SCSX1056 – ORACLE & SQL
UNIT III - DATA BASE OBJECTS

Synonym – Sequences – View- Index – OODBMS Vs DBMS – Concepts of Object oriented programming – Features of OOPS – Advantages of Object orientation – Abstract data types – Object views – Varying arrays – Nested tables – Object tables – Object views with REFs.

Create Synonym

Purpose

Use the **CREATE SYNONYM** statement to create a **synonym**, which is an alternative name for a table, view, sequence, procedure, stored function, package, materialized view, Java class schema object, user-defined object type, or another synonym.

Synonyms provide both data independence and location transparency. Synonyms permit applications to function without modification regardless of which user owns the table or view and regardless of which database holds the table or view. However, synonyms are not a substitute for privileges on database objects. Appropriate privileges must be granted to a user before the user can use the synonym.

You can refer to synonyms in the following DML statements:

SELECT, **INSERT**, **UPDATE**, **DELETE**, **FLASHBACK TABLE**, **EXPLAIN PLAN**, and **LOCK TABLE**.

You can refer to synonyms in the following DDL statements: **AUDIT**, **NOAUDIT**, **GRANT**, **REVOKE**, and **COMMENT**.

Prerequisites

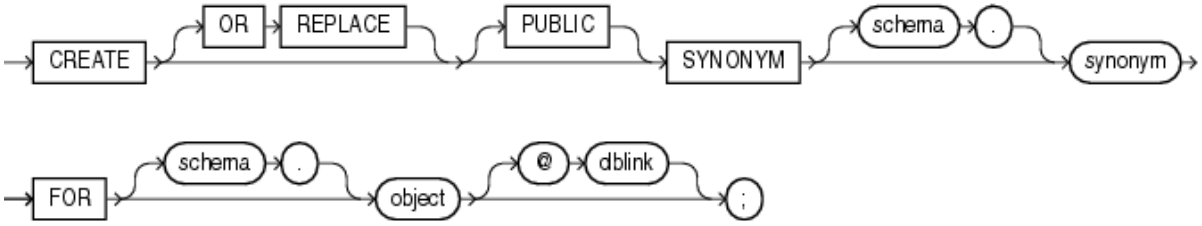
To create a private synonym in your own schema, you must have the **CREATE SYNONYM** system privilege.

To create a private synonym in another user's schema, you must havethe **CREATE ANY SYNONYM** system privilege.

To create a **PUBLIC** synonym, you must have the **CREATE PUBLIC SYNONYM** system privilege.

Syntax

***create_synonym*::=**



Semantics

OR REPLACE

Specify **OR REPLACE** to re-create the synonym if it already exists. Use this clause to change the definition of an existing synonym without first dropping it.

Restriction on Replacing a Synonym You cannot use the **OR REPLACE** clause for a type synonym that has any dependent tables or dependent valid user-defined object types.

PUBLIC

Specify **PUBLIC** to create a public synonym. Public synonyms are accessible to all users. However, each user must have appropriate privileges on the underlying object in order to use the synonym.

When resolving references to an object, Oracle Database uses a public synonym only if the object is not prefaced by a schema and is not followed by a database link.

If you omit this clause, then the synonym is private and is accessible only within its schema. A private synonym name must be unique in its schema.

Notes on Public Synonyms The following notes apply to public synonyms:

- If you create a public synonym and it subsequently has dependent tables or dependent valid user-defined object types, then you cannot create another database object of the same name as the synonym in the same schema as the dependent objects.
- Take care not to create a public synonym with the same name as an existing schema. If you do so, then all PL/SQL units that use that name will be invalidated.

schema

Specify the schema to contain the synonym. If you omit ***schema***, then Oracle Database creates the synonym in your own schema. You cannot specify a schema for the synonym if you have specified **PUBLIC**.

synonym

Specify the name of the synonym to be created.

Note:

Synonyms longer than 30 bytes can be created and dropped. However, unless they represent a Java name they will not work in any other SQL command. Names longer than 30 bytes are transformed into an obscure shorter string for storage in the data dictionary.

FOR Clause

Specify the object for which the synonym is created. The schema object for which you are creating the synonym can be of the following types:

- Table or object table
- View or object view
- Sequence
- Stored procedure, function, or package
- Materialized view
- Java class schema object
- User-defined object type
- Synonym

The schema object need not currently exist and you need not have privileges to access the object.

Restriction on the FOR Clause The schema object cannot be contained in a package.

schema Specify the schema in which the object resides. If you do not qualify object with ***schema***, then the database assumes that the schema object is in your own schema.

If you are creating a synonym for a procedure or function on a remote database, then you must specify ***schema*** in this **CREATE** statement. Alternatively, you can create a local public synonym on the database where the object resides. However, the database link must then be included in all subsequent calls to the procedure or function.

dblink You can specify a complete or partial database link to create a synonym for a schema object on a remote database where the object is located. If you specify ***dblink*** and omit ***schema***, then the synonym refers to an object in the schema specified by the database link. Oracle recommends that you specify the schema containing the object in the remote database.

If you omit ***dblink***, then Oracle Database assumes the object is located on the local database.

Restriction on Database Links You cannot specify ***dblink*** for a Java class synonym.

Examples

CREATE SYNONYM: Examples To define the synonym **offices** for the table **locations** in the schema **hr**, issue the following statement:

```
CREATE SYNONYM offices

        FOR hr.locations;
```

To create a **PUBLIC** synonym for the **employees** table in the schema **hr** on the **remote** database, you could issue the following statement:

```
CREATE PUBLIC SYNONYM emp_table

FOR hr.employees@remote.us.oracle.com;
```

A synonym may have the same name as the underlying object, provided the underlying object is contained in another schema.

Oracle Database Resolution of Synonyms: Example Oracle Database attempts to resolve references to objects at the schema level before resolving them at the **PUBLIC** synonym level. For example, the schemas **oe** and s**h** both contain tables named **customers**. In the next example, user **SYSTEM** creates a **PUBLIC** synonym named **customers** for**oe.customers**:

```
CREATE PUBLIC SYNONYM customers FOR oe.customers;
```

If the user **sh** then issues the following statement, then the database returns the count of rows from **sh.customers**:

```
SELECT COUNT(*) FROM customers;
```

To retrieve the count of rows from **oe.customers**, the user **sh** must preface **customers** with the schema name. (The user **sh** must have select permission on **oe.customers** as well.)

```
SELECT COUNT(*) FROM oe.customers;
```

If the user **hr**'s schema does not contain an object named **customers**, and if **hr** has select permission on **oe.customers**, then **hr** can access the**customers** table in **oe**'s schema by using the public synonym **customers**:

```
SELECT COUNT(*) FROM customers;
```

Create Sequence

Purpose

Use the **CREATE SEQUENCE** statement to create a **sequence**, which is a database object from which multiple users may generate unique integers. You can use sequences to automatically generate primary key values.

When a sequence number is generated, the sequence is incremented, independent of the transaction committing or rolling back. If two users concurrently increment the same sequence, then the sequence numbers each user acquires may have gaps, because sequence numbers are being generated by the other user. One user can never acquire the sequence number generated by another user. After a sequence value is generated by one user, that user can continue to access that value regardless of whether the sequence is incremented by another user.

Sequence numbers are generated independently of tables, so the same sequence can be used for one or for multiple tables. It is possible that individual sequence numbers will appear to be skipped, because they were generated and used in a transaction that ultimately rolled back. Additionally, a single user may not realize that other users are drawing from the same sequence.

After a sequence is created, you can access its values in SQL statements with the **CURRVAL** pseudocolumn, which returns the current value of the sequence, or the **NEXTVAL** pseudocolumn, which increments the sequence and returns the new value.

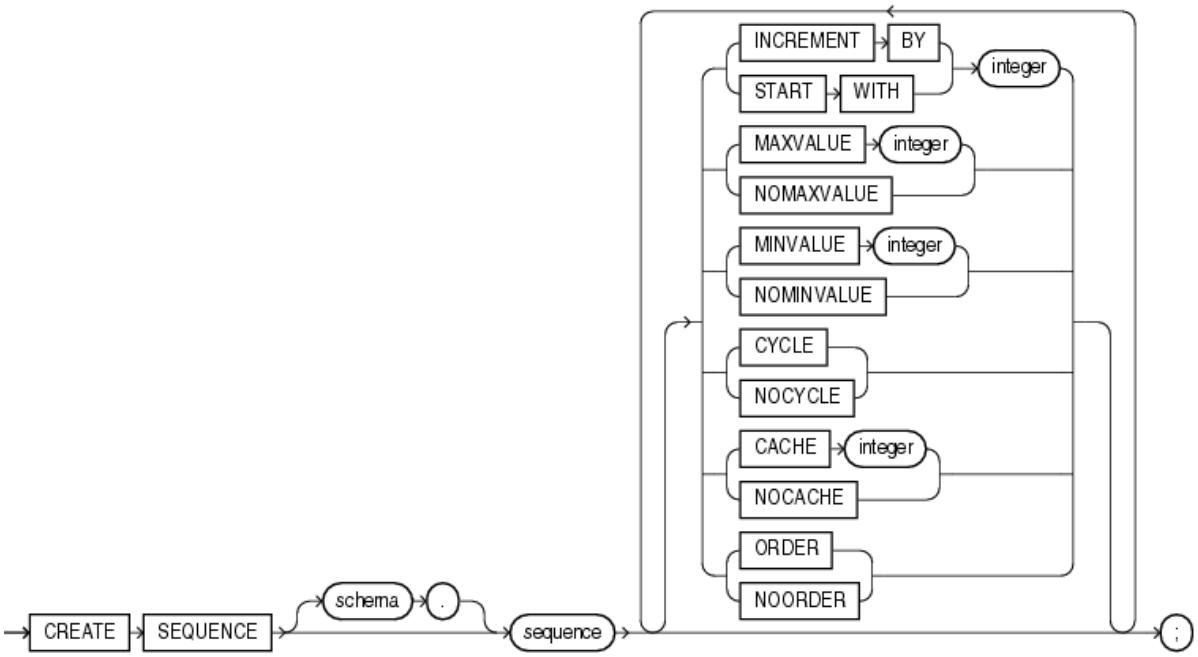
Prerequisites

To create a sequence in your own schema, you must have the **CREATE SEQUENCE** system privilege.

To create a sequence in another user's schema, you must have the **CREATE ANY SEQUENCE** system privilege.

Syntax

***create_sequence*::=**



Semantics

schema

Specify the schema to contain the sequence. If you omit ***schema***, then Oracle Database creates the sequence in your own schema.

sequence

Specify the name of the sequence to be created.

If you specify none of the following clauses, then you create an ascending sequence that starts with 1 and increases by 1 with no upper limit. Specifying only **INCREMENT BY** -1 creates a descending sequence that starts with -1 and decreases with no lower limit.

- To create a sequence that increments without bound, for ascending sequences, omit the **MAXVALUE** parameter or specify **NOMAXVALUE**. For descending sequences, omit the **MINVALUE** parameter or specify the **NOMINVALUE**.
- To create a sequence that stops at a predefined limit, for an ascending sequence, specify a value for the **MAXVALUE** parameter. For a descending sequence, specify a value for the **MINVALUE** parameter. Also specify **NOCYCLE**. Any attempt to generate a sequence number once the sequence has reached its limit results in an error.
- To create a sequence that restarts after reaching a predefined limit, specify values for both the **MAXVALUE** and **MINVALUE** parameters. Also specify **CYCLE**. If you do not specify **MINVALUE**, then it defaults to **NOMINVALUE**, which is the value 1.

INCREMENT BY Specify the interval between sequence numbers. This integer value can be any positive or negative integer, but it cannot be 0. This value can have 28 or fewer digits. The absolute of this value must be less than the difference

of **MAXVALUE** and **MINVALUE**. If this value is negative, then the sequence descends. If the value is positive, then the sequence ascends. If you omit this clause, then the interval defaults to 1.

START WITH: Specify the first sequence number to be generated. Use this clause to start an ascending sequence at a value greater than its minimum or to start a descending sequence at a value less than its maximum. For ascending sequences, the default value is the minimum value of the sequence. For descending sequences, the default value is the maximum value of the sequence. This integer value can have 28 or fewer digits.

Note:
This value is not necessarily the value to which an ascending cycling sequence cycles after reaching its maximum or minimum value.

MAXVALUE Specify the maximum value the sequence can generate. This integer value can have 28 or fewer digits. **MAXVALUE** must be equal to or greater than**START WITH** and must be greater than **MINVALUE**.

NOMAXVALUE: Specify **NOMAXVALUE** to indicate a maximum value of 10^{27} for an ascending sequence or -1 for a descending sequence. This is the default.

MINVALUE Specify the minimum value of the sequence. This integer value can have 28 or fewer digits. **MINVALUE** must be less than or equal to **START WITH** and must be less than **MAXVALUE**.

NOMINVALUE: Specify **NOMINVALUE** to indicate a minimum value of 1 for an ascending sequence or -10^{26} for a descending sequence. This is the default.

CYCLE: Specify **CYCLE** to indicate that the sequence continues to generate values after reaching either its maximum or minimum value. After an ascending sequence reaches its maximum value, it generates its minimum value. After a descending sequence reaches its minimum, it generates its maximum value.

NOCYCLE: Specify **NOCYCLE** to indicate that the sequence cannot generate more values after reaching its maximum or minimum value. This is the default.

CACHE Specify how many values of the sequence the database preallocates and keeps in memory for faster access. This integer value can have 28 or fewer digits. The minimum value for this parameter is 2. For sequences that cycle, this value must be less than the number of values in the cycle. You cannot cache more values than will fit in a given cycle of sequence numbers. Therefore, the maximum value allowed for **CACHE** must be less than the value determined by the following formula:

```
(CEIL (MAXVALUE - MINVALUE)) / ABS (INCREMENT)
```

If a system failure occurs, then all cached sequence values that have not been used in committed DML statements are lost. The potential number of lost values is equal to the value of the **CACHE** parameter.

Note:
Oracle recommends using the **CACHE** setting to enhance performance if you are using sequences in an Oracle Real Application Clusters environment.

NOCACHE: Specify **NOCACHE** to indicate that values of the sequence are not preallocated. If you omit both **CACHE** and **NOCACHE**, then the database caches 20 sequence numbers by default.

ORDER Specify **ORDER** to guarantee that sequence numbers are generated in order of request. This clause is useful if you are using the sequence numbers as timestamps. Guaranteeing order is usually not important for sequences used to generate primary keys.

ORDER is necessary only to guarantee ordered generation if you are using Oracle Real Application Clusters. If you are using exclusive mode, then sequence numbers are always generated in order.

NOORDER: Specify **NOORDER** if you do not want to guarantee sequence numbers are generated in order of request. This is the default.

Example

Creating a Sequence: Example The following statement creates the sequence **customers_seq** in the sample schema **oe**. This sequence could be used to provide customer ID numbers when rows are added to the **customers** table.

```
CREATE SEQUENCE customers_seq
 START WITH      1000
 INCREMENT BY  1
 NOCACHE
 NOCYCLE;
```

The first reference to **customers_seq.nextval** returns 1000. The second returns 1001. Each subsequent reference will return a value 1 greater than the previous reference.

View – Index

Overview of Views

A **view** is a logical representation of one or more tables. In essence, a view is a stored query. A view derives its data from the tables on which it is based, called **base tables**.

Base tables can be tables or other views. All operations performed on a view actually affect the base tables. You can use views in most places where tables are used.

Note:
Materialized views use a different data structure from standard views.

Views enable you to tailor the presentation of data to different types of users. Views are often used to:

- Provide an additional level of table security by restricting access to a predetermined set of rows or columns of a table

  For example, Figure 4-6 shows how the staff view does not show the salary or commission_pct columns of the base table employees.

- Hide data complexity

  For example, a single view can be defined with a **join**, which is a collection of related columns or rows in multiple tables. However, the view hides the fact that this information actually originates from several tables. A query might also perform extensive calculations with table information. Thus, users can query a view without knowing how to perform a join or calculations.

- Present the data in a different perspective from that of the base table

  For example, the columns of a view can be renamed without affecting the tables on which the view is based.

- Isolate applications from changes in definitions of base tables

  For example, if the defining query of a view references three columns of a four column table, and a fifth column is added to the table, then the definition of the view is not affected, and all applications using the view are not affected.

For an example of the use of views, consider the hr.employees table, which has several columns and numerous rows. To allow users to see only five of these columns or only specific rows, you could create a view as follows:

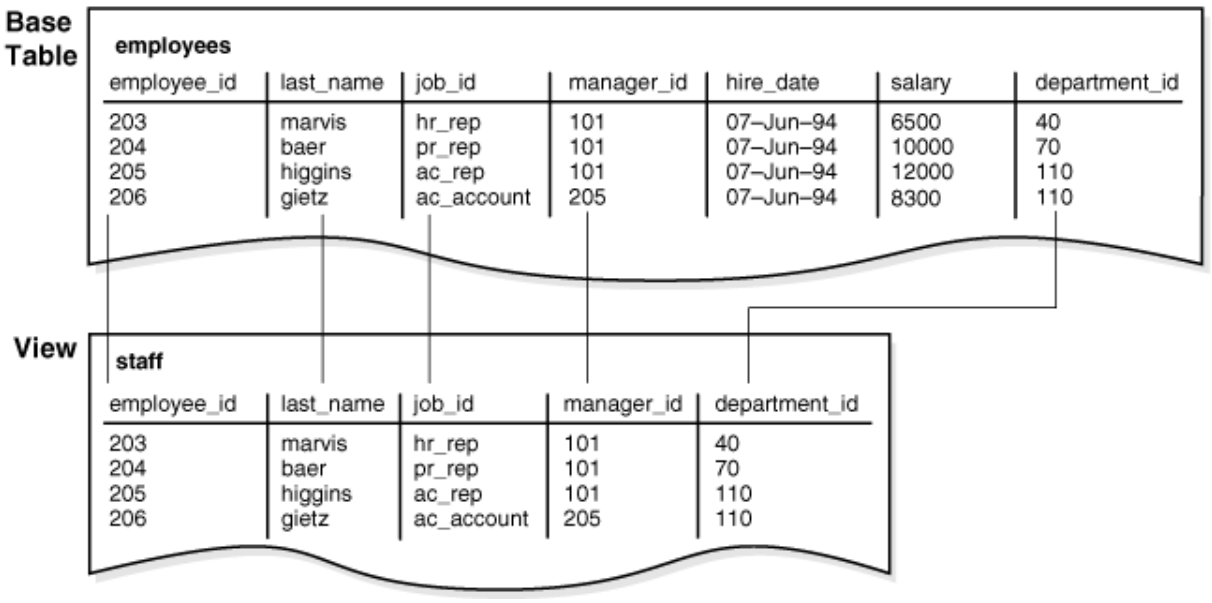
```
CREATE VIEW staff AS

SELECT employee_id, last_name, job_id, manager_id, department_id

FROMemployees;
```

As with all **subqueries**, the query that defines a view cannot contain the FOR UPDATE clause. Figure 4-6 graphically illustrates the view named staff. Notice that the view shows only five of the columns in the base table.

Figure 4-6 View



Characteristics of Views

Unlike a table, a view is not allocated storage space, nor does a view contain data. Rather, a view is defined by a query that extracts or derives data from the base tables referenced by the view. Because a view is based on other objects, it requires no storage other than storage for the query that defines the view in the **data dictionary**.

A view has dependencies on its referenced objects, which are automatically handled by the database. For example, if you drop and re-create a base table of a view, then the database determines whether the new base table is acceptable to the view definition.

Data Manipulation in Views

Because views are derived from tables, they have many similarities. For example, a view can contain up to 1000 columns, just like a table. Users can query views, and with some restrictions they can perform DML on views. Operations performed on a view affect data in some base table of the view and are subject to the integrity constraints and triggers of the base tables.

The following example creates a view of the hr.employees table:

```
CREATE VIEW staff_dept_10 AS

SELECT employee_id, last_name, job_id,

        manager_id, department_id

FROM employees

WHERE department_id = 10
```

```
WITH CHECK OPTION CONSTRAINT staff_dept_10_cnst;
```

The defining query references only rows for department 10. The CHECK OPTION creates the view with a constraint so that INSERT and UPDATEstatements issued against the view cannot result in rows that the view cannot select. Thus, rows for employees in department 10 can be inserted, but not rows for department 30.

How Data Is Accessed in Views

Oracle Database stores a view definition in the data dictionary as the text of the query that defines the view. When you reference a view in a SQL statement, Oracle Database performs the following tasks:

1. Merges a query (whenever possible) against a view with the queries that define the view and any underlying views

   Oracle Database optimizes the merged query as if you issued the query without referencing the views. Therefore, Oracle Database can use indexes on any referenced base table columns, whether the columns are referenced in the view definition or in the user query against the view.

   Sometimes Oracle Database cannot merge the view definition with the user query. In such cases, Oracle Database may not use all indexes on referenced columns.

2. Parses the merged statement in a **shared SQL area**

   Oracle Database parses a statement that references a view in a new shared SQL area *only* if no existing shared SQL area contains a similar statement. Thus, views provide the benefit of reduced memory use associated with shared SQL.

3. Executes the SQL statement

The following example illustrates data access when a view is queried. Assume that you create employees_view based on the employees anddepartments tables:

```
CREATE VIEW employees_view AS

SELECT employee_id, last_name, salary, location_id

FROM employees JOIN departments USING (department_id)

WHERE department_id = 10;
```

A user executes the following query of employees_view:

```
SELECT last_name

FROM employees_view
```

```
WHERE employee_id = 200;
```

Oracle Database merges the view and the user query to construct the following query, which it then executes to retrieve the data:

```
SELECT last_name

FROM employees, departments

WHERE employees.department_id = departments.department_id

AND departments.department_id = 10

AND employees.employee_id = 200;
```

Difference between OODBMS and DBMS

An object-oriented database management system (OODBMS), sometimes shortened to *ODBMS* for *object database management system*), is a database management system (DBMS) that supports the modelling and creation of data as objects. This includes some kind of support for classes of objects and the inheritance of class properties and methods by subclasses and their objects. There is currently no widely agreed-upon standard for what constitutes an OODBMS, and OODBMS products are considered to be still in their infancy. In the meantime, the object-relational database management system (ORDBMS), the idea that object-oriented database concepts can be superimposed on relational databases, is more commonly encountered in available products. An object-oriented database interface standard is being developed by an industry group, the Object Data Management Group (ODMG). The Object Management Group (OMG) has already standardized an object-oriented data brokering interface between systems in a network.

Concepts of Object oriented programming

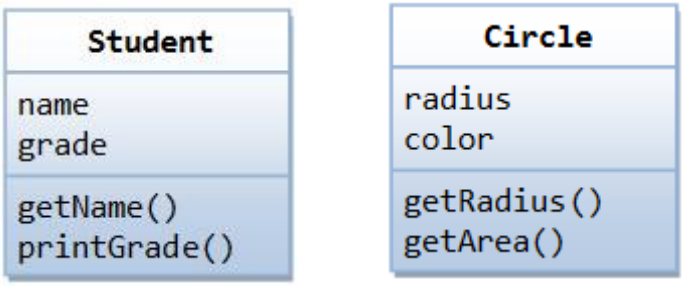
The general concepts of OOPS comprises the following.

1. Object
2. Class
3. Data abstraction
4. Inheritance
5. Polymorphism
6. Dynamic Binding
7. Message passing.

1. Object

Object is an entity that can store data and, send and receive messages. They are runtime entities; they may represent a person, a place a bank account, a table of data or any item that the program must handle. It is an instance of a class.

They may also represent user-defined data such as vectors, time and lists. When a program is executed, the object interacts by sending messages to one another. Each object contain data and code to manipulate the data objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted and the type of response returned by the objects.



2. Classes

A class is a collection of objects of similar type. Classes are user defined data types and behave like the built in types of a programming language. For example mango, apple and orange are members of the class fruit. Then the statement FRUIT MANGO; will create an object mango belonging to the class fruit. The syntax used to create an object is no different than the syntax used to create an integer object in C. If **fruit** has been defined as a class, then the statement

fruit mango;

will create an object **mango** belonging to the class **fruit**.

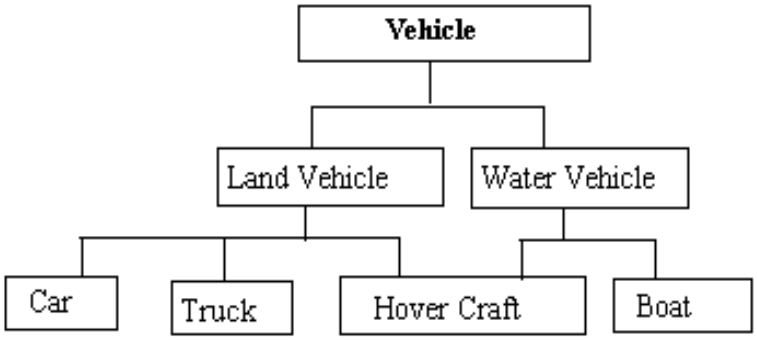
3. Data abstraction and encapsulation:

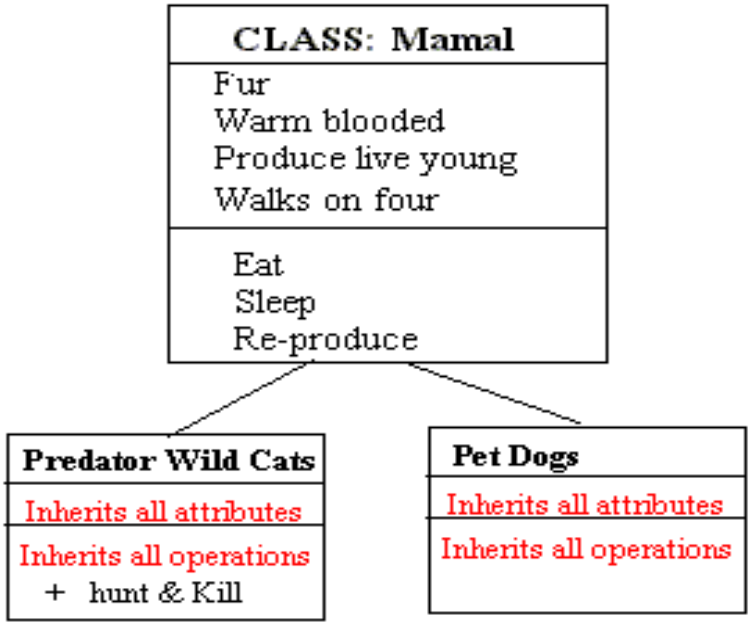
The wrapping up of data and its functions into a single unit (class) is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapped in the class can assess it> these functions provide the interface between the objects data and the program> this insulation of data from direct access by the program is called **DATA HIDING (or data abstraction).** Since the classes use the concept of data abstraction they are known as ABSTRACT DATA TYPES (ADT)

4. Inheritance:

In heritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. For example the bird **robin** is a part of the class **flying birds** which again a part of **bird**. As given in the diagram below each derived class shares common characteristics with the class from which it is derived.

In **OOP**, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly, what he wants.

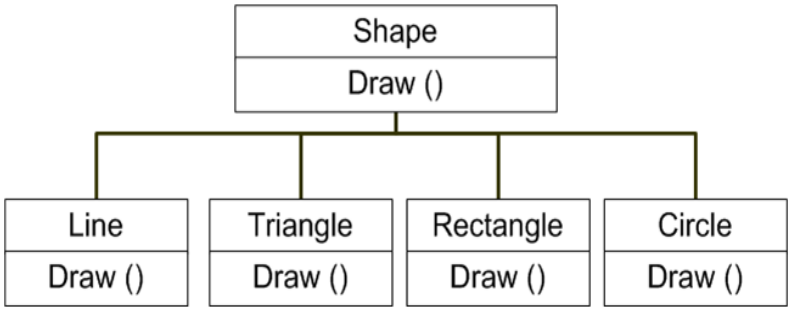




5. Polymorphism:

Polymorphism means the ability to take more than one form. For example an operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation. For example consider the operation addition. For two numbers, the operation will generate a sum . if the operands are strings, then the operation would produce a third string by concatenation.

Here in the below given diagram a single function draw () does different operation according to the behavior of the type derived. I.e. Draw () function works in different form.



Polymorphism plays an important role in allowing objects having internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in inheritance.

6. Dynamic Binding

Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call is associated with a polymorphic reference depends on the dynamic type of that reference.

7. Message Communication

An object oriented program consists of a set of objects that communicate with each other. Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

Advantages of Object orientation

- The complexity of software can be managed easily.
- Data hiding concept help the programmer to build secure programs
- Through the class concept we can define the user defined data type
- The inheritance concept can be used to eliminate redundant code
- The message-passing concept helps the programmer to communicate between different objects.
- New data and functions can be easily added whenever necessary.
- OOPS ties data elements more closely to the functions that operates on.

Object Views

Just as a view is a virtual table, an **object view** is a virtual object table. Each row in the view is an **object**, which is an instance of an **object type**. An object type is a user-defined data type.

You can retrieve, update, insert, and delete relational data as if it was stored as an object type. You can also define views with columns that are object data types, such as objects, REFs, and collections (nested tables and VARRAYs).

Like relational views, object views can present only the data that you want users to see. For example, an object view could present data about IT programmers but omit sensitive data about salaries. The following example creates an employee_type object and then the view it_prog_viewbased on this object:

```
CREATE TYPE employee_type AS OBJECT

(

 employee_id  NUMBER (6),

 last_name    VARCHAR2 (25),

 job_id       VARCHAR2 (10)

);

/
```

```
CREATE VIEW it_prog_view OF employee_type

 WITH OBJECT IDENTIFIER (employee_id) AS

SELECT  e.employee_id, e.last_name, e.job_id

FROM    employees e

WHERE   job_id = 'IT_PROG';
```

Object views are useful in prototyping or transitioning to object-oriented applications because the data in the view can be taken from relational tables and accessed as if the table were defined as an object table. You can run object-oriented applications without converting existing tables to a different physical structure.

Collection Types

Each collection type describes a data unit made up of an indefinite number of elements, all of the same datatype. The collection types are **array types**and **table types**.

Array types and table types are schema objects. The corresponding data units are called **VARRAYs** and **nested tables**. When there is no danger of confusion, we often refer to the collection types as **VARRAY**s and nested tables.

Collection types have constructor methods. The name of the constructor method is the name of the type, and its argument is a comma separated list of the new collection's elements. The constructor method is a function. It returns the new collection as its value.

An expression consisting of the type name followed by empty parentheses represents a call to the constructor method to create an empty collection of that type. An empty collection is different from a null collection.

ARRAYs

An **array** is an ordered set of data **elements**. All elements of a given array are of the same datatype. Each element has an **index**, which is a number corresponding to the element's position in the array.

The number of elements in an array is the **size** of the array. Oracle allows arrays to be of variable size, which is why they are called **VARRAY**s. You must specify a maximum size when you declare the array type.

Creating an array type does not allocate space. It defines a datatype, which you can use as:

- The datatype of a column of a relational table
- An object type attribute

- A PL/SQL variable, parameter, or function return type.

A **VARRAY** is normally stored in line; that is, in the same tablespace as the other data in its row. If it is sufficiently large, however, Oracle stores it as a**BLOB**.

Nested Tables

A **nested table** is an unordered set of data **elements**, all of the same datatype. It has a single column, and the type of that column is a built-in type or an object type. If an object type, the table can also be viewed as a multicolumn table, with a column for each attribute of the object type. If compatibility is set to Oracle9*i* or higher, nested tables can contain other nested tables.

A table type definition does not allocate space. It defines a type, which you can use as:

- The datatype of a column of a relational table
- An object type attribute
- A PL/SQL variable, parameter, or function return type

When a table type appears as the type of a column in a relational table or as an attribute of the underlying object type of an object table, Oracle stores all of the nested table data in a single table, which it associates with the enclosing relational or object table.

A convenient way to access the elements of a nested table individually is to use a nested cursor.