN
   FOR r_course IN c_course
   LOOP
        IF r_course.cost < 2500
        THEN
```


Cursors

```
      UPDATE course
          SET crsecost = r_course.cost + 10
          WHERE course_no = r_course.course_no;
           END IF;
      END LOOP;
   END;
```

- WHERE CURRENT OF CLAUSE
- WHERE CURRENT OF <cursor_name> can be used to update the most recently fetched row as in:

Cursors

```
DECLARE
      v_zip zipcode.zip % TYPE;
      CURSOR c_student IS
            SELECT student_id, first_name, last_name, zip, phone
            FROM student
            FOR UPDATE;
BEGIN
            FOR r_stud_zip IN c_student
            LOOP
                  DBMS_OUTPUT.PUT_LINE(r_stud_zip.student_id);
                  UPDATE student
                  SET  phone = '718' || SUBSTR(phone, 4)
                  WHERE CURRENT OF c_student;
            END LOOP;
END;
```


User-Defined Exceptions

- User-Defined Exceptions
- Exceptions can be defined by programmer and must first be declared with the following syntax:

  DECLARE

  Exception_name EXCEPTION;
- The executable statements of a user declared exception are specified in the exception-handling section of the block.
- E.g., use of user defined exception

User-Defined Exceptions

```
DECLARE
   e_invalid_id EXCEPTION;
BEGIN
   WHEN e_invalid_id
   THEN
     DBMS_OUTPUT.PUT_LINE('A negative id is not allowed');
END;
```

- User defined exceptions have to be raised explicitly by defining what conditions should cause them to be triggered.
- How is given below:

User-Defined Exceptions

```
DECLARE
   Exception_name EXCEPTION;
BEGIN
……
   IF CONDITION
   THEN
     RAISE exception_name;
   ELSE
     ……
   END IF;
EXCEPTION
   WHEN exception_name
   THEN
     ERROR-PROCESSING STATEMENTS;
END;
```

User-Defined Exceptions

- Exception Propagation
- The rules governing how exceptions are raised in declaration and exception-handling sections are called Exception Propagation.
- When a runtime error occurs in the declaration or exception handling section, the exception handling section of this block is not able to catch it.
- In a program with nested PL/SQL blocks, when a runtime error occurs in the declaration section of the inner block, the exception immediately propagates to the enclosing outer block.

User-Defined Exceptions

- Exception: Advanced Concepts
- Raise_Application_Error
- Raise_Application_Error is used to assign an exception number and message to a user_defined exception.
- The syntax of the use of this procedure is:

  RAISE_APPLICATOIN_ERROR(error_number, error_message);
- or

  RAISE_APPLICATION_ERROR(error_number, error_message, keep_errors);
- E.g.,

User-Defined Exceptions

```
SET SERVEROUTPUT ON;
DECLARE
v_student_id  NUMBER := &sv_student_id;
v_total_courses NUMBER;
e_invalid_id EXCEPTION;

BEGIN
      IF v_student_id < 0
      THEN
            RAISE e_invalid_id;
      ELSE
            SELECT count(*)
            INTO v_total_courses
            FROM enrollment
```


User-Defined Exceptions

```
WHERE student_id = v_student_id;
            DBMS_OUTPUT.PUT_LINE('The student is registered
   for ' || v_total_courses || ' Courses');
      END IF;
END;
EXCEPTION
      WHEN e_invalid_id
      THEN
        DBMS_OUTPUT.PUT_LINE('The entered id is
   invalid');
END;
```

User-Defined Exceptions

- EXCEPTION_INIT_PRAGMA
- The EXCEPTION_INIT PRAGMA
- is used to associate an Oracle error number with a name of a user-defined error so that a handler may be written for it.
- The EXCEPTION_INIT pragma appears in the declaration section as:

```
DECLARE
    exception_name EXCEPTION;
        PRAGMA EXCEPTION_INIT(exception_name,
    error_code);
```

- The user_defined exception has to be declared before the EXCEPTION_INIT pragma that uses it.

User-Defined Exceptions

- SQLCODE and SQLERRM
- Oracle exception handler OTHERS can trap all Oracle errors.
- However, it is hard to know which error occurred if OTHER is used to trap it.
- Two built-in functions SQLCODE and SQLERRM can be used with the OTHERS exception handler to return the error number and message respectively.
- SQLERRM returns a message that is less than or equal to 512 bytes, while SQLCODE generally returns a negative error number.
- Example

User-Defined Exceptions

```
DECLARE
  v_zip VARCHAR2(5) := '&sv_zip';
  v_city VARCHAR2(15);
  v_state CHAR(2);
  v_err_code NUMBER;
  v_err_msg VARCHAR2(200);
BEGIN
        SELECT city, state
        INTO v_city, v_state
        FROM zipcode
        WHERE zip = v_zip;
        DBMS_OUTPUT.PUT_LINE(v_city || ' ' || v_state);
EXCEPTION
    WHEN OTHERS
    THEN
        v_err_code  := SQLCODE;
        v_err_msg := SUBSTR(SQLERRM, 1, 200);
        DBMS_OUTPUT.PUT_LINE('Error code: '|| v_err_code);
        DBMS_OUTPUT.PUT_LINE('Error message : '|| v_err_msg);
END;
```


Procedures

- SQLCODE, however returns a 0 if it is referenced outside the exception section, it returns +1 for user_defined exceptions and 100 for NO_DATA_FOUND exception.
- PROCEDURES
- Procedures allow structuring a program into modules (distinct subsolutions)
- Each module performs a specific task that contributes toward the final program goal.
- Modular code stored on database server is called a database object or subprogram that is available to other program units for repeated use.
- To save code into the database, it needs to be compiled into p-code and stored in database server.

Procedures

- A PL/SQL module is a complete logical unit of work and four types exist as:
- anonymous blocks
- procedures
- functions, and
- packages
- modular codes are more usable and manageable.

Procedures

- 1. ANONYMOUS BLOCKS
  - These have no names and no parameters.
  - Consists of Declaration, Execution and optional Exception parts.
  - They are not stored in the database as they cannot be called by other blocks. All examples before now are anonymous blocks.
- 2. PROCEDURES
  - A procedure may have 0 or more parameters and must have a name. The syntax of a procedure is:
- CREATE OR REPLACE PROCEDURE
  name [(parameter1, parameter2, …)]
  AS IS [local declarations]
  BEGIN
  Executable statements
  [EXCEPTION exception handlers]

  END [name];

Procedures

- A procedure consists of (1) the header [everything before the AS or IS keyword used interchangeably], (2) the body [everything after the AS or IS keyword].
- The word REPLACE is optional but if not used, changing procedure code will entail dropping and re-creating.
- E.g.,

Procedures

```
CREATE OR REPLACE PROCEDURE Discount
AS
   CURSOR c_group_discount
IS
   SELECT distinct s.course_no, c.description
   FROM section s, enrollment e, course c
   WHERE s.section_id = e.section_id
   AND c.course_no = s.course_no
   GROUP BY s.course_no, c.description, e.section_id,
   s.section_id
   HAVING COUNT(*) >= 8;
```

Procedures

```
BEGIN
    FOR r_group_discount IN c_group_discount
    LOOP
        UPDATE course
        SET cost = cost * .95
        WHERE course_no = r_group_discount.course_no;
        DBMS_OUTPUT.PUT_LINE ('A 5% discount has been
    given to' || r_group_discount.course_no || ' ' ||
r_group_discount.description);
    END LOOP;
END;
```


Procedures

- To have the procedure update the database, a COMMIT needs to be issued after running the procedure (after END). It can also be placed after the END LOOP statement.
- A procedure can become invalid when the table it is based on is deleted or changed.
- To re_compile an invalid procedure, use:
- ALTER procedure procedure_name compile;

Procedures

- PROCEDURES AND DATA DICTIONARY
- Data dictionary provides information on stored procedures in either
  - USER_OBJECTS view (information about objects), or
  - USER_SOURCE view (source code text)
- Data dictionary also has an ALL_ and DBA_ version of these views.
- PASSING PARAMETERS IN AND OUT OF PROCEDURES
- Parameters are used to pass values to and from calling procedures to the server.
- Parameters are available in 3 modes as IN, OUT, and INOUT.
- Parameter mode specifies whether it is:

Procedures

- IN: an input parameter that simply passes a value to the procedure for read only and this parameter cannot be changed by the procedure.
- OUT: an output parameter that passes result back from the procedure
- INOUT: both input and output parameter for passing value in and sending result back.

- Example Procedure with Parameters

```
CREATE OR REPLACE
    PROCEDURE FIND_NAME( ID IN NUMBER,  LNAME OUT
    VARCHAR2,
    FNAME OUT VARCHAR2)      AS
        BEGIN
            SELECT last_name, first_name
            INTO LNAME, FNAME
             FROM student
              WHERE student_id = ID;
```

Procedures

```
EXCEPTION
        WHEN OTHERS
        THEN
        DBMS_OUTPUT.PUT_LINE('
   Student id not found ');
END FIND_NAME;
```

- In the example, the parameters ID and LNAME, FNAME in the procedure header are formal parameters

- Formal parameters are place holders for actual data values passed in or out with actual parameters during procedure call.

Procedures

- Formal parameters do not require datatype constraints like size, e.g.,
- VARCHAR2(60) is stated as VARCHAR2.
- When matching actual and formal parameters, use positional notation or named notation.
- Named notation associates formal parameter to its actual value during procedure call explicitly using the format: (formal parameter => actual parameter).
- Calling a Stored Procedure
- The procedure find_name defined above can be called in another anonymous block

Procedures

- as:

```
DECLARE
      ID student.student_id%TYPE;
      v_local_fname student.first_name%TYPE;
      v_local_lname student.last_name%TYPE;
BEGIN
      ID := 250;
      find_name(ID, v_local_lname, v_local_fname);
DBMS_OUTPUT.PUT_LINE('Student ' || ID || ' is
   '||v_local_fname || '  ' || v_local_lname);
END;
```


Functions

- FUNCTIONS
- Function is a PL/SQL procedure that returns a single value.
- Function definition structure is:

```
CREATE [OR REPLACE] FUNCTION
     function_name (parameter list)
RETURN datatype
IS
BEGIN
     <body>
     RETURN (return_value);
END;
```

- In a function, there should be a RETURN statement for each exception
- Function parameters can be of IN, OUT or INOUT types.

Functions

- E.g.,

```
CREATE OR REPLACE FUNCTION
Show_description(i_course_no NUMBER)
   RETURN VARCHAR2
AS
   v_description VARCHAR2(50);
BEGIN
   SELECT description
   INTO v_description
   FROM course
   WHERE course_no = i_course_no;
   RETURN v_description;
```


Functions

```
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
     RETURN ('The cursor is not in the database');

   WHEN OTHERS
   THEN
     RETURN ('Error in running show_description');
END;
```

- The function declared above can be invoked in the SELECT statement below:

Packages

SELECT course_no, show_description(course_no)
FROM course;

PACKAGES

- **A collection of PL/SQL objects grouped together as a logical unit under one package name is called a package.**
- **Packages include procedures, functions, cursors, declarations, types and variables.**
- **First call to a package causes loading the package in memory, while subsequent calls save compilation and loading time.**
- **Packages encourage top down design and improve on information hiding and security of code.**
- **A package consists of Specification and Body, which may be compiled separately.**

Packages

- **Package Specification contains declaration information about objects in the package (procedures, functions and not their codes, global/public variables). All objects in a package specification are public objects.**
- **Private Procedures/Functions are not in the package specification but coded in its body.**

CREATE OR REPLACE PACKAGE
manage_students
AS

Packages

```
PROCEDURE FIND_NAME
( ID IN NUMBER,  LNAME OUT  VARCHAR2,
   FNAME OUT VARCHAR2);
FUNCTION id_is_good(i_student_id NUMBER)
RETURN BOOLEAN;
END manage_students;
```

- An example package specification consisting of a procedure and a function is given above.

Package Body

- Package Body
- The package body contains actual executable code of the objects described in the package specification
- Package body may contain additional code for private objects not declared in the specification of the package.
- The headers of the cursor and modules and their definitions in the package specification should match exactly.
- Elements declared in the specification can be referenced in the body and should not be re-declared.

Package Body

- Package elements can be referenced outside the package using the notation:
- package_name.element
- Elements referenced inside the body of the package do not need to be qualified.
- The package body of the above specification is:

```
CREATE OR REPLACE PACKAGE BODY manage_students
AS
PROCEDURE FIND_NAME
      (ID IN NUMBER,
      LNAME OUT student.last_name % TYPE,
      FNAME OUT student.first_name % TYPE)
IS
```


Package Body

```
BEGIN
   SELECT first_name, last_name
   INTO o_fname, o_lname
   FROM student
   WHERE student_id = ID;
   EXCEPTION
         WHEN OTHER
         THEN
               DBMS_OUTPUT.PUT_LINE('Error in finding
   student id:'|| ID);
END find_sname;
```

Package Body

```
    FUNCTION id_is_good
                ( i_student_id NUMBER)
    RETURN BOOLEAN
    IS
       v_id_cnt number;
    BEGIN
       SELECT COUNT(*)
       INTO v_id_cnt
       FROM student
       WHERE student_id = i_student_id;
       RETURN v_id_cnt=1;
    EXCEPTION
    WHEN OTHERS
    THEN
       RETURN FALSE;
    END id_is_good;
END manage_students;
```


Calling Stored Packages

- **CALLING STORED PACKAGES**
- **The following anonymous block shows how elements of manage_student package are called by other blocks.**

```
DECLARE
   v_first_name student.first_name % TYPE;
   v_last_name student.last_name % TYPE;
BEGIN
   IF manage_students.id_is_good (& v_id)
   THEN
     manage_students.find_sname
```

Calling Stored Packages

```
(&&v_id, v_first_name, v_last_name);
        DBMS_OUTPUT.PUT_LINE('Student No' || && v_id || 'is' ||
        v_last_name || ';' || v_first_name);
        ELSE
                DBMS_OUTPUT.PUT_LINE('Student ID' || &&
                v_id || 'is not in the database.');
        END IF;
END;
```

- Find out why actual parameter v_id is passed with & and &&
- Type the above code in a file and run the script in a sqlplus session
- The package body manage_students is compiled into the database.

Stored Code

- Functions in packages need to meet additional restrictions in order to be used in a SELECT statement (must be row functions and using only SQL datatypes, and have no DML(insert, update, delete), have certain level of purity achieved with PRAGMA RESTRICT_REFERENCES, p 332, 358-361, 366-368).
- Getting Stored Code Information from the Data Dictionary
- 1. DESC USER_ERRORS
  [used to determine details of a compilation error]
- 2. SHO ERR
  [displays the line number the error occurred in USER_SOURCE view]
- 3. DESC <packagename>
  To query the data dictionary to determine all stored objects in the current schema of the database including the current status of the stored code, use:

Stored Code

- SELECT OBJECT_TYPE, OBJECT_NAME, STATUS
  FROM USER_OBJECTS
  WHERE OBJECT_TYPE IN
  ('FUNCTION', 'PROCEDURE', 'PACKAGE',
  'PACKAGE_BODY')
  ORDER BY OBJECT_TYPE;
- 4. We can retrieve information from USER_ERRORS view with
  SELECT line || '/' || position "LINE/COL", TEXT "ERROR"
  FROM user_errors
  WHERE name = 'FORCE_ERROR';
- 5. DESC USER_DEPENDENCIES
  [used to analyze impact of table changes]
- 6. SELECT referenced_name
  FROM user_dependencies
  WHERE name = 'SCHOOL_API';
- The above lists all objects referenced in the package.

Stored Code

- 7. DEPTREE is an Oracle utility that shows which objects are dependent on a given object, but DBA access is needed to use this utility [see page 365 for details]
- 8. What is purity level of a function in a package?
  Purity level of a function describes the extent to which the function is free of side effects (altering public values also used by other functions)
- Available Purity levels are
  - WNDS (write no database state) or does not change any database tables
  - WNPS (write no package state) or does not alter any package variables
  - RNPS (reads no package state)
  - RNDS (reads no database state or table)

Stored Code

- To assert Purity Level, use
  PRAGMA RESTRICT_REFERENCES
  (function_name, WNDS[, WNPS][,RNDS][,RNPS]);
- 10. With the Purity level set as:

PRAGMA RESTRICT_REFERENCES (school_api, WNDS, WNPS);

- Inside the package specification, any update instruction will result in a purity level violation error.
- Only the WNDS level is mandatory and we need a separate pragma statement for each packaged function used in an SQL statement.
- The pragma must come after the function declaration in the package specification

Overloading Modules

- OVERLOADING MODULES
- When we overload modules, we give two or more modules the same name.
- The parameter lists of the modules should differ enough to have the versions distinguishable.
- Modules can be overloaded in the following 3 contexts.
- in a local module in the same PL/SQL block
- in a package specification
- in a package body.
- [see page 359-361]
- E.g., the following two procedures cannot be overloaded.
- PROCEDURE calc_total (reg_in IN CHAR);
- PROCEDURE calc_total (reg_in IN VARCHAR2);

Triggers

- **TRIGGERS**
- **A database trigger is a named PL/SQL block stored in a database and executed when a triggering event occurs.**
- **Executing a trigger is called firing a trigger.**
- **A triggering event is a DML (INSERT, UPDATE, or DELETE) statement executed against a database table.**
- **A trigger can fire before or after a triggering event**
- **For example, a trigger can be defined to fire before an INSERT statement on the STUDENT table and it fires each time before you insert a row in the STUDENT table.**

Triggers

- **The general syntax for creating a trigger is:**

```
CREATE [OR REPLACE] TRIGGER trigger_name {BEFORE |
    AFTER}
    Triggering_event ON table-name [FOR EACH ROW]
    [WHEN condition]
DECLARE
    Declaration statements
BEGIN
    Executable statements
EXCEPTION
    Exception-handling statements
END;
```

Triggers

- Dropping a table also drops all triggers on the table.
- Triggers can be used to enforce complex business rules not handled with integrity constraints.
- Maintaining security rules
- Automatically generating values for derived columns
- Collecting statistical information on table access.
- Preventing invalid transactions
- For auditing
- A trigger may not issue a COMMIT, SAVEPOINT or ROLLBACK statement.

Triggers

- Any function or procedure called by a trigger may not issue a transactional control statement (COMMIT, SAVEPOINT, ROLLBACK)
- Datatype LONG and LONG RAW cannot be used in a trigger, E.g.,

```
CREATE OR REPLACE TRIGGER student_bi
BEFORE INSERT ON student
FOR EACH ROW
DECLARE
```

Triggers

```
v_student_id STUDENT.STUDENT_ID % TYPE;
BEGIN
   SELECT STUDENT_ID_SEQ.NEXTVAL
   INTO v_student_id
   FROM dual;
   :NEW.student_id := v_student_id;
   :NEW.created_by := USER;
   :NEW.created_date := SYSDATE;
   :NEW.modified_by := USER;
   :NEW.modified_date := SYSDATE;
END;
```


Triggers

- The above trigger fires before each INSERT statement on the student table.
- The pseudo-record :NEW accesses a row currently being processed.
- The :NEW record is a type TRIGGERING_TABLE % TYPE and in this case, it is of type STUDENT % TYPE and members (attributes) of this record are accessed using the dot notation (eg, :NEW.student_id).
- Once the above trigger is used to populate the record with student_id, user and creation dates, the attributes left to insert values in this record would be last and first names, zip and registration date.
- Thus, the shorter version of INSERT used is to accomplish this is:

Triggers

- INSERT INTO student (first_name, last_name, zip, registration_date)
- VALUES ('John', 'Smith', 'OO914', SYSDATE);
- BEFORE triggers should be used
- When the trigger provides values for derived columns before an INSERT or UPDATE statement is completed.
- When the trigger determines whether an INSERT, UPDATE or DELETE statement should be allowed to complete. (E.g., determining if an inserted ZIP is valid)

Triggers

- AFTER TRIGGERS
- Example: the statistics table with structure statistics (Table_Name, Transaction_Name, Transaction_user, Transaction_Date);
- A trigger on the Instructor table, which fires after an UPDATE or INSERT statement is:

```
CREATE OR REPLACE TRIGGER instructor_aud
    BEFORE UPDATE OR DELETE ON INSTRUCTOR
    DECLARE
   v_type VARCHAR2(10);
    BEGIN
   IF UPDATING
   THEN
     v_type := 'UPDATE';
```

Triggers

```
    ELSEIF DELETING
    THEN
      v_type := 'DELETE';
    END IF;
    UPDATE statistics
    SET transaction_user = USER
          transaction_date = SYSDATE
    WHERE table_name = 'INSTRUCTOR'
          AND  transaction_name = v_type;
    IF SQL % NOTFOUND
    THEN
      INSERT INTO statistics
      VALUES (' INSTRUCTOR', v_type, USER, SYSDATE);
    END IF;
END;
```


Triggers

- Note that the functions UPDATING and DELETING are Boolean.
- This trigger updates or inserts a record in the statistics table when an UPDATE or DELETE operation against the instructor table occurs.
- Once trigger is created on the instructor table, any UPDATE or DELETE causes modification of old record or creating of new records, in the statistics.
- After triggers should be used when
- a trigger should be fired after a DML statement is executed.

Triggers

- When a trigger performs actions not specified in a BEFORE trigger.
- Consider the following UPDATE statement.

```
UPDATE student
SET zip = '01247'
WHERE zip = '02189';
```

- The value "01247" of the ZIP column is a new value and trigger would reference it as :NEW.ZIP. The value "02189" in the ZIP column is the previous value and is referenced as :OLD.ZIP.

Triggers

- :OLD is not defined for INSERT statements and :NEW is not defined for DELETE statements.
- These pseudo variables are referenced in the condition of a WHEN statement without : as in:

```
CREATE TRIGGER student_au
BEFORE UPDATE ON STUDENT
FOR EACH ROW
WHEN (NVL(NEW.ZIP, ' ') < > OLD.ZIP)
   Trigger Body ……
```

Types of Triggers

- TYPES OF TRIGGERS
- Row Triggers
- A row trigger is defined with a statement including FOR EACH ROW as in

```
CREATE OR REPLACE TRIGGER course_au
AFTER UPDATE ON COURSE
FOR EACH ROW
……
```

- A row trigger fires as many times as there are rows affected by the trigger.
- Statement trigger
- A statement trigger does not include FOR EACH ROW in its definition, E.g.,

```
CREATE OR REPLACE TRIGGER enrollment_ad
AFTER DELETE ON ENROLLMENT
……
```


Types of Triggers

The trigger fires once after a DELETE statement is issued against the enrollment table.

- Statement triggers are used for actions that do not depend on individual records.
- INSTEAD OF TRIGGERS
- An instead of trigger is a row trigger that is defined on views to fire instead of the DML statement.

Mutating Table Issues/Trigger Restrictions

- MUTATING TABLE ISSUES
- A mutating table is a table having a DML statement issued against it. For a trigger, it is the table on which this trigger is defined.
- A constraining table is a table read from, for a referential integrity constraint.
- TRIGGER SQL Statement Restrictions
- An SQL statement may not read or modify a mutating table.
- An SQL statement may not modify columns of constraining table having primary, foreign, or unique constraints defined on them.

PL/SQL Tables

- PL/SQL Tables
- PL/SQL tables are PL/SQL arrays and DML statements cannot be issued on them.
- PL/SQL tables exist in memory only and not in database.
- Declaration of PL/SQL table
- To declare PL/SQL table,
- Define the table structure using TYPE statement.
- Declare the actual table.
- E.g., declaration of PL/SQL table

PL/SQL Tables

```
DECLARE
   TYPE LnameType IS TABLE OF
--Table structure definition
   Student.last_name % TYPE
   INDEX BY BINARY_INTEGER;
   --Create the actual table
   Slname LnameType;
   Ilname LnameType;
BEGIN
     NULL;
     ……
END;
```


PL/SQL Tables

- Referencing and Modifying PL/SQL Table Rows
- A particular table row is referenced as:
- <table_name> (<index_value>)
- The datatype of the index value is compatible with BINARY_INTEGER datatype and we assign values to a row using the := operator.
- E.g.

```
SET SERVEROUTPUT ON
DECLARE
   CURSOR c_slname IS
   SELECT last_name, student_id, ROWNUM
   FROM student
   WHERE student_id < 110
   ORDER BY last_name;
```

PL/SQL Tables

```
TYPE type_lname_tab IS TABLE OF
        student.last_name % TYPE
        INDEX BY BINARY_INTEGER;
        tab_slname type_lname_tab;
        v_slname_counter NUMBER:=0;
 BEGIN
        FOR r_slname IN c_slname
        LOOP

         v_slname_counter := v_slname_counter + 1;

        tab_slname(v_slname_counter):=r_slname.last_name;
        END LOOP;
```


PL/SQL Tables

```
FOR i_slname IN 1..v_slname_counter
      LOOP
            DBMS_OUTPUT.PUT_LINE('Here is a
last     name:' || Tab_slname(i_slname));
       END LOOP;
END;
```

PL/SQL Attributes

- PL/SQL Table Attributes
- Attributes used to gain information on a PL/SQL table are:
- 1. DELETE – deletes rows in a table
- 2. EXISTS – returns TRUE if specified entry exists in table.
- 3. COUNT- returns number of rows in table.
- 4. FIRST – returns the index of the first row in table.
- 5. LAST – returns the index of the last row in table.
- 6. NEXT – returns the index of the next row in table.
- 7. PRIOR – returns index to previous row in table.

PL/SQL Attributes

- Syntax of Use of Table Attributes
- PL/SQL table attributes are used with the following syntax
- <table_name>. <attribute>
- E.g., with a table name t_student, we can assign the row count of this table to variable v_count as follows:
- v_count := t_student.count;
- t_student.delete deletes all rows from the t_student table.
- t_student.delete(15) deletes only the 15th row. Also t_student.exists(100) will work on the 100th row.
- Thus, for some attributes, the syntax involes specifying which rows as:
- <table_name>.<attribute> (<index number>[, <index number>])

Part C: Oracle Forms

- Software needed
- Oracle Developer 6.0 or higher (our case, Oracle 10g)
- Oracle Developer is Oracle's application development tool suite containing components like Oracle Forms
- Oracle 10g server: This is the Oracle's RDBMS [There is Oracle Personal Edition or Oracle Enterprise Edition]
- SQLPLUS
- Windows 2000 or higher or Unix
- Access to www.
- Book website :
- http://www.phptr.com/phptrinteractive/
- http://www.phptr.com/motivala

Oracle Forms

- <u>Knowledge or Background Requirements</u>
- - Should be familiar with:
- Relational Database Design
- SQL DDL and DML [for manipulating tables, constraints, sequences etc]
- PL/SQL procedures including [Local variables, conditional logic and cursors, etc.]
- May need to be able to configure Windows Registry so that Oracle Forms can properly locate all files you create.

Part C: ORACLE FORMS IN A WRAP (slide 1 of 13)

- Oracle Developer suite software has about 20 individual components including Oracle Forms and Reports.
- Oracle Forms topics necessary to master the design and implementation of a database application with nice graphical user interface are summarized into categories A to E with references to full discussions in the course notes slides as follows:
- A: Main Concepts: [Events, Triggers and Items], Oracle Forms Files (Running Forms), Master-Detail Forms
  - A1: Events, Triggers and Items: Forms applications are event driven because they respond to events.

    1. An Event is a user or system action

    Example user action is clicking a button and example system action is checking that an entered course id being registered for, exists.

Part C: ORACLE FORMS IN A WRAP (slide 2 of 13)

- 2. Triggers are code objects that respond (or fire) in response to events. (sl 275 - 292)
- Example WHEN-BUTTON-PRESSED trigger [the programmer writes PL/SQL code inide the trigger to say what to do when button is pressed, e.g., display message to indicate name of user.]
- Triggers are classified by:
  - (1)Name: When event triggers, On event triggers, Pre event triggers, Post event triggers and Key triggers.
  - (2)Function: Queries triggers, Validation triggers, transactional triggers, Key triggers.
- Triggers make use of available Forms built-ins (sl 275 - 292)
- Forms Built-ins fall into one of 3 categories (1. Get Built-ins, 2. Set Built-in and 3. Find Built-ins).

Part C: ORACLE FORMS IN A WRAP (slide 3 of 13)

- 3. ITEMS: Users interact with Forms application through items. (see section B of Forms in Wrap).
- Example items are push buttons, text items, display items, check boxes, radio buttons, list items.
- Items present information from database (data items) or are control (non-base table items).
- An example data items shows a student's name and an example control item gives the number of students enroled in a course section.

• A2: Oracle Forms Files (Compilation and Running)(sl 250-251)

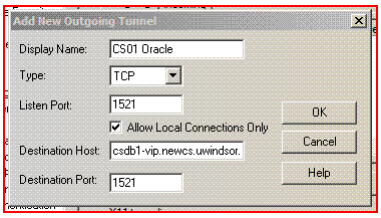
- Forms Builder creates files saved as .fmb and creates .fmx files after compilation.

Part C: ORACLE FORMS IN A WRAP (slide 5 of 13) : Running from Home PC

From Home PC

1. Create a Tunnel for the Oracle Listener through ssh client as:

- In SSH Secure Shell, select Edit ¨Settings from the pull down menu.
- Then select Tunneling from the list on the left
- Add new Outgoing Tunnel with dialog as shown below:



- The 'Destination Host' should be csdb1-vip.newcs.uwindsor.ca
- Click OK. Create an ssh connection to luna.cs.uwindsor.ca in the normal way and login before using the tunnel.

Part C: ORACLE FORMS IN A WRAP (slide 6 of 13) : Running from Home PC

- Now you can start the database client application on your PC (as detailed in step 2 below) and open a connection to local host port 1521 which will be automatically forwarded to the Oracle server in the School of Computer Science. you may have to modify the tnsnames.ora file on your PC as follows:

```
CS01 =
(DESCRIPTION =
        (ADDRESS_LIST =
        (ADDRESS = (PROTOCOL = TCP)(HOST localhost)(PORT = 1521))
        )
(CONNECT_DATA =
        (SERVICE_NAME = cs01)
        (INSTANCE_NAME = cs011)
)
)
```


Part C: ORACLE FORMS IN A WRAP (slide 7 of 13) : Running from Home PC

- 2. From Windows start Menu, Find Oracle Forms through Oracle Developer Suite/ Forms Developer / Start OC4J Instance.
  [Note: the OC4J Instance has to be running for your forms application to run well and when you are done, this instance can be shut down with Shutdown OC4J Instance.]
- 3. Now, launch the Forms Builder through Developer Suite/ Forms Developer/ Forms Builder.
- 4. When Done, shut down the OC4J through Developer suite/ Forms Developer / Shutdown OC4J Instance.

Part C: ORACLE FORMS IN A WRAP (slide 8 of 13)

- A3: Master-Detail Forms (sl 266-269)
  - Master-Detail forms are used to implement primary-foreign key relationship between two tables.
  - They are created with the data block wizard, where the primary key items are always in the master block, while the foreign key items are in the detail block.
- B. Mandatory Forms Objects (Physical or Logical)
  - B1: Physical Interface Objects are
    - (1) Items (e.g., buttons, text items) (sl 270-275)
    - (2)Canvases (items must be placed on a cavase to be seen) (sl 310-322).
    - (3) Windows (physical containers of canvases) (sl 310-322)

Part C: ORACLE FORMS IN A WRAP (slide 9 of 13)

- B2: Logical Interface Objects (do not have physical properties).
  - (4) Blocks (blocks are logical containers of items) (sl. 240-249, 255-259).
  - (5) Modules (Forms: logical containers of all objects) sl. 237-247).
- C. Objects Owned by a Form and Used by a Form (sl 216, 230-239)

Part C: ORACLE FORMS IN A WRAP (slide 10 of 13)

- 1.Forms
  - -ModuleName
  - 1.1 Triggers
  - 1.2Alerts
  - 1.3Attached Libraries
  - 1.4Data Blocks
    - -DataBlockName
    - 1.4.1 Triggers
    - 1.4.2 Items
    - 1.4.3 Relations
  - 1.5Canvases
    - -CanvasName
    - 1.5.1 Graphics
  - 1.6 Editors
  - 1.7 LOVs
  - 1.8 Object Graphics
  - 1.9 Parameters
  - 1.10 Popup Menus
  - 1.11 Program Units
  - 1.12 Property Classes
  - 1.13 Record Groups
  - 1.14 Reports
  - 1.15 Visual attributes
  - 1.16 Windows
- 2.Menus
- 3.PL/SQL Libraries
- 4. Object Libraries
- 5. Built-in Packages
- 6. Database Objects

Part C: ORACLE FORMS IN A WRAP (slide 11 of 13)

- **D. Tools for Creating and Manipulating Forms Objects**
  - **D1: Data Block and Layout Wizards (for creating a block and assigning its items to a canvas) (sl. 240-249)**
  - **D2: Three Main Development Tools**
    - **1. Object Navigator (sl. 252 – 257)**
    - **2. Property Palette (sl. 260 – 263)**
    - **3. The Layout Editor (sl. 264 – 265)**
  - **D3: Menu Editor (sl 369 – 377)**
    - **Used for building menus in Form Builder. First menus are built using Forms Builder and assigned to forms using menu module property.**

Part C: ORACLE FORMS IN A WRAP (slide 12 of 13)

- E: Other Forms Objects and Modules Used by Form (sl 270 – 272)
  - E1: LOVs (sl. 293 – 302) and ALERTS (sl 303 – 309)
  - E2: Visual Attributes and Property Classes (sl 306-308)
    - Visual attribute object contains all properties that determine font and color (sl. 328 – 3331).
    - Property classes contain all properties in a property class not just font and color.
  - E3: OBJECT GROUPS AND OBJECT LIBRARIES (sl 332-333)
    - An object group is an object within a Forms module, while an object library is a module unto itself.

Part C: ORACLE FORMS IN A WRAP (slide 13 of 13)

- E4: PROGRAM UNITS, PL/SQL Libraries (sl. 335-341)
  - Program units can be PL/SQL packages, procedures, or functions and can accept parameters and return values.
  - PL/SQL libraries are used for storing program units so that the program units can be used by many forms in an application. The library files are saved as .pll files and attached to needed forms.
- E5: Database Objects (sl. 342 – 345)
  - These are PL/SQL packages, procedures and functions stored in the database server. Forms modules can also call these PL/SQL stored objects as they do PL/SQL objects stored in the attached libraries.
- E6: Parameters and Built-in Packages (sl. 353 – 367)
  - Values can be passed to called forms through global variables or parameter list. The object navigator can be used to create parameters.

Part C: ORACLE FORMS IN A WRAP (slide 11 of 11)

- E7: Oracle Forms and Oracle Reports (sl. 358 – 367)
  - Oracle Report modules can be run by themselves using Report Runtime environment or called from form.
  - RUN_PRODUCT built-in or its later versions like RUN_REPORT_OBJECT can be used to call any of the three types of Oracle Developer modules Forms, reports, graphics.
- E8: MENUS (sl. 369 – 377)
  - Menus allow implementation of custom functionalities and security features for applications.

Windows Registry [Optional Part]

- The Windows Registry
- Windows Registry is a database maintained by Windows OS[95/98 and NT]
- Windows Registry contains configuration information about your computer, the hardware it uses and the software it runs.
- We are interested in knowing how Oracle Forms uses the Registry
- Oracle Forms and other Oracle Developer modules use the Registry to find where Oracle environment variables are stored.
- Other OS version (not Windows 95/98 and NT) may use other methods for managing environment variables.

Windows Registry [Optional Part]

- **Editing Windows Registry**
- **- To Edit Windows Registry with Register Editor, do:**
- **From Start Menu, select the RUN option to open the Run dialog window,**
- **type:**
- **regedit**
- **On the Registry Editor GUI, do:**
- **Expand the HKEY_LOCAL_MACHINE folder to see subfolders**
- **Expand the SOFTWARE node and scroll down to see folder ORACLE [do not expand] on the left pane.**
- **Scroll down on the right pane until you see FORMS60-PATH (a Forms environment variable containing a lot of directories used by FORMS)**

Windows Registry [Optional Part]

- **For Example:**
- **An application has an example that runs one form from another using the command:**
- **OPEN_FORM('STUDENT');**
- **This command refers to the STUDENT module name without specifying the directory path where this file STUDENT.fmx will be found. This is because FORMS would look for all such modules in the Registry director FORMS60_PATH.**
- **TO EDIT REGISTRY DO:**
- **Double-click the FORMS60_PATH to open the Edit String dialog.**
- **Using the cursor keys, navigate to the very front of the string. [Do not delete entries already there].**

Windows Registry [Optional Part]

- Type the following at the beginning of the string to enable Forms to reference modules stored in C:\guest\forms\exercises;
- [must include the semi-colon to separate entries]
- Click OK button.

Windows Registry [Optional Part]

- REPORTS60_PATH
- The REPORTS60_PATH Registry value is the environment variable that stores directory paths that Oracle Reports uses.
- We shall be calling Oracle Reports from Forms and will need to adjust the REPORTS60_PATH as well.
- To adjust REPORTS60_PATH, do:
  - Double-click the REPORTS60_PATH to open Edit String dialog
  - Repeat steps 2-4 above.
- <u>REGISTRY ADVANTAGES</u>
- Alternative to Registry is explicit specification of paths in applications developed to have for example:
- OPEN_FORM('c:\applications\forms\modules\student.fmx');
- This will force all users of the application to maintain this same directory structure.

Concepts and Objects

- Oracle Forms is part of a larger product called Oracle Developer with close to 20 individual components:- (Oracle Forms, Oracle Reports, Oracle Graphics etc) supported by subcomponents and utilities (e.g., project managers, debuggers, database schema builder etc)).

Events and Triggers

- Oracle Forms application are event-driven (application responds to events like user action or system action).
- User actions like clicking a button, tabbing from one item to another and opening or closing a window are called interface driven.
- Events drive Forms applications and the programmer can respond to every event that occurs with a piece of code.
- The code objects that respond to events are called triggers.
- One or more triggers may fire when an event occurs.
- E.g., to quit an application on windows, user would click on the close button at the top right corner of the window and this is an event.

ITEMS

- In response to the "Close_Window" event, Forms fires the WHEN_WINDOW_CLOSED trigger.
- The programmer writes code inside the trigger to tell the application what to do (e.g., flash a message reminding user to save work etc).

- ITEMS
- The Form interface consists of items.
- Example items are Buttons (push buttons), text fields (called text items or display items in Forms), check boxes and radio groups, list items.
- Items are used to present information from the database (base-table or data items) or to act as controls (non-base-table or control items).

ITEMS

- Most item types like display items are used as data or control items.
- E.g. a display item that presents information taken directly from the database, such as student's name or address is a data item because it is based on a column in the database.
- However, a display item to show the number of students enrolled in a certain section is a control item since its value must be calculated rather than retrieved from the database.
- Assume the database has a STUDENT table with an attribute for ZIP, there is also a ZIPCODE table that holds all valid zipcodes.
- You have a base item in form called ZIP for the column of table STUDENT.

EVENTS, TRIGGERS AND ITEMS WORKING TOGETHER

- When the user enters or changes values in the ZIP item in the form, we want the value validated by checking that it exists in the ZIPCODE table.
- To set up the form for this validation requires the following sequence of actions.
- User changes the ZIP items value
- User presses TAB key
- The above interface event causes a number of internal processing events, one of which is validate item event.
- The validate item event fires the WHEN_VALIDATE_ITEM trigger
- The code in the WHEN_VALIDATE_ITEM trigger validates the value in ZIP.
- This series of occurrences represents the essence of a Forms application.

Mandatory Forms Objects

- The five mandatory Forms objects are:
  - (1) items (2) canvases (3) windows (4) blocks and (5) modules
- - While items, canvases and windows are physical interface objects, blocks and modules are logical container objects.
- 1. ITEMS
  - Items are interface objects (buttons, text items, list items, radio group, check boxes) that allow Forms users to interact with Forms applications.
  - Items are defined by their properties including physical attributes such as Height, Width, X Position and Y position, etc; examples of data attributes are Column Name, Primary Key, Insert Allowed etc.

Mandatory Forms Objects

- The look, feel and behavior of an item are changed by adjusting properties.
- Properties are accessed through a window called Property Palette.
- Properties can be adjusted either during design or at run-time.
- At design, item properties can be changed by;
- (1). clicking the property palette and adjusting properties, or
- (2). Using the Layout editor, a graphical WYSIWYG tool that lets you position and size screen objects by dragging and dropping.
- To change properties at run time requires using code within the form, eg, using a trigger code to change background colour property of an item if the item's value is negative.

Canvases

- An item type can be determined in one of three ways:
- (i) By looking at the Item Type Property in the Property Palette.
- (ii) By looking at the item itself in the Layout Editor.
- (iii) By looking at the icon to the left of the items name in the object navigator.
  - There are 15 item types in Forms and the 6 most important types are display item, text item, radio group, list item, check box, push button..
- 2. CANVASES
  - Items need to be positioned on canvases to be seen by users. Thus, a Forms Canvas is the surface on which you position, size and color different objects.
  - The layout editor tool in Form Builder presents a WYSIWYG view of canvas and its items.

Canvases

- Layout, positioning, coloring and so on are done in the Layout Editor.
- Canvases like items have properties that can be viewed and changed through the property palette or at run-time.
- Non-mandatory graphical objects called frames are contained in canvases. A frame can hold a group of items and adjusting the frame properties would adjust properties of all its items.
- Fig 1.3, p13 has a canvas in the Layout Editor with two frames and items.

Windows/ Blocks

- 3. WINDOWS
  - Windows are physical containers of canvases and window's properties can also be changed at design or run-time.
  - Form's windows resemble Microsoft Windows and can be opened and closed manually or programmatically by the application.
- 4. BASE-TABLE BLOCKS AND NON-BASE TABLE BLOCKS
  - Canvases are physical containers of items, while blocks are logical containers of items.
  - Base-table blocks must contain at least one item that is based on a column in a database table or view.

Windows/ Blocks

- E.g. If we create a block based on the STUDENT table, at least one of its items must be based on a column in the STUDENT table.
- Not all items in a base-table block have to be based on columns in the table.
- E.g. Fig 1.4, P15 shows an INSTRUCTOR BLOCK which has most attributes in the INSTRUCTOR table, but includes additional non-table attributes like CITY and STATE.
- Blocks can also be based on an Oracle stored procedure (advanced topic).
- Non-Base-table blocks are not based on any database objects but contain non-base-table items like buttons.

Blocks

- Blocks are logical and do not have physical properties like X position, Y position, Height, Width etc.
- E.g. a Student Form has only one block that is based on the STUDENT table and includes all attributes of this table. The form has 2 windows (Student and Record History). Some of the items on the block are on a canvas in the student window, while the rest of the items are on a canvas in the Record History window.

Modules

- **MODULES**
  - **Modules are referred to simply as “forms”.**
  - **They are logical containers of all the objects in a form.**
  - **Fig 1.5, P16 of book shows physical objects contained within a forms module called COURSE.fmb.**
  - **A typical application is made up of a group of modules (tens or hundreds of modules).**

RELATIONSHIP BETWEEN MANDATORY FORMS OBJECTS

- **Blocks contain items; items can be positioned on canvases and windows contain canvases.**
- **Items can be logically contained in one block, but physically positioned across multiple canvases and windows.**

Objects owned or contained by a form and Instances

Object	Instance
1. Trigger	ON_CLEAR_DETAILS WHEN_NEW_FORM_INSTANCE
2. Alerts	DEMO_OBJECTS
3. Attached Libraries	-
4. Data Blocks	COURSE SECTION
5. Canvases	COURSE_SECTION
6. Editors	-
7. LOVS	-
8. Object Groups	-
9. Parameters	-
10. Popup Menus	-
11. Program Units	CHECK_PACKAGE_FAILURE CLEAR_ALL_MASTER_DETAIL
12. Property Classes	TABLE_ITEM_PROMPT_ALLIGMENT
13. Record Groups	-
14. Reports	-
15. Visual Attributes	-
16. Windows	COURSE_INFORMATION

The Data Block and Layout Wizards

- The Data block and layout wizards allow you to quickly create a block and assign its items to a canvas.
- The Data Block Wizard is first used to create a block, followed with creation of canvas and frame using the layout wizard.
- Each screen in a wizard is called a page.

The Data Block and Layout Wizards

- To use the Data Block Wizard do:
- Set the environment through the following steps.
  - Open the Form Builder.
  - From the main menu select File|New|Form to create a new form.
  - From the main menu, select File|Connect to connect to the database.
  - From the Main Menu, select Tools|Data Block Wizard to open the Data Block Wizard.

The Data Block and Layout Wizards

- The Data Block Wizard has three pages.

1. Type page
2. Table page
3. Finish page

- 1. Type Page
- -For selecting database object for block.
- - First choose base Table or View field or click the Browse button for the table name from a list.
  - To select columns, move them from the Available columns text list to the Database items text list.
    - This can be done by using the arrow buttons positioned between the two text lists. Or
    - Double-clicking individual items.
  - Multiple items can also be selected and moved.

The Data Block and Layout Wizards

- Although not all attributes need to be moved, selecting primary and foreign key attributes for the block is always encouraged.
- -Check Enforce data integrity for the block.
- -Click Next button to take to FINISH PAGE.
- FINISH PAGE
- -To move on to the Layout Wizard, select Create the Block and then call the Layout Wizard radio button and click Finish.
- THE LAYOUT WIZARD
- -The Layout Wizard has six pages:
  - 1. Canvas page
  - 2. Data Block page
  - 3. Items page
  - 4. Style page
  - 5. Records page
  - 6. Finish page

THE LAYOUT WIZARD

1. Canvas Page
   - You can make selections from the List items
   - Choose the Canvas [New canvas Existing Canvas Name]
   - Choose the canvas type[Content or Tab]
   - Choose tab page[Enabled when type is Tab & list shows]
   - -Next button takes the wizard to Data block page.
2. Data Block Page
   - -The tasks to perform here are:
   - (i) Select the items to be displayed [like in the data block wizard move items from the Available items list to the Displayed Item List].
   - (ii) Order the Items
   - [The Layout Wizard will lay the items onto the screen in the order they appear in the Displayed Items text list. To reorder the items, drag and drop to get in proper positions]

THE LAYOUT WIZARD

- (ii) Select the items type.
- [set the type for each item by selecting the items in the Displayed items list and changing its type using the Item type drop-down list].
- -Next button takes you to Items page.

3. Items Page
   - Two tasks performed here are:
   - Adjusting the prompt values.
   - Adjusting the Height and Width values.
   - This is done by positioning the cursor on the value we like to change and editing it.
   - -Next button takes it to style page.

THE LAYOUT WIZARD

4. Style Page
   - Two styles to choose from after laying out the items are:
   - Form style
   - Tabular style
   - -Next button advances it to Record page.

5. Records Page
   - -The four tasks to perform are:
   - (i) Choose a Frame Title.
   - (ii)Indicate the number of Records Displayed.
   - (iii)Set the Distance Between Records.
   - (iv)Include a scrollbar
   - -Next button advances it to finish page.

6. Finish Page
   - -click the Finish button, this causes the Layout Wizard to complete and canvas displayed in the Layout Editor.

SAVE YOUR FORM

- -From the main menu, select File|Save. Type a Test.fmb file.
- -You can now edit the form manually using the Layout Editor and Property Palette or the Wizards again by re-entering them.
- Name 6 ways to access the Data Block Wizard
  - From the Main Menu, select Tools|Data Block Wizard.
  - In the Object Navigator double-lick the Data Blocks node. (This will only work for the first block of each module).
  - In the Object Navigator, select the Data Blocks node and click the create button. The create button has a green plus sign as its icon.
  - In the object Navigator, select the Data Blocks node and right-click. Select Data Block Wizard.
  - In the Layout Editor, click the Data Block Wizard button. The Data Block Wizard button is situated in the middle of the Layout Editor's horizontal toolbar. It has a gray cylinder and a magic ward as its icon.
  - On the Form Builder welcome dialog in the section labeled Designing, select use the Data Block Wizard, and click the OK button.

ADDING NEW ITEMS TO BLOCK AND CANVAS

- ADDING NEW ITEMS TO BLOCK AND CANVAS
- It is important to have the proper block or one of its items selected in the object Navigator in order to re-enter the Data Block Wizard.
- You can add new items to a Block by re-entering the Data Block Wizard. However, this action will not add the items to the canvas or its frame.
- To add new items to the canvas, you need to reenter the Layout Wizard.

REMOVING ITEMS FROM BLOCK AND CANVAS

- -The Data Block does not allow you to remove items from a block.
- -Items can be removed simply by selecting them in the Object Navigator or
- Layout Editor and then deleting them manually by clicking the DELETE key.
- -Frames are graphical objects belonging to canvases that make it easier to control layout of multiple groups of items. When layout wizard positions items on canvas, it lays them out within frames.
- -For example in Master Detail Forms we can have forms that will allow us to have two groups of items on a single canvas. One of those groups is laid out in form style, while the other is laid out in tabular style. Each group of items will be in a frame. Having two frames lets you re-enter the layout wizard for each frame individually

Compiling and Running Oracle Forms Files

Oracle Forms Files

- -Form Builder creates binary files which have .fmb extension.
- -Compilation of .fmb files creates executable files with .fmx extension.
- - The .fmx files are run and delivered to users as executables.
- - while the binary .fmb files are platform independent as they can be edited and complied on Windows or Unix platform, the executable .fmx are platform dependent.

COMPILING

- -the .fmb files can be compiled into .fmx files in form builder.
- -To compile select Program|Compile Module
- -Compilation can also be done outside the Form Builder with a utility tool called Form Compiler that is installed with Form Builder.

Compiling and Running Form Files

- **RUNNING FORMS**
- **-Forms can be run directly in Form Builder or outside Form Builder with a utility tool called Forms Runtime.**
- **-To run, click Run Form Client|Server button. It has a traffic light icon.**
- **-If the form module compiles successfully, the Form compiler will simply close. You confirm the compilation by looking for the .fmx file in the file system.**
- **-If there are any errors during compilation, they will be displayed in a Forms Compilation Error window and written to a text file with .err extension.**
- **-A running form can be in one of three modes: Normal, Enter Query, or Fetch mode.**
- **-When a form first opens, the default behavior is for it to be in Normal mode, which means it is capable of accepting new records or updating existing ones.**

The Oracle Forms development environment

- **-In Enter Query mode, form is set to accept a query by example.**
- **-You can get out of Enter Query mode and back to Normal mode by clicking the Cancel Query button on the toolbar .**

- **The Development Environment**
- **-The Oracle Forms development environment has three main tools namely:**
- **(1) Object Navigator**
- **(2)Property Palette**
- **(3)Layout Editor**

The Object Navigator

1. Object Navigator

- -The object navigator presents hierarchical view of all the objects in a form.
- -It organizes these objects by node and lets us expand or collapse nodes.
- -Objects can be created, named, deleted, moved and manipulated within object navigator.
- -The object navigator vertical toolbar shows the object hierarchy as :

The Object Navigator

- 1.Forms
  - -ModuleName
  - 1.1 Triggers
  - 1.2Alerts
  - 1.3Attached Libraries
  - 1.4Data Blocks
    - -DataBlockName
    - 1.4.1 Triggers
    - 1.4.2 Items
    - 1.4.3 Relations
  - 1.5Canvases
    - -CanvasName
    - 1.5.1 Graphics
  - 1.6 Editors
  - 1.7 LOVs
  - 1.8 Object Graphics
  - 1.9 Parameters
  - 1.10 Popup Menus
  - 1.11 Program Units
  - 1.12 Property Classes
  - 1.13 Record Groups
  - 1.14 Reports
  - 1.15 Visual attributes
  - 1.16 Windows
- 2.Menus
- 3.PL/SQL Libraries
- 4. Object Libraries
- 5. Built-in Packages
- 6. Database Objects

Object Navigator with Nodes

- Object Navigator with Nodes
- -Here, Data Blocks and Canvases nodes are expanded to reveal their objects.
- -The highest nodes in the object hierarchy are Forms, Menu, PL/SQL Libraries, Object Libraries, Built-ins and Database Objects.
- -The Database Object node lets you view and edit database objects.
- -To open form do:
- Go to Main Menu and select File|Open
- -A small plus sign (+) to the left of a node like Alerts for form means that the form has this object. However, empty box beside a node (e.g LOV) means it does not have it.
- -There are three states for objects in the Navigator: deselected, selected and name-editable.
- -When you create objects, Forms gives them a default name (objects name + #)

To create a block manually

- -To create a block manually do:
- 1. Select the Data Blocks node in the Object Navigator.
- 2. Click the create button
- 3. Select Build a new data block manually and click the OK button.
- 4. Select the block created and name it CONTROL.
- -When a form is running, the user can navigate through the form by tabbing from item to item.
- -Forms default navigation order is based on how items are positioned in the object navigator.

The object Navigator's Data Block

- E.g. The object Navigator's Data Block for a form is:
- Data Block

? Zipcode

Triggers

- Items

Ô STATE

Ô CITY

Ô ZIP

Relations

The Layout Editor's Data Block

- However, in the Layout Editor, the order of the State, City and Zip items are different as [Zip, City, State].
- -This means that when the form is run and the user tabs from text item to text items, navigation will be in the order in objects navigation which is STATE-CITY-ZIP.
- -You can control the navigation order by dragging items and blocks up and down in the object navigation by writing triggers.
- -Items created in the object navigator are not visible until their canvas property is set in the property palette including their X and Y positions on the canvas.
- By default, the object navigator creates new items as text items, but to set to any other type, we can change the properties accordingly.

Object Navigator

- Whenever we paste or create objects in the object navigator, they are positioned directly below the object that is currently selected.
- -If you have multiple forms open at one time, it is essential to get proper focus on a specific form you want to work with (i.e. you want to run, close, compile, save etc).
- -To put the focus of the Form Builder on a certain form in the Object Navigator, select any object within that forms module.
- -Stored Program Units, PL/SQL Libraries Tables, Views and Types are objects that are visible to the programmer.
- -Tables views and columns are “read only”. However, the Database Objects Node is a handy way to examine the contents of database tables and views. It shows table, view and column names as well as data types and lengths.

Editing PL/SQL object through the Form Builder

- You can create and edit stored PL/SQL object through the Form Builder by selecting the Trigger node of the object and clicking the object navigator’s create button. This opens the database trigger editor and you can set the trigger type and write its code.
- -We can also write PL/SQL stored procedure packages and functions through the Form Builder.
- -Be careful not to drop needed PL/SQL objects from the database through Form Builder (?).
- The Property Palette
- -The look, feel and behavior of an object can be defined by its properties through the property palette.
- -The properties displayed in the property palette are shown as:

The Property Palette

The Property Palette	
1. General Name	STUDENT_ID
2. Item Type	Text Item
Subclass Information	
Comments Help Block Topic	
3. Functional	
Enabled	Yes
Justification	Start
Implementation class	
Multi-Line	No
Wrap Style	Word
Case Restriction	Mixed
Conceal Data	No
Keep Cursor Position	No
Automatic Skip	No
Popup Menu	<Null>
Navigation	
Keyboard Navigatable	Yes
Previous Navigation Items	<Null>
Next Navigation Item	<Null>
Data	
Data Type	Number
Maximum Length	9
Fixed Length	No
Item Type	

The Property Palette

- Any changes made to the object's property through the property palette are visible immediately on the layout editor. Also, object properties changed graphically on the layout editor are changed as well on their property palette.
- - Properties of a selected group of objects can also be change at once.
- Properties.
- -Item's Bevel property changes the appearance of the item's border.
- -The help system has a lot of details for explaining property restrictions for using a property programmatically.
- -When you changed properties of an item, the small icons to the left of the property names change from small dots to small green squares.
- - Not all properties are available for mass changes (e.g. Name and Subclass Property can not be mass changed).

ACCESSING THE PROPERTY PALETTE FROM THE FORM BUILDER

- **FIVE WAYS TO ACCESS THE PROPERTY PALETTE FROM THE FORM BUILDER**
- **From the Main Menu, select Tools|Property Palette.**
- **In the object navigator, select the objects whose properties you want to see and right click.**
- **In the object navigator, double-click on the icon on the left of the object whose properties you would like to see (this does not work for canvasses).**
- **In the Layout editor, select the object whose properties you would want to see.**
- **Press F4.**

The Layout Editor

- **The Layout Editor**
- **-The Layout editor allows you to visually position, arrange, size and color objects.**
- **While all objects (logical or physical) can be created in the object navigator you can only create physical objects on the layout editor that can appear on a canvas.**
- **-Physical objects that can appear on a canvas include items, other canvases and graphics.**
- **-Graphics include frames and any other non-item objects like rectangles, circles, lines and static text.**
- **-The layout editor has three toolbars that provide utility, formatting and create functions. They are called respectively utility, formatting and Vertical Toolbars [Fig3.7, P94].**
- **1. Utility Toolbar: allows you to open save, and run forms as well as cut and paste etc. It also allows you to coordinate canvases and block being worked on and gain access to the wizards.**

The Layout Editor

- 2. Formatting Toolbars: For formatting, positioning and arranging text.
- 3. Vertical Toolbar or tool Palette: Lets you select, rotate and reshape objects, create graphic objects, create items and color objects.
- For graphical objects such as items, stacked canvases and frames, it is common to use the Layout Editor for creation in order to position and size the object.
- -The upper left-hand corner of an object indicates its position. If the X position and Y position properties are set to 10,10 the object's upper left-hand corner is at the coordinates 10,10. This is true for items, frames, canvases, windows and other graphical objects.

Master – Detail Forms

- Master – Detail Forms
- -Master-Detail Forms allow us to create a form that is based on multiple base table blocks and establish a relationship between them.
-
- -The relationship allows us to issue a query in the master block, which causes the form to issue a corresponding query in the detail block.
- -The master and detail blocks are created using wizards. This automatically creates an object called a relation.
- -To change the behavior of the form, we adjust the properties of the relation object.
- -The relation object hold the join condition that relates the master and detail blocks. The relation object also has some triggers and program units written by Form Builder associated with them.

Master-Detail Forms

- -Master-Detail Forms is used to implement relationships like primary-foreign key relationship that may exist between two database tables.
- -When creating the first block in a form the wizard prompts you with master-detail pages and asks if there is already another block in the form.
- -When you create the detail block, the wizard asks if you want to create a relation object for establishing the join condition between two blocks and managing the coordination of their records.
- -Checking the Auto-join data blocks causes the wizard to write the join condition.
- -If Auto-Join data blocks is not checked, then you would use the Detail Item and Master Item poplists to create the join condition yourself.

Master-Detail Forms

- To create the join condition yourself, you would select the items that make up the logical joins and the wizard would use those items to write the relation objects join condition.
- -The Master block always contains the relation object even though the wizards create the relation object during the creation of the detail block.
- -When the Form Builder issues a query in the Master block, it needs to return corresponding rows to the detail block. This ensures that querying a record in the master block, brings back corresponding records to the detail block.
- -E.g. STUDENT and ENROLLMENT

  tables have a primary –foreign key relationship between them that tells for each student record the 0 to many enrollment records that exist.

Master-Detail Forms

- -The STUDENT items make the master block laid out in form style, while the ENROLLMENT items are in the detail block laid out in Tabular style. E.g. Fig 4.1, P109
- -A master-detail form is used to establish and display a one-to-many (primary-key-to-foreign key) relationship between blocks.
- -The primary key items are always in the master block and the foreign key items are always in the detail block.

ITEMS

- ITEMS
- 1. Text Items
- -are mostly, database items based on database table columns. These can be created with data block wizard. There are also non-database text items.
- 2. Display Items
- -Display database or non-database items not directly based on database columns but results of calculations or values of other tables. E.g. display items are time, date, database name etc.
- -For an item to be visible at runtime, it should be assigned to a canvas that is visible in a window.
- -When the canvas property of an item is set to Null, it is a null-canvas item (used as variables that can be referenced by PL/SQL).
- -Null canvas items properties cannot be configured and they are not visible in the Layout Editor.

ITEMS

- The Visibility property of an Item defaults to Yes but can be set to No to keep the item from not being visible when the form runs although it is always visible through the layout editor.
- -Having an item's Enabled Property set to No will allow values of the items to be displayed but prevent editing or updating those values.
- -It is wise to adjust the item's Data Type property to match the data type of its base column.
- -Forms can make use of database sequences by setting the initial value property of an Item with the following syntax:

  :SEQUENCE.sequence_name.NEXTVAL

Items

- -Format mask lets you display information in a format different from that stored in the database.
- -The Database Item property informs that this item is based on a column in the database. pp140-141.
- e.g. formula written by programmer for a display item (seats-left) after setting calculating mode to formula and writing the following expressions in the formula property:

  :SECTION.CAPACITY - :SECTION.NO_OF_ENROLL
- -The default value of this property is set by the Number of Records Displayed Property. The Query Array Size property of the block decides how many records are fetched from the database at a time.

Buttons, Lists, Items, Radio Groups and Check Boxes

- **Setting the Query All Records property to Yes overrides the Query Array Size property and is useful for computing averages correctly.**
- **3. Buttons, Lists, Items, Radio Groups and Check Boxes**
- **-Other Forms items in addition to text and display items are:**
- **(i)Images, sounds, Active X Controls, and**
- **(ii)Buttons, List items, radio groups, check boxes, etc.**
- **-The list of items in (ii) with text and display items are the most common item types.**

Buttons, Lists, Items, Radio Groups and Check Boxes

Buttons

- **-Creating and positioning buttons in Forms is easy, what is challenging is writing the code that goes behind the button.**
- **-Each button has a WHEN-BUTTON-PRESSED trigger associated with it so that it can respond to the Button Pressed event.**
- **-You can use the help system to locate Forms built-in to place behind your buttons.**

LIST ITEMS, RADIO GROUPS AND CHECK BOXES

- **-These items present the user with a number of choices.**
- **LIST ITEMS**
- **-These can serve as either database or non-database items.**
- **RADIO GROUPS**
- **-These are small circles and selection of one radio button deselects the previously selected button in the radio group items.**

Buttons, Lists, Items, Radio Groups and Check Boxes

- **CHECK BOXES**
- **-Theses are useful for storing Yes/No, True/False, and On/Off-type values.**
- **-The label property positions text on the button itself. Button also has prompt property that positions text somewhere next to the items.**
- **-Always, use labels for buttons.**
- **-Every list element must have a corresponding list item value.**
- **-The list elements dialog boxes lets you set both of these values.**
- **BUTTON TRIGGERS & BUILT-INS**
- **-The triggers to create for corresponding to the event of a user clicking a button is:**
- **WHEN-BUTTON-PRESSED trigger**
- **-The built-in to be used to exit the form is:**
- **EXIT_FORM built-in.**

Triggers

- **-The built-in for saving changes to database is:**
- **COMMIT_FORM built-in**
- **-The WHEN_CHECKBOX_CHANGED trigger is available for check box.**
- **TRIGGERS**
- **-A trigger contains PL/SQL code that responds to Forms events.**
- **-Oracle Forms and Oracle database both use PL/SQL programming language to store and retrieve data and present them in a GUI interface.**
- **-Triggers are always attached to other objects and have the same scope as their objects.**
- **-Triggers can be attached to items, block, or forms. WHEN_BUTTON_PRESSED trigger attached to a button item has item scope as it fires only when this item is accessed. The same trigger if placed on a CONTROL block with a set of buttons, will have a block scope if the trigger is set at block level.**
- **CATEGORIES OF TRIGGERS**
- **-Forms help system categorize triggers either (1) by name or (2) by function.**

Categorizing Triggers by Name

- -There are 5 named trigger categories. The first word in a trigger's name tells how it will affect Forms default processing and when it will fire relative to Forms default processing.
- When event triggers which augment Forms Processing
- On event triggers, which replace default processing
- Pre event triggers, which fire just before a When event or On event
- Post event triggers, which fire just after a When event or an On event
- Key triggers, which fire when a user presses a certain key.
- -For example, a trigger to the commit Transaction event which fires each time a form inserts a record could be the ON_INSERT, PRE_INSERT or POST_INSERT triggers depending on whether we want to override Forms default action, own logic before or after default action.

Categorizing Triggers By Function

- -Triggers can be categorized by the functions to which they are related.
- -A WHEN_BUTTON_PRESSED trigger is an interface event triggers. ON_INSERT and PRE_INSERT TRIGGERS ARE Transactional triggers.
- -Some Trigger categories are:
- 1. Query triggers which respond to events regarding queries
- 2. Validation triggers, which respond to events regarding the validation of items and records.
- 3. Transactional triggers, which respond to events regarding inserting, updating and committing of records.
- 4. Key triggers, which respond to key press events.
- -Each trigger falls into both a named and a functional category.

Triggers

- **The WHEN_VALIDATE_ITEM code body would look like: [Note that there is also a DECLARE section]**

```
BEGIN
   SELECT city, state
   INTO :STUDENT.CITY, :STUDENT.STATE
   FROM Zipcode
   WHERE zip =:STUDENT.ZIP;
   EXECUTION
     WHEN NO_DATA_FOUND THEN
     MESSAGE ('Zipcode does not exist in Zipcode table');
     RAISE FORM_TRIGGER_FAILURE;
End;
```


Triggers

- **Whenever we want to refer to an item and its block in an SQL statement in a trigger, we use the following syntax:**

  :block.item

- **-The STUDENT.EXIT button's WHEN_BUTTON_PRESSED trigger code is:**

  EXIT_FORM;

- **This is a built-in and even though there are no BEGIN or END statements listed, it is PL/SQL.**
- **If there is nothing to declare in the DECLARE statement, it is not mandatory that you include a BEGIN and an END STATEMENT.**

Triggers

1. Query Triggers.
- These triggers are used to make one or more of the base-table items more meaningful e.g using a trigger to fetch course description that is displayed with section id in the section block.

2 .Validation Triggers.
- –Two Validation triggers commonly used in Forms are WHEN_VALIDATE_ITEM and WHEN_VALIDATE_RECORD.
- -An example trigger CODE from WHEN_VALIDATE_ITEM for an application where section cost is never more than 15,000 is :

```
If SECTION.COST > 5000 then
        MESSAGE ('Course cost must be less than $5,000');
        RAISE FORM_TRIGGER_FAILURE;
END IF;
```


Triggers

- 3. Transactional Triggers
- –These are used to augment or replace Forms default transaction processing.
- -E.g. are ON_POPULATE_DETAILS and _ON_CHECK_DELETE_MASTER, PRE_CHANGE, POST_FORMS_COMMIT, POST_DATABASE_COMMIT, etc.
- 4. KEY TRIGGERS
- -These fire whenever a user presses a corresponding key on the key-board. E.g KEY_DOWN trigger. Key trigger can be used if you want to change or replace default key processing.
- -Another key trigger is KEY_DELREC written whenever primary-foreign key constraints exist in the database that correspond to one of the items in the block.
- -E.g. use of a trigger key.

```
DO_KEY('EXIT_FORM');
```

- -The code for PRE_INSERT triggers is:

Triggers

```
DECLARE
    v-block VARCHAR(30);
    v_username VARCHAR(30);
    v_date date;
BEGIN
    v_username:=GET_APPLICATION_PROPERTY(USERNAME);
    v_date:= SYSDATE;
    v_block:= :SYSTEM.CURSOR_BLOCK;
COPY(v_date,v_block||'.CREATED_DATE');
COPY(v_username,v_block||'.CREATED_BY');
COPY(v_date,v_block||'.MODIFIED_DATE');
COPY(v_username,v_block||'.MODIFIED_BY');
END;
```

- -PL/SQL editor is used for writing and debugging codes.

Triggers

- If there is a mistake in the code, the PL/SQL editor provides a gray area below the trigger code that lists error messages.
- -A trigger can issue the command RAISE FORM_TRIGGER_FAILURE if data value is invalid.
- -FORM_TRIGGER_FAILURE is a predefined, built-in Forms exception, which is used to halt forms processing when an error occurs.
- -FORM_TRIGGER_FAILURE can be used by any Forms PL/SQL object but not a database PL/SQL object.
- -GET_APPLICATION_PROPERTY built-in can be used to get user's name.

Triggers

- -System variables hold internal information about the form. SYSTEM.CURSOR_BLOCK holds the value of the current navigation block.
- -The COPY built-in copies a value to somewhere else.
- -In the above code the value being passed to the copy built-in is a value and a block.item name.
- -The code for a PRE-UPDATE trigger is:

Triggers

```
DECLARE
   v_block VARCHAR2(30);
   v-username VARCAHR2(30);
   v_date DATE;
BEGIN
   v_username:=GET_APPLICATION_PROPERTY(USERNA
   ME);
   v_date:=SYSDATE;
   v_block:=SYSTEM.CURSOR_BLOCK;
   COPY(v_date,v_block ||'.MODIFIED_DATE');
   COPY(v_username,v_block||'.MODIFIED_BY');
End;
```

Triggers/Built_ins

- -DO_KEY('EXECUTE_QUERY');
  This DO_KEY built-in fires the key trigger associated with the built-in, which is KEY_EXEQRY trigger.
- -DO_KEY('COMMIT_FORM');
  will fire the KEY_COMMIT trigger.
- -DO_KEY('ENTER_QUERY');
  will fire KEY_ENTQRY trigger
- FORMS BUILT-INS
- -Forms Built-ins are set of PL/SQL functions and procedures that perform standard application functions. E.g. EXIT_FORM and COMMIT_FORM.
- -Use of Built-ins without parameters
  EXIT_FORM;
- EXIT_FORM Built-in with parameter to specify what the form should do when it exists is:

Built_ins

EXIT_FORM('DO_COMMIT");

- -There are hundreds of built-ins in Oracle Forms and a comprehensive list and their functions and uses are provided by the Forms online help system.
- 1.GET_BUILT_INS
- -A number of built-in are defined with the word "GET_"An example is GET_APPLICATION_PROPERTY. It can be used as:
  :COURSE.CREATED_BY:=GET_APPLICATION_PROPERTY(USERNAME);
- -Other GET_built-ins exit for getting properties of Forms objects like items, blocks, canvases and so on.

Built_ins

- 2. SET-BUILT_INS
- -These are built-ins prefixed with the word SET_. Eg.
- SET_BLOCK_PROPERTY can be used to set properties of a block at run time.
- -The following two statements set the ORDER BY and the WHERE clauses for a block called SECTION.
- SET_BLOCK_PROPERTY('SECTION',DEFAULT_WHERE,'INSTRUCTOR_ID=101');
- SET_BLOCK_PROPERTY('SECTION',ORDER_BY,'SECTION_ID');
- -There are SET_built-ins for other objects in Forms like windows, items, canvases etc.

Built_ins

- 3. FIND-BUILT-INS
- -E.g. FIND_BLOCK for getting the Block id. To get an object's id, we use one of the FIND_built-ins. -The code involving use of FIND_built-in is.

```
DECLARE
    v_block_id BLOCK;
BEGIN
    v_block_id:=FIND_BLOCK('SECTION');
SET_BLOCK_PROPERTY(v_block_id,DEFAULT_WHERE,'INSTRU
    CT R_ID=101');
SET_BLOCK_PROPRETY(v_block_id, ORDER_BY,'SECTION_ID');
END;
```

- -Other GET_BUILT_INS are:

GET_ITEM_PROPERTY,GET_CANVAS_PROPERTY, and GET_BLOCK_PROPERTY.

Built_ins

- To size the window MAINWIN to 200,200, do:
  SET_WINDOW_PROPERTY('MAINWIN', WINDOW_SIZE, 200,200);
- -Other SET_BUILTINS are:
- SET_CANVAS_PROPERTY, SET_BLOCK_PROPERTY etc.
- -The syntax for the built-in is:
  SET_WINDOW_PROPERTY (object name, property, value);
- -where object is a VARCHAR2 parameter in quotes, the data type of value depends on property.
- -It is important to consult help files for appropriate syntax and parameters for built-ins.
- -Variables to hold the IDs of items would be of type ITEM, blocks of type BLOCK and so on.
- -You are unable to use restricted built-ins like GO-ITEM, GO-BLOCK on navigational triggers like PRE_RECORD, PRE_TEXT_ITEM, POST_BLOCK, POST_QUERY etc.
- -A use of GO_ITEM Built-in is:
  GO_ITEM ('ZIPCODE.CITY');

Built_ins

- -To prevent the error message like:
- FRM-410528 CANNOT Find Window. InvalidId
- from showing when a built-in accesses an object that does not exist, it is wise to write the trigger so it can alert you or the user to the absence of an object. -Eg.

```
DECLARE
    v_form_name VARCHAR2(50);
    v_window_id  WINDOW;
BEGIN
    v_form_name:=GET_APPLICATION_PROPERTY (CURRENT_FORM_NAME);
    v_window_id:=FIND _WINDOW('MAINWIN');
    IF ID_NULL(v_window_id) THEN
    MESSAGE('MAINTAIN does not exist. Error in WHEN-NEW-FROM-INSTANCE
    trigger');
    RAISE FORM_TRIGGER_FAILURE;
    END_IF;
    SET_WINDOW_PROPERTY(v_window_id, TITLE, 'This is form ||v_form_name);
END
```

LOVS AND ALERTS

- LOVS AND ALERTS
- -Lists of value (LOVs) and alerts are visual objects with which users can interact.
- -They are different from items because they appear in their own windows and not positioned on a canvas.
- -An LOV is used to present a list of values from which a user can choose to populate items on a form.
- -Lovs can be created manually or with a wizard.
- -Alerts are used to present an important message to the user.
- -Alerts have buttons through which the user can respond to message being displayed.

LOVS AND ALERTS

- LOVs serve a number of purposes, like making data entry easier and ensuring data validity.
- -LOVs are usually assigned/attached to text items. E.g. an LOV attached to SECTION_ID displays a list of available sections of a course with their descriptions.
- -It is also possible to configure an LOV so that it is not attached to a specific item and is available no matter where the user has navigated.
- -All LOVs are based on a Form's object called record group, which are logical objects never displayed to the user.
- -A second group is similar to a database table and stores an array of values in a column and row format.

LOVS AND ALERTS

- A record group can be based on a query or a set of static values.
- -The example sections LOV was based on a record group that contained the following query:

```
SELECT s.section_id, c.course_no, c.description
FROM section s, course c
WHERE c.course_no=s.course_no
ORDER BY section_id;
```

- -LOV wizard can be used to create and configure an LOV. The wizard is also used to create the record group that serves as the source of the LOV.
- -There are different methods for displaying LOVs.

The LOV WIZARD

- **The LOV WIZARD**
- -You can access the LOV wizard by selecting Tools|LOV wizard from the Main Menu or by right clicking on any object in the Object Navigator.
- -To re-enter the LOV wizard for an existing LOV, right click on the LOV you wish to edit.
- -Any object and property created with the wizards can always be adjusted manually using
- Form Builder.
- -The LOV wizard has 9 pages namely.
- 1. Source Page
- -LOV wizard only creates record groups based on queries; it cannot define static record groups.

The LOV WIZARD

- On the source page, the wizard will create a record group object along with the LOV object.
- -The next few wizard pages will define properties for the record group.
- 2. SQL QUERY PAGE
- -The wizard takes the query you wrote here and stores for its record groups property, the Record Group Query property.
- -With the Build SQL Query button, it opens the Query Builder for building SQL statements.
- -SQL statements can also simply be typed or imported.
- -You can confirm that the SQL written is correct by clicking the check syntax button.

The LOV WIZARD

- Remember not to put a semi-colon after the query as the wizard returns an Invalid Character, Invalid SQL Statement error.
- 3. COLUMN SELECTION PAGE
- -On this page, the wizard populates new LOV's column mapping properties property. This property has multiple values that are displayed and configured in the LOV Column Mapping dialog.
- -It defines which columns will be displayed, each column's return item and the width and title of the LOV column.
- 4. COLUMN DISPLAY PAGE
- -The wizard will continue to populate the LOV's column mapping properties property.

The LOV WIZARD

- The Return value field specifies which of the LOV columns and their subsequent values should be used to populate items on the form.
- -The Look Up Return items button presents a list of available items that can serve as return items.
- 5. ADVANCED OPTIONS PAGE
- -The wizard sets the Record Group Fetch size property for the record group here. It also sets the Automatic Refresh Property for the LOV. It can also set the Filter Before Display Property for the LOV for reducing the number of fetched records due to a filter criteria.
- 6. ITEMS PAGE
- -So far, we have created a record group and configured an LOV. We need to attach the LOV to a form item like STUDENT.ZIP item.
- -on this page, the wizard will set the List of values property for the Form items.

The LOV WIZARD

- -The LOV has some built-in features, for working with them.
- -The type of Forms item for displaying LOV is a button
- -LIST_VALUES is a built-in used to display LOVs. It will work only if there is an LOV attached to the current item.
- -The cursor must be in an item that has LOV attached to it for LIST_VALUES to work.
- -Setting the SHOW_LIST button's mouse navigator property to No prevents the cursor from making SHOW_LIST Button the current item so that LIST_VALUES built-in opens LOV.
- -thus the user cannot open LOV with enter key but with mouse.

The LOV WIZARD

- -We can change this by coding the WHEN_BUTTON _PRESSED trigger as:

```
GO_ITEM('STUDENT.ZIP');
LIST_VALUES;
```

- -The GO_ITEM built-in causes the form to navigate to the STUDENT.ZIP item so that the LIST_VALUES built-in works well.
- -We can also replace the code in the WHEN_BUTTON_PRESSED trigger in the Object Navigator(right click to get PL|SQL editor). New code is:

```
DECLARE
   V_lov       Boolean;
BEGIN
   V_lov:=SHOW_LOV('ZIP_LIST');
END;
```

- -The SHOW_LOV built-in is used to display LOV's but it does not require LOV to be attached to an item.

The LOV WIZARD

- The SHOW_LOV is also a Boolean function that returns TRUE if user picks a value from the LOV and FALSE if they press the LOV's cancel button.
- -To code WHEN_BUTTON_PRESSED trigger to display messages depending on whether the user selects a value from the LOV the code is:

```
DECLARE
   v_lov Boolean;
BEGIN
   IF NOT SHOW_LOV ('ZIP_LIST') THEN
     MESSAGE ('The user cancelled the LOV');
     ELSE MESSAGE ('The user selected from the LOV');
   END IF;
END;
```

Alerts

- **Alerts are windows that contain messages and buttons.**
- **- The MESSAGE built-in was used to send messages to the user in previous chapters but there was no way for the user to respond to the message and the message may not be seen by the user.**
- **- Alerts are centrally placed and need to be responded to, by the user.**
- **- Alerts can have up to three buttons.**
- **- Eg. Alert with three buttons is:**

Alerts

CREATING ALERTS

Alerts are created and configured using the object Navigator and property palette. An alert is a separate window that is not positioned on a canvas. It is not visible in the Layout Editor at design-time. The only way to see an alert is at runtime.

Alerts

- DISPLAY ALERTS
- To display an alert, use the following built_in
  SHOW_ALERT (‘ALERT_NAME’);
- SHOW_ALERT is a function that returns a number
- To display an alert called ALERT1, the code is :

```
DECLARE
   v_alert_button NUMBER;
BEGIN
   v_alert_button := SHOW_ALERT (‘ALERT1’);
END;
```


Alerts

- The code for the three-button alert is like:

```
DECLARE
   v_alert_button NUMBER;
BEGIN
   v_alert_button := SHOW_ALERT (‘ALERT1’);
   IF v_alert_button = ALTER_BUTTON1 THEN
     code for Re-enter another value
   ELSE IF v_alert_button = ALTER_BUTTON2 THEN
     code to display LOV of zipcode
   ELSE IF v_alert_button = ALTER_BUTTON3 THEN
     code for cancel
   END IF;
END;
```

Alerts

- The constants ALERT_BUTTON1, ALERT_BUTTON2, ALERT_BUTTON3 correspond to buttons in the alert
- FIND_ALERT built_in can be used to get the alert's object ID.
- Alerts are displayed in standard size and cannot be re-sized or re-positioned.
- Alerts buttons are only displayed if they have a label.
- Alert is displayed in a modal window where the users cannot leave the window until they exit it.
- The three Alert styles are Stop, Caution and Note.
- SET_ALERT_PROPERTY built_in can be used to set the title of the alert and the alert's message text programmatically.

Alerts

```
DECLARE
    v_alert_id ALERT;
    v_alert_button NUMBER;
BEGIN
    v_alert_id := FIND_ALERT ('EXIT_ALERT');
    SET_ALERT_PROPERTY (v_alert_id, TITLE, 'Warning');
    SET_ALERT_PROPERTY (v_alert_id,
      ALERT_MESSAGE_TEXT, 'Are you sure you want to
      exit the form?');
    V_alert_buttons := SHOW_ALERT(v_alert_id);
    ------
END;
```

Alerts

- To get the alert to show when the user presses the Exit button on the tool bar and the ZIPCODE.EXIT button, we use a KEY_EXIT trigger to show the alert.
- Moving alert to KEY_EXIT trigger will have the effect of showing the alert whenever the user pressed the EXIT key on the key board.
- To get the alert to show when the user clicks the Exit button on the canvas, we change the WHEN_BUTTON_PRESSED trigger to

  DO_KEY ('EXIT_FORM');
- This means, when the user clicks the button, it fires the KEY_EXIT trigger as well.

Canvases and Windows

- CANVASES AND WINDOWS
- A canvas and its items must be assigned to a window to be visible
- A window must contain at least one canvas to be visible
- A single form can contain multiple instances of both canvases and windows
- WINDOWS
- Windows are Forms physical object with properties that determine their size and position.
- Under functional property category, they have a series of window-specific properties that determine whether the window can be minimized, resized, closed, etc.
- Two types of windows available are: 1.document and 2.dialog.
- A third style of windows called multiple document interface (MDI) window exists.

Canvases and Windows

- The MDI window serves as a parent window to all of the other windows in a form.
- Figure 8.2, pg 264 shows example of the three window styles.
- Document windows are completely contained by the parent MDI window, while the dialog window can be moved completely outside of the MDI boundaries.
- Windows has modal property that helps determine how and when the user can leave the window.
- Navigation cannot leave a modal window until the user has completed the task (e.g, select one in LOV or cancel) called for.
- Modeless windows allow the user the convenience of switching from one window to another.
- While MDI windows are modeless, document and dialog windows are modal.

Displaying Windows

- DISPLAYING WINDOWS
- SHOW_WINDOW is a window _specific built_in that can be used to open all types of windows.
- E.g., a window named MAINWIN can be opened with statement:
  SHOW_WINDOW ('MAINWIN');
- A window can also be opened by navigating to an item in the window or to a block that contains an item in the window.
- Using built_ins to navigate to an item can open/display canvases and windows
- E.g., the following code in a trigger:
  GO_ITEM ('BLOCK1.ITEM1');
- causes the canvas and window that ITEM1 is on, to be displayed.

CANVASES

- There are four types of canvases.
- 1. Content Canvas: The most common canvas type since every window must have one of this as its main canvas.
- 2. Stacked Canvases
- 3. Toolbar Canvases
- 4. Tabbed Canvases
- Fig 8.3, page 266 shows an example Canvas. A Canvas is composed of Canvas view (or viewport) and other parts including the frame.
- The viewport is the area of the canvas that is visible to the user.
- The size of the Canvas and its viewport are determined by properties.
- The Canvas viewport is smaller rectangle surrounded by thin black line enclosing the items and the frame
- The Canvas is the larger rectangular area ending where the surface of the Layout editor appears to have raised dots.

Canvases

- Some items can be placed outside the viewport, making them initially not visible, but as the user scrolls or navigates the form, the viewport can move to expose different parts of the Canvas.
- DIPLAYING CANVASES
- A Canvas could be opened or displayed using the GO_ITEM and SHOW_VIEW built_ins as:

```
GO_ITEM ('BLOCK1.ITEM1');
SHOW_VIEW('CANVAS1');
```

- Here, ITEM1 is assumed placed on CANVAS1 as navigating to an item on a Canvas, displays the Canvas.

WINDOWS AND CANVASES

Although document windows are contained one in one on MDI window for simple application, they can be configured to be open simultaneously in applications with multiple document windows so that users can toggle back and forth between them.

- The MDI window is not visible in the object navigator and through property palette. But its properties can be adjusted with code using the SET_WINDOW_PROPERTY built_in.
- The MDI is only available on Windows platform, but the PRE_FORM trigger can be used to set properties for the MDI window for Height, Width, and Title.
- E.g., the statements in the PRE_FORM trigger should look like:
  SET_WINDOW_PROPERTY (FORMS_MDI_WINDOW, WIDTH, 550);
- FORMS_MDI_WINDOW is a Forms constant for referring to this window as it is not possible to refer to MDI window in single

CONTENT CANVASES AND WINDOWS

- quotes or with an object ID.
- HIDE_WINDOW built_in explicitly closes a window.
- A Canvas must be assigned to a window to be visible.
- A stacked Canvas must always be stacked on a content Canvas to be visible. Thus, both Canvases are assigned to the same window.
- Canvases can be assigned to a window at run time through code.
- The properties that govern the size of the Canvas are under Physical category (Height and Width).
- The properties that govern the size of the viewport are under Viewport Category (Viewport Height and Viewport Width).
- While content Canvases have viewports, they do not have properties to set the size of their Viewports.

CONTENT CANVASES AND WINDOWS

- The size of a content Canvas' Viewport is the same as the size of the window it occupies.
- E.g., if the Content Canvas for WINDOW1 is CANVAS1 and WINDOW1's Height and Width properties are set to 200,200, then, this will be the size of CANVAS1's Viewport.
- The size of a Content Canvas's viewport are set by
  - Setting the Height and Width properties of the window it is assigned to.
  - Adjusting it visually with Layout editor.
- The object Navigator is the best tool for creating Content Canvas.
- It is possible to assign two Content Canvases to the same window but only one can be visible at a time.

CONTENT CANVASES AND WINDOWS

- Navigational built_ins (eg. GO_ITEM) or explicit built_ins (eg. SHOW_CANVAS) can be used to switch between Canvases.
- You can have many instances of the Layout Editor open making it easier for comparing Canvases. Canvas list item on the Layout editor is Utility toolbar and can also be used to toggle between Canvases.

STACKED CANVASES

- Stacked canvases are never the sole Canvas in a window.
- They are stacked on top of other Canvases and partially or Completely obscuring those canvases when displayed at run-time.
- To stack a stacked Canvas, we define their positions relative to the Content Canvas.
- Stacked Canvases have Height and Width properties as well as Viewport sizes and position.
- The Viewport is positioned:
  - Relative to the stacked Canvas itself
  - Relative to the Content Canvas it is stacked upon.
- The Viewport X position on Canvas and Viewport Y position on Canvas properties determine where the viewport will be placed on the stacked canvas itself.

STACKED CANVASES

- Stacked Canvases can be used to simulate scrolling views for items being displayed in tabular fashion but they are too many to comfortably fit on a normal size Canvas.
- Stacked Canvas can be created with object navigator, Layout editor or Layout Wizard although Layout editor is the best tool for creating it.
- The View/Stacked Views feature lets you view the size and position of the stacked Canvas on their content canvases.
- GET_VIEW_PROPERTY is a built_in that returns TRUE if the canvas is visible and FALSE otherwise. Usage is :
- Variable := GET_VIEW_PROPERTY ('Canvasname',VISIBLE);

TOOLBAR CANVASES

- GO_BLOCK built_in is similar to GO_ITEM built_in as it forces navigation to the first item in the block being referenced.
- A toolbar Canvas positions toolbar items either horizontally along the top of a window or vertically along the left-hand edge of the window.
- Unlike other Canvas types, toolbar Canvases can be assigned to the MDI window.
- You must create a group of items to place on a toolbar. Eg. the utility toolbar in the Layout Editor includes two list items.
- Typical Toolbar have iconic labels to make toolbar buttons typical, adjust the iconic property.
- To quickly position all of the buttons next to each other, with the Layout Editor feature:
- Arrange | Align Objects, Stack Horizontally

TOOLBAR CANVASES

- Multiple horizontal toolbars are common
- The &SMARTBAR value in the Menu Module property assigns a menu toolbar also known as Smart bar to the default Forms menu
- The default settings (menu and smart bar) can be overridden with your own menus.
- Use the following statements for buttons:

```
SAVE_DO_KEY ('COMMIT_FORM');
EXIT_DO_KEY('EXIT_FORM');
ENTER_QUERY_DO_KEY('ENTER_QUERY');
EXECUTE_QUERY_DO_KEY('EXECUTE_QUERY');
CANCEL_QUERY_DO_KEY('EXIT_FORM');
```

REUSABLE OBJECTS

- The EXIT_FORM and ENTER_QUERY buttons should not be enabled when in Enter Query Mode.
- The CANCEL_QUERY button should be disabled when the form is not in Enter Query Mode.

- REUSABLE OBJECTS
- Copy or Subclass feature can be used to create one object based on another object.
- Subclass feature though creates a duplicate object and maintains a link between the source version and the subclassed version.
- The subclassed version inherits all of the properties of the source object and all of its changes.

REUSABLE OBJECTS

- The subclassed TOOLBAR does not have to remain an exact duplicate of the source.
- Properties of the subclass not inherited can be changed and inherited properties can be overridden.
- Any Forms object can be subclassed. Eg. by subclassing a block, you are subclassing its items and triggers.
- IF you subclass a canvas, its frames and other graphic objects are subclassed as well.
- To subclass objects in Forms do:
- Drag the source object from its Form and drop it in the target Form.
- Select subclass when prompted by an alert.

REUSABLE OBJECTS

- Subclassed objects are indicated by a red arrow over their icon in the object navigator.
- To learn about the source of the subclassed object you look at the subclassed object's Subclass Information property.
- The object Name list shows the name of the source object.
- The module list item shows the name of the forms module that the source object belongs to.
- There is a small black check next to all of the properties that are being inherited.
- The following 4 icons indicate if and how the property has been changed or inherited.

REUSABLE OBJECTS

- 1. The small circle indicates that the property value has not been changed from its default value.
- 2. The green box property indicates that this property has been changed from its default value.
- 3. The black check mark property indicates that the value in this property is being inherited from a source object, a visual attribute or a property class.
- 4. The black check mark with a red x property indicates that this property's value was inherited, but that the inheritance has been overridden by a change made at the object level.

ADVANTAGES OF SUBCLASS FEATURE

- Imagine an application with 50 forms each of which needs a PRE-FORM trigger. IF you subclass these 50 triggers from a source object, you would be able to edit and change all 50 simply by changing the source object.
- We can also use Subclass Information property to make an existing object, a subclassed version of a source object.
- The form that holds the source object must be open when we perform Subclassing.

VISUAL ATTRIBUTES AND PROPERTY CLASSES

- Visual attribute object contains all properties that determine font and color.
- Visual attributes can be applied to physical objects like items, frames, canvases, windows, LOVs and alerts. They cannot be applied to logical objects like blocks or record groups since those objects have no color or font properties.
- Property classes have a few more features than visual attributes that make them more powerful.
- 1. Not just font and color but all properties can be included in a property class.
- 2. Property classes do not have to be in the same form as the objects they are applied to.

VISUAL ATTRIBUTES AND PROPERTY CLASSES

- 3. Property classes can have triggers attached to them.
- While Visual objects have a standard set of properties that cannot be changed, property classes have no standard properties.
- Thus property classes can include properties from the Functional, Database, Record, and other property categories.
- Since a property class can include any property, its use is not limited to the physical objects as is the case with visual attributes.
- A property class can be applied to any object including logical objects like blocks, record groups and triggers.
- A property class can be created by
- 1. Selecting a set of properties to be included (eg. X Position, Width, Height, Background color, Foreground color for canvas) in the property palette, click the property class button on the property palette's toolbar.

Property Class

- 2. You can also create the property class in the object navigator and copy and paste the properties from the source item into property class.
- These are 3 values for Visual Attribute Type Property:
- 1. Common: for text and display items, canvases, windows, and LOVs [applies to item & prompt].
- 2. Prompt: for only the prompt property of an item and overrides properties set by type Common.
- 3. Title: which applies to frame titles.
- You cannot apply a property class to more than one object at a time
- To add properties to a property class use Create button on the Property Palette's toolbar and select from the list.

HOW CAN VISUAL ATTRIBUTES BE CREATED?

- To create Visual attributes
- Open form
- Locate Visual Attributes node in the object navigator and create a visual attribute. Name it test_item. Set the Font Name property and Foreground property.
- By setting an item's Visual attribute property to that of a named Visual attributes like test_item, the visual properties are inherited but not Subclassed.

OBJECT GROUPS AND OBJECT LIBRARIES

- An object group is an object within a forms module, while an object library is a module unto itself.
- Object groups are logical containers that are only visible in the object navigator.
- Eg. Object group is TOOLBAR object group consisting of:
  - TOOLBAR BLOCK
  - TOOLBAR CANVAS
  - TOOLBAR_MODULE property class
  - MAINWIN WINDOW

OBJECT LIBRARIES

- An object library is a module stored in a separate file that can be opened and configured in the Form Builder.
- Library files have .olb extension
- You can open object library modules in the LIBRARY window, view their objects and copy or Subclass their object into forms modules.
- An extensive object library called Stndrd20.olb has been installed with Oracle Developer, which you can use. It can be found in
- \ORACLE_HOME\tools\devdemo60\demo\forms\stndrd20.olb
- Object libraries hold actual instances of objects not just pointers to them.
- Object libraries should contain source objects not their subclasses.
- To edit or change an object in an object library copy the object into a form, edit and drag it back to object library.

TEMPLATE FORMS/REUSABLE CODE (TRIGGERS)

- TEMPLATE FORMS
- Sometimes, we may want to create new forms that already contain a standard set of objects.
- When you want to create a new form based on a template form, select
- FILE/NEW/FORM Using Template from the Main Menu.
- When you base a form on a template, the Form Builder remember the form so that you do not accidentally overwrite the template.
- REUSABLE CODE (TRIGGERS)
- Program units are Forms objects that allow for reusing trigger codes. Eg. You can create a single program unit that all of the WHEN_VALIDATE_ITEM triggers in a form can reuse.
- We can store the program unit in a PL/SQL library module so that triggers in other forms modules can reuse it.

PROGRAM UNITS

- Program units can be PL/SQL packages, procedures, or functions and can accept parameters and return values.
- Eg. We can create a DISPLAY_CANVAS code as a program unit that can be called by many triggers. The code for the program units is :

```
PROCEDURE DISPLAY_CANVAS IS
BEGIN
      SHOW_VIEW ('CANVAS_1');
END;
```

- Each trigger would then call it with:

  DISPLAY_CANVAS;
- The code is in a central location and accessed by other triggers and program units in the form.
- The above program unit can be changed to a more flexible version that accepts parameters as

PROGRAM UNITS

```
PROCEDURE DISPLAY_CANVAS (P_CANVAS-NAME
   VARCHAR2) IS
BEGIN
   SHOW_VIEW(P_CANVAS_NAME)
END;
```

- To call the new program unit, the trigger or calling program unit has

  DISPLAY_CANVAS('CANVAS_3');
- Program unit can be created using object navigator.
- Program unit can be subclassed to other forms or stored in the object library to increase its re-usability.
- You can call a forms level packaged PL/SQL object the same way you would if it were stored in the database by prefixing the name of the program unit with the name of the package.

PL/SQL Libraries

- PL/SQL LIBRARIES
- PL/SQL Libraries are used to store program units when the program units will be used by most or all of the forms in an application.
- PL/SQL libraries are not forms modules.
- They are separate modules that can be created, edited, and compiled in the Form Builder.
- Before a form can execute the objects in a PL/SQL Library, the Library must be attached to the form using the object navigator.
- PL/SQL Library modules are saved in two files formats .pll files containing the source and executable codes of the library and .plx files containing only the executable code.
- Once a Library is saved as a .pll file, you can attach it to as many forms as necessary.

PL/SQL Libraries

- When you attach a PL/SQL Library to a form, the Library and its code are not stored within the form, but the form knows that Library exists and can reference its subprograms.
- Direct references to bind variables use the item name which may also be a System variable as is:
- v_student_id := : STUDENT.STUDENT_ID;
- and
- v_item := : SYSTEM.CURSOR_ITEM;
- PL/SQL Library would not compile if it contains direct references to bind variables.
- To make references to bind variables in Library code, we need to do so, indirectly using the copy and NAME_IN built_ins as in :

PL/SQL Libraries

- COPY (v_date; 'COURSE.CREATED_DATE');
- This statement copies v_date into the COURSE.CREATED_DATE item but the item name is being referenced indirectly.
- NAME_IN built_in allows indirect reference to bind variables with return of their values. E.g.,
- v_item := NAME_IN(':SYSTEM.CURSOR_ITEM');
- CREATION AND ADDITION OF PROGRAM UNITS TO PL/SQL LIBRARY
- We can create new program units directly within the PL/SQL library using the object navigator's Create button.
- You can also add an existing program unit to a PL/SQL Library by dragging and dropping it.

PL/SQL Libraries

- Two ways to compile a .pll module from within the Form Builder are:
- Use CTRL/T
- From the main menu select Program/Compile Module
- This compiles all of the PL/SQL objects in the PL/SQL Library. You can also use one of the compilation options under the Program heading in Main Menu (Compile and Compile Selection). The compile option has two further options (Compile Incremental and ALL)
- Compile Incremental Compiles all PL/SQL objects that have changed since last compilation
- Compile ALL compiles all objects in current module.
- Compile Selection compiles the currently selected PL/SQL object.

PL/SQL Libraries

- These compilation options apply to forms modules as well as PL/SQL Library modules.
- To attach a .pll Library to a form do:
- Select the Attached Libraries node and click Create. The Attach dialog opens.
- The Attach dialog allows you search the file system, the database or both for the PL/SQL Library. You cannot drag a PL/SQL Library in the Object Navigator and drop it on a form.
- Whenever Forms has to reference objects or code stored in other modules, it uses the Windows Registry to locate the module. Thus, there is no need for hard-coding path to find them.
- As soon as we save changes to the common.pll Library, the changes are available to any form to which the PL/SQL library is attached.
- Other forms that have this PL/SQL library attached do not have to be opened or recompiled to benefit from the changes.

Stored PL/SQL Database Objects

- The PL/SQL Library is a separate module
- By keeping the object group and object library up-to-date, all of the forms, including template forms that have subclassed versions of the object group file will inherit the changes.
- We need to attach the .pll library to the forms that have subclassed versions of the objects (e.g., triggers).
- STORED PL/SQL Database OBJECTS
- Forms modules can call PL/SQL objects stored in the database as they do PL/SQL objects stored in attached libraries.
- Database stored packages, procedures and Functions can be called from a Forms application and can be passed parameters.

Stored PL/SQL Database Objects

- The syntax for calling stored PL/SQL object is the same as that for calling a program unit in a form.
- E.g., to call a stored PL/SQL procedure called DELETE_STUDENT and pass a value in a variable, use in forms application code:

  DELETE_STUDENT (v_student-id);
- Forms will look for a PL/SQL object named DELETE_STUDENT and execute the first instance that it finds.
- First, Forms will search the forms module for a program unit named DELETE_STUDENT. If it does not find one, it will search any attached libraries. If it does not find one there, either, it will then search the database.

Stored PL/SQL Database Objects

- Data-intensive objects are better stored in the database and can be used by both Forms and non-Forms applications.
- PL/SQL objects that successfully Compile and run in the database can also compile and run in the Forms if they do not have Forms specific statements like built_ins, or bind variables.
- PL/SQL objects can be moved back and forth between the database and a Forms application by dragging and dropping the objects in the object Navigator from the forms module to the Database Objects node.

CREATING STORED DATABASE

- To make a function / procedure a stored database function / object, in Object Navigator, we can drag the function/procedure from the program Units node to the Database Objects node.
- You need to have the eg. STUDENT Schema node expanded to see Stored Program Units node beneath it. The function is then stored in the database as one of the objects owned by STUDENT.

MULTIPLE –FORM APPLICATIONS CALLING ONE FORM FROM ANOTHER

- Three built_ins are used to call one form from another.
- 1. OPEN_FORM
- 2. CALL_FORM
- 3. NEW_FORM
- The statement for opening a form called Course looks like:
  OPEN_FORM ('COURSE');
- Since the full path is not included in the OPEN_FORM statement, the FORMS Runtime will look for the COURSE.fmx file in the paths listed in the Registry.
- Including full path makes the application less portable.
- The form name is a mandatory parameter. But, each of the three built_ins can accept other parameters that can affect the behavior and state of the calling form and called form.
- The built_ins also alter the behavior of the calling and called form.

MULTIPLE –FORM APPLICATIONS CALLING ONE FORM FROM ANOTHER

- Assume the calling form is FORM_A and the called form is FORM_B.
- OPEN_FORM ('FORM_B');
- This call has the effect of opening FORM_B and FORM_A will remain active and accessible such that the user can navigate between both forms.
- CALL_FORM('FORM_B');
- Has the effect of opening FORM_B in modal form, such that the user is not able to leave FORM_B until it has been exited or closed. FORM_A may be visible but none of its items will be accessible.
- NEW_FORM('FORM_B');
- This closes FORM_A but opens FORM_B.

MULTIPLE –FORM APPLICATIONS CALLING ONE FORM FROM ANOTHER

- The EXIT_FORM built_in exits and closes the current form. The CLOSE_FORM built_in exits and closes a form but requires a form name as a parameter.
- CLOSE_FORM('FORM_B');
- PASSING VALUES TO CALLED FORMS
- Values can be passed to called form through
  - Global variables.
  - Parameter lists.
- Global variables are user_defined variables that are visible to all objects in a Forms session.
- Thus, any PL/SQL object within a single form or multiform application can reference the value of a global variable.

MULTIPLE –FORM APPLICATIONS CALLING ONE FORM FROM ANOTHER

- An example declaration of a global variable is:
- :global.username := GET_APPLICATION_PROPERTY(USER_NAME);
- You could reference the value of :global.username in any PL/SQL object within FORM_B (the called form).
- If FORM_C and FORM_D were open as well, then, they could also reference
- :global.username and see the same value.
- To create parameters list in FORM_A (the calling form), you use the following built_in
- v_plist_id := CREATE_PARAMETER_LIST('forms_params');
- v_plist_id is a local variable that would have been declared in the PL/SQL block. forms_params is the name of the parameter list being created; its object ID is being stored in v_plist_id.
- Once you have created the parameter list you add parameter and their values to it.
- ADD_PARAMETER (v_plist_id,'P_1', text_parameter, v_value);

MULTIPLE –FORM APPLICATIONS CALLING ONE FORM FROM ANOTHER

- v_plist_id tells the ADD_PARAMETER built_in which parameter list to work with. P_1 is the name of the parameter being added to the list; text_parameter indicates the type of parameter. When passing parameters from form to form, they must be defined as text_parameters. v_value is the value being assigned to P_1.
- The parameters in the called Form (FORM_B) must be the same names as their corresponding actual parameters in the calling form (FORM_A).
- Thus, in the above example, we should use the object navigator to create a parameter also called P_1.
- Called forms need to be compiled so that calling forms can run them.

MULTIPLE –FORM APPLICATIONS CALLING ONE FORM FROM ANOTHER

- The CALL_FORM, OPEN_FORM, and NEW_FORM built_ins will only run .fmx (executable) files and not .fmb (form files).
- If forms are open, you can navigate from one to the other with the statement

  GO_FORM ('FORMNAME');
- The GO_FORM built_in is passed a name or object ID to open a specific form.
- Two other built_ins for navigating from form to form and their syntax is as follows:

  NEXT_FORM;
- and

  PREVIOUS_FORM;
- They navigate in the order the forms were opened.

MULTIPLE –FORM APPLICATIONS CALLING ONE FORM FROM ANOTHER

- The MDI toolbar works for both forms. We must explicitly exit each form.
- The OPEN_FORM built_in can accept as many as five parameters although we are not allowed to include all of the parameters as FORMS uses default values for each parameter.
- This rule applies to other built_ins that accept parameters.
- The ADD_ENROLLMENTS unit code is:

MULTIPLE –FORM APPLICATIONS CALLING ONE FORM FROM ANOTHER

```
PROCEDURE ADD_ENROLLMENTS IS
v_plist_id          PARAMLIST;
v_where             VARCHAR2(50);
BEGIN
      v_where := 'STUDENT_ID ='||:STUDENT.STUDENT_ID;
      V_plist_id := CREATE_PARAMETER_LIST('forms_params');
      ADD_PARAMETER(v_plist_id,'P_1',TEXT_PARAMETER,v_where);
      COMMIT_FORM;
      CALL_FORM('EX11_02', NO HIDE, NO_REPLACE, NO_QUERY_ONLY,
         v_plist_id);
END;
```

- The code for WHEN_NEW_FORM_INSTANCE is :

```
DECLARED
      v_where     VARCHAR2(100);
BEGIN
      IF:PARAMETER.P_1 IS NOT NULL THEN
            v_where := :PARAMETER.P_1;
      SET_BLOCK_PROPERTY('STUDENT', 'DEFAULT_WHERE, v_where);
      EXECUTE_QUERY;
      GO_BLOCK ('ENROLLMENT');
      END IF;
END;
```


MULTIPLE –FORM APPLICATIONS CALLING ONE FORM FROM ANOTHER

- **Format of the Form built_ins are**

  OPEN_FORM ('Formname', ACTIVATE, NO_SESSION, NO_SHARE_LIBRARY_DATA,'parameter_list_name');

  NEW_FORM (formname, rollbackmode, querymode, datamode, parameterlist);

  CALL_FORM (formname, display, switchmenu, querymode, datamode, parameterlist);

- **If parameter list exists and the value is not null, we can use the built_in DESTROY_PARAMETER_LIST to destroy the parameter list as in:**

```
v_plist_id := GET_PARAMETER_LIST('forms_params');
IF NOT ID_NULL (v_list_id) THEN
      DESTROY_PARAMETER_LIST(v_list_id);
END IF;
```

MULTIPLE –FORM APPLICATIONS CALLING ONE FORM FROM ANOTHER

- Segment like the above is used to destroy a parameter list before creating a new one as may be needed in a loop.
- For example, in the code below, the parameter list is being used to pass a WHERE clause from one form to another.
- New code for ADD_ENROLLMENTS is

```
PROCEDURE ADD_ENROLLMENTS IS
v_plist_id     PARAMLIST;
v_where        VARCHAR2(50);
BEGIN
      V_plist_id := GET_PARAMETER_LIST('forms_params');
      IF NOT ID_NULL (v_list_id) THEN
      DESTROY_PARAMETER_LIST(v_plist_id);
END IF;
```


MULTIPLE –FORM APPLICATIONS CALLING ONE FORM FROM ANOTHER

```
v_where := 'STUDENT_ID = '||:STUDENT.STUDENT.ID;
v_plist_id := CREATE_PARAMETER_LIST('forms_params');
ADD_PARAMETER (v_plist_id, 'P_1', TEXT PARAMETER,
    v_where);
COMMIT_FORM;
CALL_FORM('EX11_02', NO_HIDE,
    NO_REPLACE,NO_QUERY_ONLY, v_plist_id);
END;
```

- To create a parameter in form using object navigator do;
- Select the parameters node and click Create button. The parameter should be named.
- Parameters are referred to, by the
- :PARAMETER.PARAMETER_NAME syntax.
- Parameters lists are not visible across forms and you are not required to make any reference to the list itself in the called form.

MULTIPLE –FORM APPLICATIONS CALLING ONE FORM FROM ANOTHER

- The parameter in called form is an object in the form that can be referenced by any trigger or PL/SQL object in the forms module.
- If you reference the parameter in an attached PL/SQL Library, you must do so indirectly with the NAME_IN built_ins as:

  NAME_IN('PARAMETER.P_1');
- In the example above, the calling form is passing a WHERE clause to the called form, setting the WHERE clause for a block in the called form, and then executing a query.
- The POST built_in applies changes to the database but does not commit them. These changes can then be committed or rolled back later.

ORACLE FORM AND ORACLE REPORTS

- Oracle Forms and Oracle Reports products have been tightly integrated so that forms modules can use built_in to call reports modules and pass parameters to them.
- RUNNING ORACLE REPORTS FROM FORMS
- The Report Builder is used to design and build reports modules. It has a similar interface to the Form Builder with object navigator, property palette, layout editor and PL/SQL editor.
- Reports modules can be run by themselves using Reports Runtime or called from forms.

  RUN_PRODUCT
- RUN_PRODUCT built_in can be used to call any of the three types of Oracle Developer modules: forms, reports, graphics.
- A report can be called from a trigger or program unit with only one single statement with the following syntax:

  RUN_PRODUCT (product, module, commode, execommode, location, parameter_list, display);

ORACLE FORM AND ORACLE REPORTS

- **Eg. a call to run a report called COURSE looks like:**
  RUN_PRODUCT (REPORTS, 'COURSE', ASYNCHRONOUS, RUNTIME, FILESYSTEM, 'NULL',NULL);
- **A tighter and less flexible built_in for calling only Oracle Report is RUN_REPORT_OBJECT.**
- **Information on use is available on help system.**
- **Report modules have .rdf extensions.**
- **The Commode parameter determines whether or not the user should be able to return to the calling module before the called module has been exited. ASYNCHRONOUS setting for Commode means can navigate back and forth between modules, while SYNCHRONOUS means must exit the called module (eg. report in this case) before returning to calling module.**

ORACLE FORM AND ORACLE REPORTS

- **The execmode parameter is set to RUNTIME above but could have been set to BATCH. BATCH mode sends the report results to a file or printer and does not allow it to be viewed on the screen.**
- **The location parameter indicates where the Reports Runtime should look for the module to be run. Modules are stored in the filesystem or database.**
- **With the above Call, the location is set to filesystem but we have not included the full path with the report name. Thus, the Reports Runtime will search the directories indicated in the Windows Registry for the report file.**
- **The sixth parameter in the RUN_PRODUCT statement is for the parameter list that form is passing to report. 'NULL' (in single quotes) indicate there are no parameters being passed.**

ORACLE FORM AND ORACLE REPORTS

- The last parameter is the display parameter, which requires a value only if RUN_PRODUCT is being used to call on Oracle graphics module. This parameter is set to NULL (without single quotes).
- CREATING A REPORT
- To create a report, we open a form in Form Builder. Then, use the Reports node in the object based on an existing report.
- In the new Report dialog, select use Existing Report File radio button and enter the name of the reports module in the Filesave field.
- If you were to select Create New Report File, the Form Builder would launch the Report Builder to create a new reports module.

ORACLE FORM AND ORACLE REPORTS

- On the other hand, the RUN_REPORT_OBJECT built_in does not accept parameters like commode, execmode, etc. They can be set using the properties for the report object.
- The report object has additional properties for other SYSTEM.PARAMETERS like Destination name and Destination Format.
- Destination Name and Format can also be set using the RUN_PRODUCT built_in but not set using properties. They would have to be added to a parameter list.
- The code for the WHEN_BUTTON_PRESSED trigger looks like the following in order to open the report with RUN_REPORT_OBJECT built_in.

ORACLE FORM AND ORACLE REPORTS

```
DECLARE
    v_repobj_id      REPORT_OBJECT;
    v_repins_id      VARCHAR2(100);
BEGIN
    v_repobj_id := FIND_REPORT_OBJECT('STUDENT');
    v_repins_id := RUN_REPORT_OBJECT(v_repobj_id);
END;
```

- FIND_REPORT_OBJECT built_in returns the object ID of the report object.
- RUN_REPORT_OBJECT runs the report and returns a value (a unique ID for this job).
- Eg. REPORT_OBJECT_STATUS(v_repins_id);
- returns a VARCHAR2 value, indicating whether this job has finished, is cancelled or still running.

ORACLE FORM AND ORACLE REPORTS

- For more details, see the online help for Oracle Reports.
- The following statement shows the passing report object ID and ID or name of a parameter list.

  v_repins_id := RUN_REPORT_OBJECT (v_repobj_id, v_plist_id);
- The properties of the report object at runtime could be set with the
- SET_REPORT_OBJECT built_in.
- PASSING PARAMETERS TO REPORTS
- - To pass parameters to Reports do:
- 1. Create a parameter list in a form
- 2. Add parameters to it
- 3. Pass it to the called reports module

ORACLE FORM AND ORACLE REPORTS

- Both the RUN_PRODUCT_OBJECT and RUN_PRODUCT built_ins accept parameter list names or parameter list object IDs.
- - Eg. assume we had created a parameter list and stored its object ID in a variable called v_plist_id, to call a report named SECTIONS, we use:
  RUN_PRODUCT (REPORTS,'SECTIONS',ASYNCHRONOUS, RUNTIME, FILESYSTEM, v_plist_id, NULL);
- OR
  v_repobj_id := FIND_REPORT_OBJECT('SECTIONS');
  v_repins_id := RUN_REPORT_OBJECT (v_repins_id, v_plist_id);
- For the report to accept the parameter, it should be created in the Report Builder at design_time.
- Similar to CALL_FORM requirements for passing parameters between forms, if a parameter called 'P_1' had been defined in the reports module at design_time, the ADD_PARAMETER statement in the forms module would look like the following:
  ADD_PARAMETER(v_plist_id,'P_1', text_parameter, v_value);

PARAMETERS IN REPORTS MODULES

- P_1 is a user parameter in the reports module because the programmer defined it. There is a standard set of system parameters pre_defined by the system (eg. DESNAME and DESTYPE).
- Reports by default, displays user defined Form parameter like P_1 on its result. When not described report can be forced to not display user defined parameter like P_1 with the following:
  ADD_PARAMETER (v_plist_id, 'PARAMFORM', text_parameter, 'NO');
- Example
- Open the form EX12_01.fmb in Form Builder
- The name of the parameter is STUDENTL report is P_1.

PARAMETERS IN REPORTS MODULES

- The code for creating a parameter list and adding the current STUDENT_ID to the list as a parameter is:

```
DECLARE
    v_plist_id  PARAMLIST;
BEGIN
    v_plist_id := GET_PARAMETER_LIST ('rep_params');
    IF NOT ID_NULL (v_plist_id) THEN
      DESTROY_PARAMETER_LIST(v_plist_id);
    END IF;
    v_plist_id := CREATE_PARAMETER_LIST('rep_params');
    ADD_PARAMETER(v_plist_id, 'P_1', text_parameter,
    :STUDENT.STUDENT_ID);
    ADD_PARAMETER(v_plist_id, 'PARAMFORM', text_parameter,
    'NO');
END;
```


PARAMETERS IN REPORTS MODULES

- Note that the above includes code segments to destroy the parameter list if it already exists and to add a parameter to suppress the reports module's parameter form.
- To call STUDENTL report, the code is in the form trigger:

  RUN_PRODUCT (REPORTS, 'STUDENTL', ASYNCHRONOUS, RUNTIME, FILESYSTEM, v_plist_id, NULL);
- To call STUDENT report in the form using RUN_PRODUCT_OBJECT built_in instead, modify the code above with the following.

```
DECLARE
---
```

Forms Menus

```
v_repobj_id   REPORT_OBJECT;
v_repins_id   VARCHAR2(100);
BEGIN
--
--
v_repobj_id := FIND_REPORT_OBJECT(STUDENTL);
v_repins_id := RUN_REPORT_OBJECT(v_repins_id, v_plist_id);
END;
```

- FORMS MENUS
- Rather than run our forms with default menus with basic functionalities for editing, navigation, querying and so on, we can create our own custom menus that provide more functionalities.
- We can also use menu modules to implement security features that control access to the application.

MENU MODULES

- Menu modules and their objects can be created in the Form Builder.
- The 4 main components of a menu module are:
- 1. The menu module
- 2. The main menu
- 3. Individual menus
- 4. Menu items
- The menu module itself, like a forms module, is not visible to the user. It is the logical container object that owns all of the other menu objects.
- Eg. Fig-13.1, pg 422.\
- The main menu is the horizontal bar across the top that contains the labels INDIVIDUAL_MENU_1, INDIVIDUAL_MENU_2, and Window.
- Each of these labels is an individual menu that contains a group of menu items.
- The menu items are the most important objects in the menu module because they are what the users select to initiate actions.

MENU ITEMS

- MENU ITEMS
- There are 5 types of menu items: plain, magic, check, radio, and separator.
- Plain menu items display text labels and have PL/SQL behind them.
- Eg. a plain menu item labeled Save might have the following code behind it:

  DO_KEY ('COMMIT_FORM');
- Magic items are the most convenient items because they already have code associated with them. They can be used to perform functions common to most applications.
- Eg. a magic item of type Copy has the Code for copying text already associated with it.
- At design time, we indicate that a certain menu item should be a magic item and Forms will associate the proper Code with it.

CREATING AND CONFIGURING MENU MODULES

- CREATING AND CONFIGURING MENU MODULES
- Form Builder provides Menu Editor for creating and defining menus. Menu editor is a better tool for menu's than object navigator.
- ATTACHING MENU MODULES TO FORMS
- For a menu to be visible and accessible to the user, it must be attached to a forms module.
- Menu modules have binary formats (.mmb files) and executable formats (.mmx files).
- When a menu is attached to a form, and the form is run, The Forms Runtime searches for and executes the .mmx version of the menu module.
- You must explicitly compile a menu module using CTRL/T before using it.

CREATING MENUS AND MENU ITEMS

- It is important to re-compile the menu module prior to each test or an order version of .mmx may be used by the Form Buillder.
- CREATING MENUS AND MENU ITEMS
- Open a form (eg. EX13_01.fmb) in Form Builder and Run the form.
- The second individual menu is Edit. Its menu items are Cut, Copy, Paste, Error and Display List.
- Cut, Copy, and Paste are magic items. Error and Display List are plain menu items with codes behind them.
- Default menu is attached to every newly created form and cannot be edited directly. E.g., you cannot make changes to a menu module called DEFAULT.mmb in Form Builder.
- You must either use DEFAULT as is, or attach a different menu module to the form.

CREATING MENUS AND MENU ITEMS

- You can reuse some features within DEFAULT through access to a menu module called menudef.mmb that is installed along with Oracle Developer and can be found in the directory
- \ORACLE_HOME\DEVDEM60\DEMO\FORMS
- DEFAULT MENU FORM
- It is common to attach a functionless form to a menu to serve as a Starting point or Splash screen. It can contain a simple message or an image to introduce the application. It can also contain functional items like buttons or display items.
- Even though not used, block and item are part of the form.
- It is common to attach a menu to a form that is always opened first by the user.
- In the object navigator, the objects created automatically are: Main Menu, MENU1, and individual MENU called ITEM2. In the Menu Editor, there is an individual menu labeled New Item.
- How is the Menu Item Created?

MENU SECURITY

- Click the Create Down button on the Menu Editor's toolbar.
- Hot keys can be set for the labels of the menu items as well so that the user can press the key (eg. for S for Save or X for Exit).
- If you want the hot key for Exit to be "x" , you write the label as :
- E&xit
- To make the menu item Exit a magic item that quits the application set its Menu Item Type property to Magic and its Magic Item property to Quit.
- MENU SECURITY
- You can grant and restrict access to individual menus and menu items through a single menu system.
- The menu security system is integrated with the database and we can base menu item access on database roles.
- Eg. In the database, we have created two roles:
  - STUDENT_USERS
  - OFFICE_USERS

MENU SECURITY

- Various users are assigned to each role. While STUDENT_USERS can only query the course and SECTION tables, OFFICE_USERS can perform all DML operations.
- When creating the menu modules, you can set properties for the menu items so that STUDENT_USERS role can only access the menu items that calls the course and section master detail form.
- You would use the same properties to indicate that the OFFICE_USERS role will be able to access all menu items in the application.
- STEPS FOR CONFIGURING MENU SECURITY
- Indicate which database roles have access to the menu module.
- For each individual menu item, indicate whether or not the roles specified in / should be granted access.

MENU SECURITY

- To practice menu security, we need to create some sample database users and roles.
- Run Scripts to create sample database users and roles.
- Build Forms schema objects and grant the sample database users access to them.
- Go over exercises on pp 436-444 for practice.