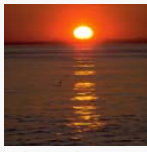
# Chapter 9: Object-Based Databases

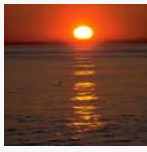
Database System Concepts





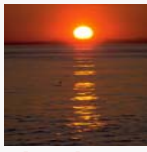
# Chapter 9: Object-Based Databases

- Complex Data Types and Object Orientation
- Structured Data Types and Inheritance in SQL
- Table Inheritance
- Array and Multiset Types in SQL
- Object Identity and Reference Types in SQL
- Implementing O-R Features
- Persistent Programming Languages
- Comparison of Object-Oriented and Object-Relational Databases



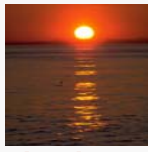
# Object-Relational Data Models

- Extend the relational data model by including object orientation and constructs to deal with added data types.
- Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- Upward compatibility with existing relational languages.



# Complex Data Types

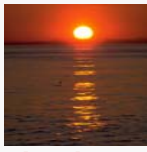
- Motivation:
  - Permit non-atomic domains (atomic ≡ indivisible)
  - Example of non-atomic domain:  set of integers, or set of tuples
  - Allows more intuitive modeling for applications with complex data
- Intuitive definition:
  - Allow relations whenever we allow atomic (scalar) values — relations within relations
  - Retains mathematical foundation of relational model
  - Violates first normal form.



# Example of Complex Data Types

- **Addresses are composite attributes**
  - composed of State, City, Street, Postal Code
- If not split but represented as String
  - we might not use these components in queries
- If split among tables
  - More complex queries since we would have to mention each subfield apart
- Solution
  - Structured data types

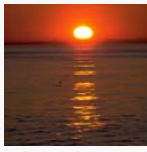
- **Phone numbers are multivalued attributes**
  - People have more than one phone number
  - Normalization with new relation is expensive and artificial



# Example of a Nested Relation

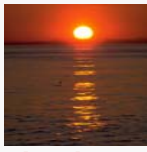
- Example:  library information system
- Each book has
  - title,
  - a set of authors,
  - publisher, and
  - a set of keywords
- Non-1NF relation *books*

| *title* | *author-set* | *publisher* | *keyword-set* |
|---|---|---|---|
| | | (*name*, *branch*) | |
| Compilers | {Smith, Jones} | (McGraw-Hill, New York) | {parsing, analysis} |
| Networks | {Jones, Frick} | (Oxford, London) | {Internet, Web} |



# 4NF Decomposition of Nested Relation

- Remove awkwardness of *flat-books* by assuming that the following multivalued dependencies hold:
  - *title* $\twoheadrightarrow$ *author*
  - *title* $\twoheadrightarrow$ *keyword*
  - *title* $\twoheadrightarrow$ *pub-name, pub-branch*
- Decompose *flat-doc* into 4NF using the schemas:
  - (*title, author* )
  - (*title, keyword* )
  - (*title, pub-name, pub-branch* )



# 4NF Decomposition of *flat–books*

| *title* | *author* |
|---|---|
| Compilers | Smith |
| Compilers | Jones |
| Networks | Jones |
| Networks | Frick |

*authors*

| *title* | *keyword* |
|---|---|
| Compilers | parsing |
| Compilers | analysis |
| Networks | Internet |
| Networks | Web |

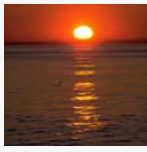
*keywords*

| *title* | *pub-name* | *pub-branch* |
|---|---|---|
| Compilers | McGraw-Hill | New York |
| Networks | Oxford | London |

*books4*

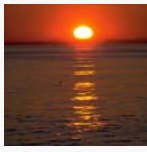
# Problems with 4NF Schema

- 4NF design requires users to **include joins** in their queries.
- 1NF relational view *flat-books* defined by join of 4NF relations:
  - loses the **one-to-one** correspondence between tuples and documents, (book is the document, spread among tuples)
  - And has a large amount of **redundancy**
- Nested relations representation is much more natural here.
- A simple, easy-to-use interface requires a one-to-one correspondence between the user's intuitive **notion of an object** and the database system's **notion of data item**.



# Complex Types and SQL:1999

- Extensions to SQL to support complex types include:
  - Collection and large object types
    - Nested relations are an example of collection types
  - Structured types
    - Nested record structures like composite attributes
  - Inheritance
  - Object orientation
    - Including object identifiers and references
- Our description is mainly based on the SQL:1999 standard
  - **Not fully** implemented in any database system currently
  - But some features are present in each of the major commercial database systems
    - Read the manual of your database system to see what it supports



# Structured Types and Inheritance in SQL

- Structured types can be declared and used in SQL

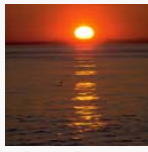
```
create type Name as
    (firstname      varchar(20),
     lastname       varchar(20))
    final

create type Address as
    (street         varchar(20),
     city           varchar(20),
     zipcode        varchar(20))
    not final
```

  - Note: **final** and **not final** indicate whether subtypes can be created

- Structured types can be used to create tables with composite attributes

```
create table customer (
    name        Name,
    address     Address,
    dateOfBirth date)
```

- Dot notation used to reference components: *name.firstname*



# Structured Types (cont.)

- User-defined row types

  **create type** *CustomerType* **as** (

  *name Name,*

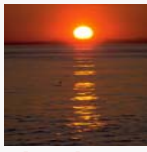
  *address Address,*

  *dateOfBirth* **date**)

  **not final**

- Can then create a table whose rows are a user-defined type

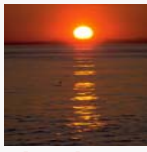
  **create table** *customer* **of** *CustomerType*



# Unnamed row type

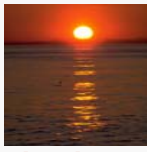
- Another alternative to defining composite attributes in SQL is to use the unnamed **row type**.

- **create table** customer_r (
  name **row** (firstname **varchar**(20),
  lastname **varchar**(20))
  address **row** (street **varchar**(20),
  city **varchar**(20),
  zipcode **varchar**(20)),
  dateOfBirth **date**)



# Access of component attributes

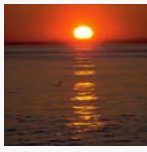
- DOT notation
- **select** name.lastname, address.city
  **from** customer
- **select** name.firstname, name.lastname
  **from** customer



# Methods

- A structured type can have methods defined on it.

  **create type** *CustomerType* **as** (
  *name Name,*
  *address Address,*
  *dateOfBirth* **date**)
  **not final**
  **method** *ageOnDate* (*onDate* **date**)
  **returns interval year**



# Methods

- Method body is given separately.

  **create instance method** *ageOnDate* (*onDate* **date**)

  **returns interval year**

  **for** *CustomerType*

  **begin**

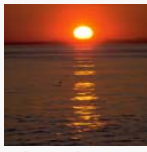
  **return** *onDate* - **self**.*dateOfBirth*;

  **end**

- We can now find the age of each customer:

  **select** *name.lastname, ageOnDate* (**current_date**)

  **from** *customer*

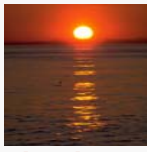


# Constructor function

- In SQL:1999 constructor functions are used to create values of structured types. A function with the same name as a structured type is a constructor function for the structured type.

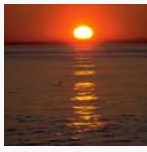
- **create function** Name (firstname **varchar**(20), lastname **varchar**(20))
  **returns** Name
  **begin**
    **set self**.firstname = firstname;
    **set self**.lastname = lastname;
  **end**

- To create a value of the type Name
  **new** Name('John', 'Smith')



# Default constructors

- By default every structured type has a **constructor with no arguments**, which sets the attributes to their default values.
- Any other constructors have to be defined explicitly.
- There can be **more than one constructor** for the structured type, although they have the same name. they must be distinguishable by the number of arguments and types of their arguments.
  - OO-Programming technique



# Insertion

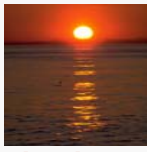
- **insert into** Customer

  **values**

  (**new** Name(‘John’, ‘Smith’),

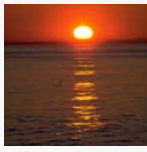
  **new** Address(‘Rr. Komuna e Parisit’, ‘Tirana’, ‘1000’)

  **date** ‘1990-2-22’)



# Inheritance

- Software Engineering
- In object-oriented programming (OOP), **inheritance** is a way to form new classes (instances of which are called objects) using classes that have already been defined.
- Inheritance is employed to help **reuse** existing code with little or no modification.



# Inheritance

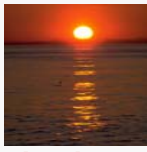
- Suppose that we have the following type definition for people:

  **create type** *Person*
  (*name* **varchar**(20),
  *address* **varchar**(20))

- Using inheritance to define the student and teacher types

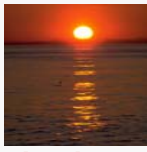
  **create type** *Student*
  **under** *Person*
  (*degree* **varchar**(20),
  *department* **varchar**(20))
  **create type** *Teacher*
  **under** *Person*
  (*salary* **integer**,
  *department* **varchar**(20))

- The SQL standard also requires an extra field at the end of the type definition whose value is either **final** or **not final.** Keyword **final** says that subtypes may not be created from the given type.



# Inheritance and methods

- Methods of a structured type are inherited by its subtypes just as attributes are.
- Subtypes can redefine methods by using **overriding method** in place of **method** in the method declaration



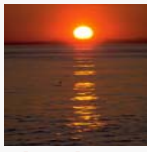
# Multiple Inheritance

- SQL:1999 and SQL:2003 do not support multiple inheritance
- If our type system supports multiple inheritance, we can define a type for teaching assistant as follows:

  **create type** *Teaching Assistant*
  **under** *Student, Teacher*

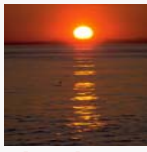
- To avoid a conflict between the two occurrences of *department* we can rename them

  **create type** *Teaching Assistant*
  **under**
  *Student* **with** (*department* **as** *student_dept* ),
  *Teacher* **with** (*department* **as** *teacher_dept* )



# Consistency Requirements for subtypes

- In SQL as in most other languages a value of a structured type must have exactly one "**most-specific type**".
  - That is each value must be associated with one specific type called its **most-specific type.**
- By means of inheritance it is also associated with each of the supertypes of its most specific type.
- Suppose an entity has the type Person and type Student.
  - The **most specific type** of the entity is **Student**.



# Table inheritance

- Subtables in SQL:1999 correspond to the E-R notion of specialization/generalization.
- For instance, suppose we define the *people* table as follows:

  **create table** *people* **of** *Person*

- We can then define tables *students* and *teachers* as **subtable**s of *people*, as follows:

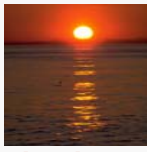
  **create table** *students* **of** *Student*

  **under** *people*

  **create table** *teachers* **of** *Teacher*

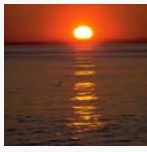
  **under** *people*

- **The types of the subtables must be subtypes of the type of the parent table.**
  - **Thereby, every attribute present in *people* is also present in the subtables.**



# Table inheritance

- Further, when we declare *students* and *teachers* as subtables of *people*, every tuple present in *students* or *teachers* becomes also implicitly present in *people*.
- Thus, if a query uses the table *people*, it will find not only tuples directly inserted into that table, but also tuples inserted into its subtables, namely *students* and *teachers*.
- **However, only those attributes that are present in *people* can be accessed.**



# Multiple Inheritance

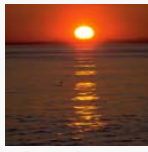
- Multiple inheritance is possible with tables, just as it is possible with types. (We note, however, that multiple inheritance of tables is not supported by SQL:1999.) For example, we can create a table of type *TeachingAssistant*:

  **create table** *teaching-assistants*

  **of** *TeachingAssistant*

  **under** *students*, *teachers*

- As a result of the declaration, every tuple present in the *teaching-assistants* table is also implicitly present in the *teachers* and in the *students* table, and in turn in the *people* table.
- SQL:1999 permits us to find tuples that are in *people* but not in its subtables by using "**only** *people*" in place of *people* in a query.



# Delete operation

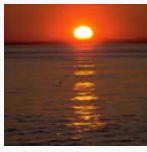
- With keyword “only” we can delete only tuples from the parent table

  **Delete from** people **where** P

  - Would delete tuples from the parent table and the inherited tables.

- Substituting “people” with “only people”

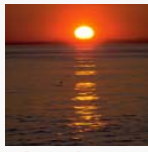
  **Delete from only** people **where** P



# Consistency Requirements for Subtables

- Consistency requirements on subtables and supertables.
  - Each tuple of the supertable (e.g. *people)* can correspond to at most one tuple in each of the subtables (e.g. *students* and *teachers)*
    - *Without this condition we would have two tuples in student corresponding to the same person.*
  - Additional constraint in SQL:1999:

    All tuples corresponding to each other (that is, with the same values for inherited attributes) must be derived from one tuple (inserted into one table).
    - That is, each entity must have a most specific type
    - We cannot have a tuple in *people* corresponding to a tuple each in *students* and *teachers*

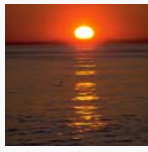


# Array and Multiset Types in SQL

- Array types were added with SQL:1999, while multiset types were added with SQL:2003.
- **Array** is an **ordered** collection, while a **multiset** is an **unordered** one.
  - Multisets are like sets except that a set allows each element to occur at most once.
- Example of array and multiset declaration:

```
create type Publisher as
    (name            varchar(20),
     branch          varchar(20))
create type Book as
    (title           varchar(20),
     author-array    varchar(20) array [10],
     pub-date        date,
     publisher       Publisher,
     keyword-set     varchar(20) multiset )

create table books of Book
```

- Similar to the nested relation books, but with array of authors instead of set



# Creation of Collection Values

- Array construction

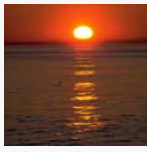
  **array** [‘Silberschatz’,`Korth’,`Sudarshan’]

- Multisets
  - **multiset** [‘computer’, ‘database’, ‘SQL’]
- To create a tuple of the type defined by the *books* relation:

  (‘Compilers’, **array**[`Smith’,`Jones’],
  *Publisher* (`McGraw-Hill’,`New York’),
  **multiset** [`parsing’,`analysis’ ])
- To insert the preceding tuple into the relation *books*

  **insert into** *books*
  **values**
  (‘Compilers’, **array**[`Smith’,`Jones’],
  *Publisher* (`McGraw-Hill’,`New York’),
  **multiset** [`parsing’,`analysis’ ])

- We can access or update elements of an array by specifying the array index, for example author-array[1].



# Querying Collection-Valued Attributes

- To find all books that have the word “database” as a keyword,

  **select** *title*
  **from** *books*
  **where** ‘database’ **in** (**unnest**(*keyword-set* ))

- We can access individual elements of an array by using indices
  - E.g.: If we know that a particular book has three authors, we could write:

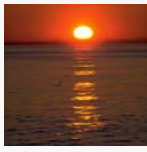
    **select** *author-array*[1], *author-array*[2], *author-array*[3]
    **from** *books*
    **where** *title* = `Database System Concepts’

- To get a relation containing pairs of the form “title, author-name” for each book and each author of the book

  **select** *B.title, A.author*
  **from** *books* **as** *B*, **unnest** (*B.author-array*) **as** *A* (*author* )

- To retain ordering information we add a **with ordinality** clause

  **select** *B.title, A.author, A.position*
  **from** *books* **as** *B*, **unnest** (*B.author-array*) **with ordinality as**
  *A* (*author, position* )



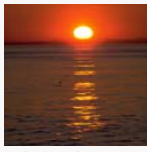
# Unnesting

- The transformation of a nested relation into a form with fewer (or no) relation-valued attributes is called **unnesting**.
- E.g.

**select** *title*, *A* **as** *author*, *publisher.name* **as** *pub_name*,
*publisher.branch* **as** *pub_branch*, *K.keyword*

**from** *books* **as** *B*, **unnest**(*B.author_array* ) **as** *A* (*author* ),

**unnest** (*B.keyword_set* ) **as** *K* (*keyword* )

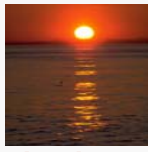


# 1NF Version of Nested Relation

1NF version of *books*

| *title* | *author* | *pub-name* | *pub-branch* | *keyword* |
|---|---|---|---|---|
| Compilers | Smith | McGraw-Hill | New York | parsing |
| Compilers | Jones | McGraw-Hill | New York | parsing |
| Compilers | Smith | McGraw-Hill | New York | analysis |
| Compilers | Jones | McGraw-Hill | New York | analysis |
| Networks | Jones | Oxford | London | Internet |
| Networks | Frick | Oxford | London | Internet |
| Networks | Jones | Oxford | London | Web |
| Networks | Frick | Oxford | London | Web |

*flat-books*



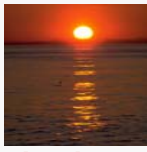
# Nesting

- **Nesting** is the opposite of unnesting, creating a collection-valued attribute
- NOTE: SQL:1999 does not support nesting
- Nesting can be done in a manner similar to aggregation, but using the function **collect**() in place of an aggregation operation, to create a multiset
- To nest the *flat-books* relation on the attribute *keyword*:

  **select** *title*, *author*, *Publisher* (*pub_name, pub_branch* ) **as** *publisher*,
  **collect** (*keyword*) **as** *keyword_set*
  **from** *flat-books*
  **groupby** *title, author, publisher*

- To nest on both authors and keywords:

  **select** *title*, **collect** (*author* ) **as** *author_set*,
  *Publisher* (*pub_name, pub_branch*) **as** *publisher*,
  **collect** (*keyword* ) **as** *keyword_set*
  **from** *flat-books*
  **group by** *title*, *publisher*



# Object-Identity and Reference Types

- Define a type *Department* with a field *name* and a field *head* which is a reference to the type *Person,* with table *people* as scope:

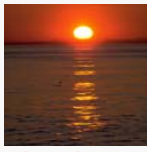
  **create type** *Department* (
  *name* **varchar** (20),
  *head* **ref** (*Person*) **scope** *people*)

- We can then create a table *departments* as follows

  **create table** *departments* **of** *Department*

- We can omit the declaration **scope** people from the type declaration and instead make an addition to the **create table** statement:

  **create table** *departments* **of** *Department*
  (*head* **with options scope** *people*)



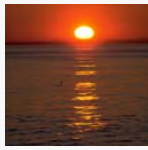
# Reference table

- The reference table must have an attribute that stores the identifier of the tuple.
- We declare this attribute, called the **self-referential attribute**.

**create table** people **of** Person

**ref is** person_id **system generated**

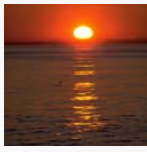
- person_id is an attribute name not a keyword and the identifier is generated automatically by the database system.



# Initializing Reference-Typed Values

- To create a tuple with a reference value, we can first create the tuple with a null reference and then set the reference separately:

```
insert into departments
       values (`CS', null)
update departments
    set head = (select p.person_id
                    from people as p
                     where name = `John')
    where name = `CS'
```



# User Generated Identifiers

- The type of the object-identifier must be specified as part of the type definition of the referenced table, and
- The table definition must specify that the reference is **user generated**

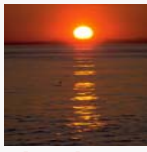
```
create type Person
    (name varchar(20)
     address varchar(20))
   ref using varchar(20)
create table people of Person
  ref is person_id user generated
```

- When creating a tuple, we must provide a unique value for the identifier:

```
insert into people (person_id, name, address ) values
    ('01284567', 'John', `23 Coyote Run')
```

- We can then use the identifier value when inserting a tuple into *departments*
  - Avoids need for a separate query to retrieve the identifier:

```
insert into departments
values(`CS', `02184567')
```



# User Generated Identifiers (Cont.)

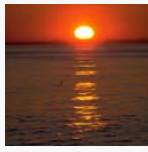
- Can use an existing primary key value as the identifier:

```
create type Person
      (name varchar (20) primary key,
       address varchar(20))
    ref from (name)
create table people of Person
    ref is person_id derived
```

- Note that the reference is **derived**

- When inserting a tuple for *departments*, we can then use

```
insert into departments
    values(`CS',`John')
```



# Path Expressions

- References are dereferenced in SQL:1999 by the -> symbol.
- Find the names and addresses of the heads of all departments:

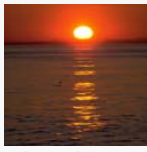
  **select** *head –>name*, *head –>address*
  **from** *departments*

- An expression such as “head–>name” is called a **path expression**
- Path expressions help **avoid explicit joins**
  - If department head were not a reference, a **join of *departments* with *people*** would be required to get at the address
  - Makes expressing the query much easier for the user

- We can use the operation deref to return the tuple pointed to by a reference, and then access its attributes:

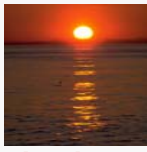
  **select deref**(head).name

  **from** departments



# Implementing O-R Features

- Similar to how E-R features are mapped onto relation schemas
- Multivalued attributes in E-R correspond to multiset-valued attributes in the O-R.
- Composite attributes in E-R roughly correspond to structured types in O-R.
- ISA hierarchies in E-R correspond to table inheritance in the O-R model.

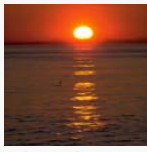


# Implementing O-R Features

- **Subtable implementation**. Subtables can be stored in an efficient manner without the replication of all inherited fields. Two ways:
  - Each table stores primary key (which may be inherited from a parent table) and the attributes are defined locally.
    - Inherited attributes (other than the primary key) do not need to be stored, and can be derived by means of a join with the supertable, based on the primary key.

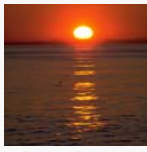
  or

  - Each table stores all inherited and locally defined attributes. When a tuple is inserted, it is stored only in the table in which it is inserted and its presence is inferred in each of the supertables. Access to all attributes of a tuple is faster since a join is not required.



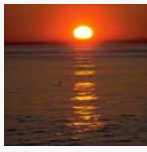
# ODBC and JDBC

- **Open Database Connectivity (ODBC)** provides a standard software API method for using database management systems (DBMS).
  - The designers of ODBC aimed to make it independent of programming languages, database systems, and operating systems.
- JDBC is an API for the Java programming language that defines how a client may access a database.
  - It provides methods for querying and updating data in a database. JDBC is oriented towards relational databases.
- These APIs have been extended to retrieve and store structured types.
- JDBC provides a method getObject() which is similar to getString() but return a Java Struct object, from which the components of the structured type can be extracted.
- It is also possible to associate a Java class with an SQL structured type, and JDBC will then convert between the types.



# JDBC Driver

- A JDBC driver is a software component enabling a Java application to interact with a database.
- To connect with individual databases, JDBC (the Java Database Connectivity API) requires drivers for each database.
- The JDBC driver gives out the connection to the database and implements the protocol for transferring the query and result between client and database.



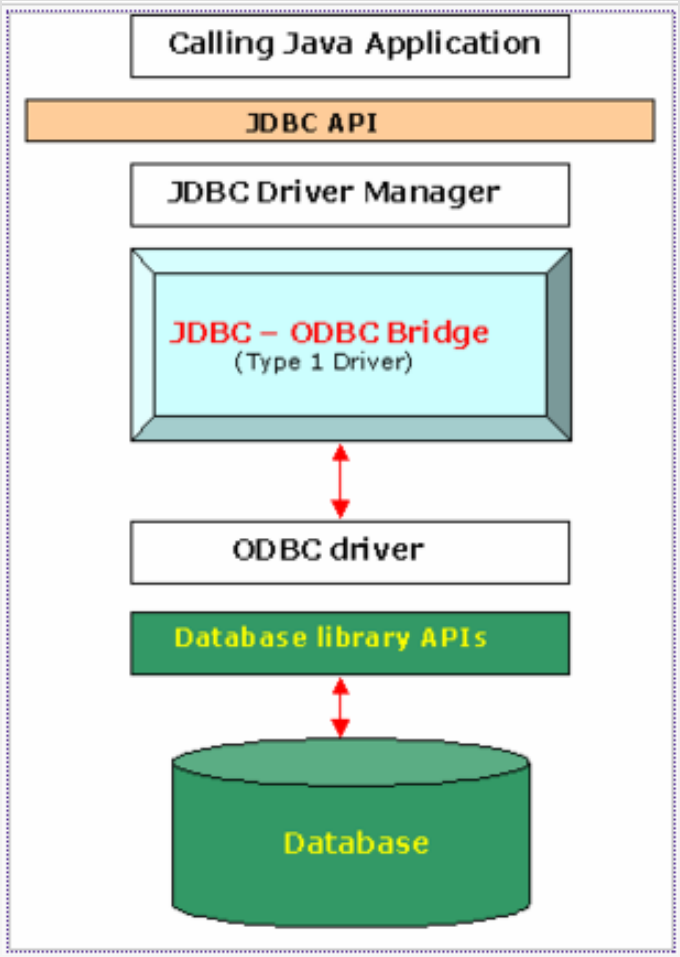
# JDBC Through ODBC

**Advantages**

- Almost any database for which ODBC driver is installed, can be accessed.
- Driver is easy to install.

**Disadvantages**

- Performance overhead since the calls have to go through the JDBC overhead bridge to the ODBC driver, then to the native db connectivity interface.
- The ODBC driver needs to be installed on the client machine.



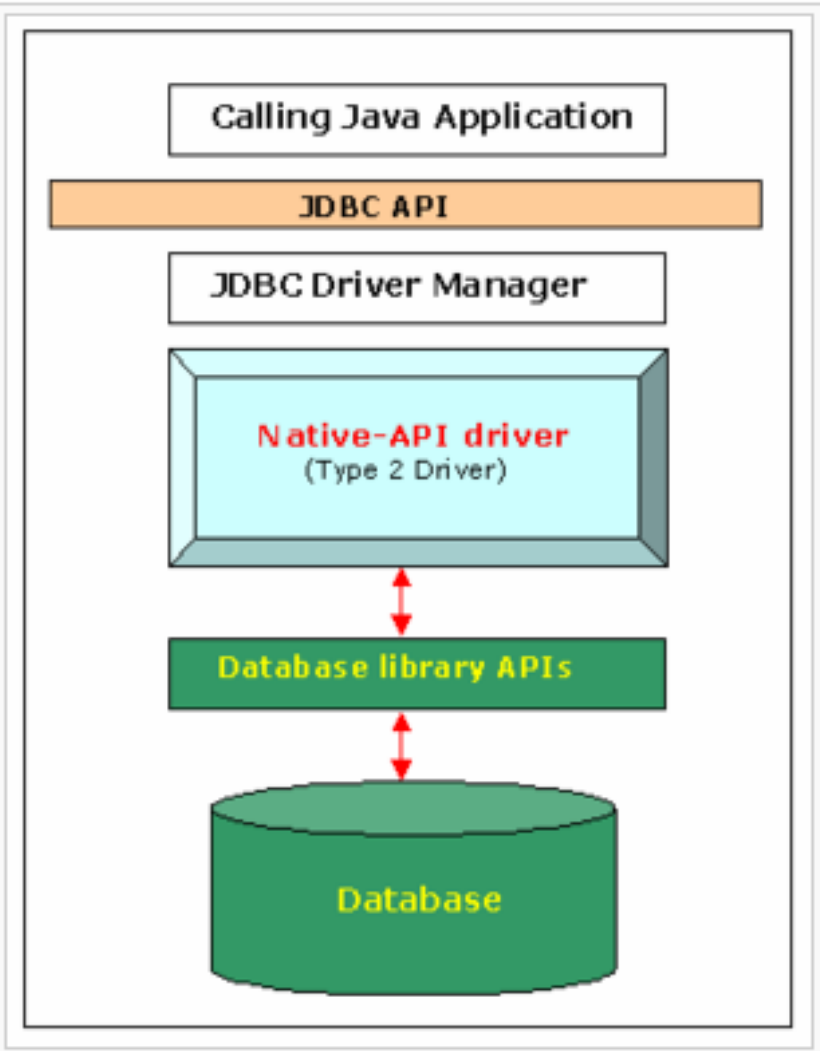
# Native JDBC

**Advantages**

- Better performance than Type 1 Driver (JDBC-ODBC bridge).

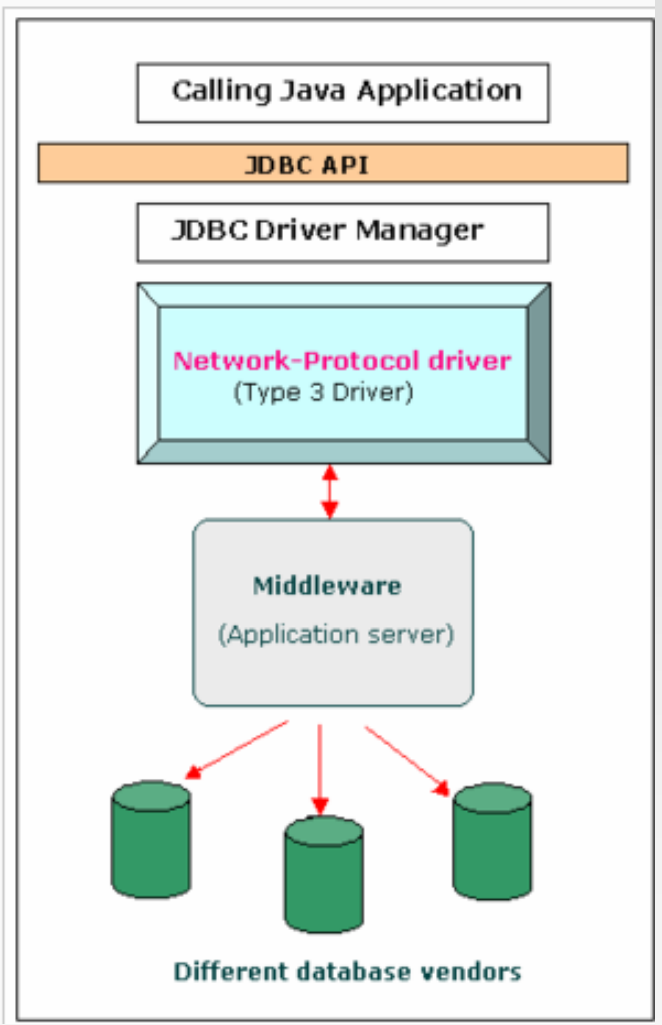
**Disadvantages**

- The vendor client library needs to be installed on the client machine.
- This driver is platform dependent



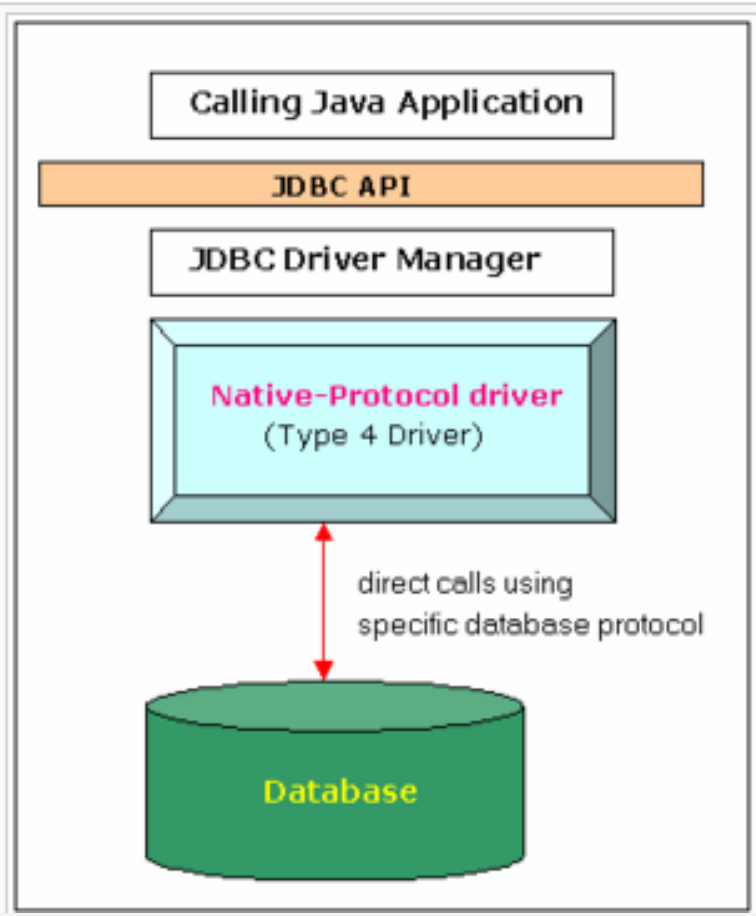
# Pure Java Driver for Database Middleware

- The JDBC type 3 driver, also known as the Pure Java Driver for Database Middleware, is a database driver implementation which makes use of a middle tier between the calling program and the database.
- The middle-tier (application server) converts JDBC calls directly or indirectly into the vendor-specific database protocol.



# JDBC - Native-Protocol Driver

- The JDBC type 4 driver, also known as the Direct to Database Pure Java Driver, is a database driver implementation that converts JDBC calls directly into the vendor-specific database protocol.
- The type 4 driver is written completely in Java and is hence platform independent. It is **installed inside the Java Virtual Machine** of the client.
- It provides better performance over the type 1 and 2 drivers as it does not have the overhead of conversion of calls into ODBC or database API calls.



# JDBC: steps

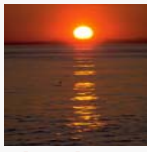
- The method **Class.forName(String)** is used to load the JDBC driver class. The line below causes the JDBC driver from some jdbc vendor to be loaded into the application.

```
Class.forName( "com.somejdbcvendor.TheirJdbcDriver" );
```

- When a Driver class is loaded, it creates an instance of itself and registers it with the DriverManager.

- Now when a connection is needed, one of the DriverManager.getConnection() methods is used to create a JDBC connection.

```
Connection conn = DriverManager.getConnection(
    "jdbc:somejdbcvendor:other data needed by some jdbc vendor",
    "myLogin", "myPassword" );
```

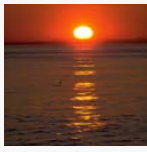


# JDBC: steps

- Once a connection is established, a statement must be created.

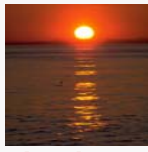
```
Statement stmt = conn.createStatement();
try {
stmt.executeUpdate( "INSERT INTO MyTable( name ) VALUES ( 'my name' ) " );
} finally {
//It's important to close the statement when you are done with it
stmt.close();
}
```

- Note that Connections, Statements, and ResultSets often tie up operating system resources such as sockets or file descriptors.
- In the case of Connections to remote database servers, further resources are tied up on the server, e.g., cursors for currently open ResultSets.
  - **It is vital to close()** any JDBC object as soon as it has played its part; garbage collection should not be relied upon. Forgetting to close() things properly results in spurious errors and misbehaviour.



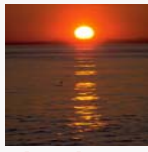
# JDBC: getting the data

```
Statement stmt = conn.createStatement();
try {
    ResultSet rs = stmt.executeQuery( "SELECT * FROM MyTable" );
    try {
        while ( rs.next() ) {
            int numColumns = rs.getMetaData().getColumnCount();
            for ( int i = 1 ; i <= numColumns ; i++ ) {
              // Column numbers start at 1.
              // Also there are many methods on the result set to return
              //  the column as a particular type. Refer to the Sun documentation
              //  for the list of valid conversions.
              System.out.println( "COLUMN " + i + " = " + rs.getObject(i) );
            }
        }
    } finally {
        rs.close();
    }
} finally {
    stmt.close();
}
```



# Persistent Programming Languages

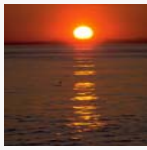
- Languages extended with constructs to handle persistent data
- Programmer can manipulate persistent data directly
  - no need to fetch it into memory and store it back to disk (unlike embedded SQL)
- PPL are different form embedded SQL in two ways:
  - With an embedded language, the type system of the host language usually differs from the type system of the data manipulation language.
  - The programmer using an embedded query language is responsible for writing explicit code to fetch data from the database into memory.



# Persistence of Objects

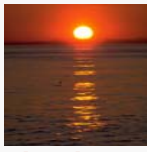
- Objects in OO programming languages are transient
  - When the program terminates, the objects vanish

- Several approaches for persistent objects:
  - **by class** - explicit declaration of persistence for all objects of a class
  - **by creation** - special syntax to create persistent objects
  - **by marking** - make objects persistent after creation. First the object is created as transient, then if it is to be saved, it is marked as persistent.
  - **by reachability** - object is persistent if it is declared explicitly to be so or is reachable from a persistent object
    - All objects referenced by the root persistent objects, are also persistent.



# Object Identity and Pointers

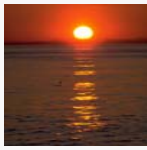
- In OO languages, when an object is created, a **transient object identifier** is created.
- Degrees of permanence of object identity
  - **Intraprocedure**: only during execution of a single procedure
  - **Intraprogram**: only during execution of a single program or query
  - **Interprogram**: across different program executions, with pointers to file-system
    - pointers may change if data-storage format on disk changes
  - **Persistent**: interprogram, plus persistent across data reorganizations



# Storage and access of persistent objects

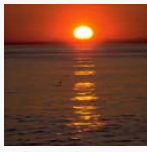
- The data part of an object is stored individually for each object
- The code part is simply stored in files outside the database

- Access of objects
  1. Give names to objects
     - Works for a small number of objects
  2. Expose object identifiers to objects which can be stored externally
  3. Store collections of objects and allow programs to iterate over the collections to find required objects.
     - Collections of objects can themselves be modeled as objects of a *collection type*.



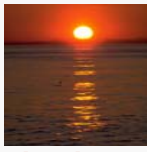
# Class Extent

- **Class extent**
  - A special case of a collection is the ***class extent***, which is the collection of all the objects belonging to the class.
  - In many implementations, there is a ***class extent*** for all the classes that can have persistent objects.



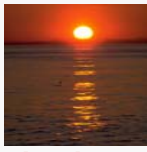
# Persistent C++ and Java

- Persistent versions of C++ and Java have been implemented
  - C++
    - ODMG C++ (Object Data Management Group)
    - ObjectStore
  - Java
    - Java Database Objects (JDO)



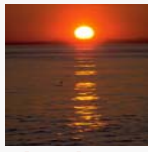
# Persistent C++

- Some persistent C++ implementations support extensions to the C++ syntax.
- **d_** version of many standard types provided: d_Long, d_String
- Persistent pointers
  - A new data type to represent persistent pointers
  - In ODMG C++, there is a template class **d_Ref< T >** to represent pointers to a class T.
  - Template class **d_Set<Class>** used to define sets of objects.
- Creation of persistent objects
  - Overloaded version of new: **new (db) T(),** where **db** identifies the database.
- Class extents
  - These are created and **maintained automatically** for each class
  - ODMG C++ requires the name of the class to be passed as an additional parameter to the new operation.



# Persistent C++

- Relationships
  - Represented by **storing pointers** from each object to the objects it is related with.
- Updates
  - Transparent persistence
  - A function that operates on an object should not know that the object is persistent.
- Query language
  - **Iterators** provide support for simple selection queries. To support more complex queries, persistent C++ systems define a query language.



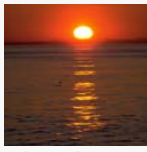
# Built-in Persistent Classes

Class d_Database provides methods to

- open a database: open(databasename)
- give names to objects: set_object_name(object, name)
- look up objects by name: lookup_object(name)
- rename objects: rename_object(oldname, newname)
- close a database (close());

Class d_Object is inherited by all persistent classes.

- provides methods to allocate and delete objects
- method mark_modified() must be called *before* an object is updated.
  - Is automatically called when object is created

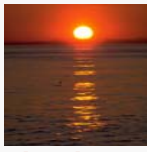


# ODMG C++

```
class Branch  :  public d_Object {
 ....
}
class Person  :  public d_Object {
  public:
    d_String    name;       // should not use String!
    d_String    address;
};
class Account : public d_Object {
  private:
    d_Long      balance;
  public:
    d_Long      number;
    d_Set <d_Ref<Customer>> owners;
    int         find_balance();
    int         update_balance(int delta);
};
```

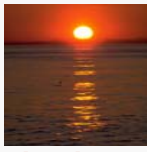
Relationships

```
class Customer  :  public Person {
   public:
     d_Date             member_from;
     d_Long             customer_id;
     d_Ref<Branch> home_branch;
     d_Set <d_Ref<Account>> accounts; };
```



# ODMG C++

```
int create_account_owner(String name, String Address){
    Database bank_db.obj;
    Database * bank_db= & bank_db.obj;
    bank_db =>open("Bank-DB");
    d.Transaction Trans;
    Trans.begin();

    d_Ref<Account> account = new(bank_db) Account;
    d_Ref<Customer> cust = new(bank_db) Customer;
    cust->name - name;
    cust->address = address;
    cust->accounts.insert_element(account);
    ... Code to initialize other fields

    Trans.commit();
}
```



# Extents in ODMG C++

Class extents maintained automatically in the database.

To access a class extent:

```
d_Extent<Customer> customerExtent(bank_db);
```

Class d_Extent provides method

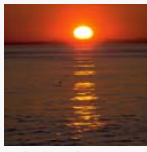
```
d_Iterator<T> create_iterator()
```

to create an iterator on the class extent

Also provides select(pred) method to return iterator on objects that satisfy selection predicate pred.

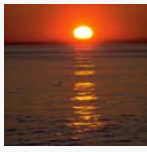
Iterators help step through objects in a collection or class extent.

Collections (sets, lists etc.) also provide create_iterator() method.



# Iterators in ODMG C++

```
int print_customers() {
    Database bank_db_obj;
    Database * bank_db = &bank_db_obj;
    bank_db->open (“Bank-DB”);
    d_Transaction Trans; Trans.begin ();

    d_Extent<Customer> all_customers(bank_db);
    d_Iterator<d_Ref<Customer>> iter;
    iter = all_customers–>create_iterator();
    d_Ref <Customer> p;
    while{iter.next (p))
          print_cust (p);  // Function assumed to be defined
    elsewhere
    Trans.commit();
}
```



# OQL

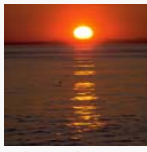
Declarative query language OQL, looks like SQL

- Form query as a string, and execute it to get a set of results (actually a bag, since duplicates may be present)

```
d_Set<d_Ref<Account>> result;
d_OQL_Query q1("select a
                from Customer c, c.accounts a
                where c.name='Jones'
                      and a.find_balance() > 100");
d_oql_execute(q1, result);
```

Provides error handling mechanism based on C++ exceptions, through class d_Error

Provides API for accessing the schema of a database.



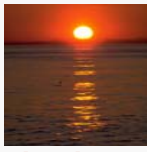
# ODMG C++ Vs. ObjectStore

Drawback of the ODMG C++ approach:

- Two types of pointers
- Programmer has to ensure mark_modified() is called, else database can become corrupted

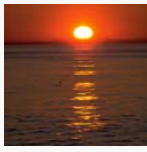
ObjectStore approach

- Uses *exactly* the same pointer type for in-memory and database objects
- Persistence is transparent applications
  - Except when creating objects
- Same functions can be used on in-memory and persistent objects since pointer types are the same



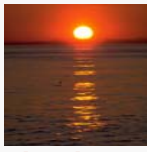
# Persistent Java Systems

- Persistence in Java, first led by ODMG, then transferred to **Java Database Objects (JDO).**
- An object is made persistent by using the **makePersistent()** method of the **PersistenceManager** class.
- **Persistent by reachability**
  - Any other object reachable from a persistent object becomes persistent.
- **Byte code enhancement**
  - Instead of declaring a class to be persistent, classes whose objects may be persistent are specified in a **configuration file**.
  - An **enhancer** program is executed which reads the configuration file and carries out two tasks
    - Creates structures in a database to store objects of the class
    - Modifies the bytecode to handle tasks related to persistence



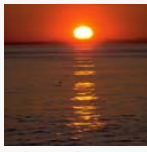
# Persistent Java Systems

- **Database mapping**
  - JDO does not define how data are stored in the back-end database.
  - The objects can be stored in a relational database and the enhancer program may create an appropriate schema in the database. Then, how objects are stored is implementation dependent and not defined by JDO.
- **Class extents**
  - Created and maintained automatically for each class declared to be persistent
  - The Iterator interface provided by Java can be used to create iterators on class extents.
- **Single reference types**
  - There is no difference in type between a reference to a transient object and a reference to a persistent object.



# JDOQL

- JDO provides a query language called **JDO Query Language (JDOQL)** to access instance based on specific search criteria.
- The **JDOQL** can be used in any type of database like Relational Database, Object Database, Hierarchical database etc.
- The JDO queries is performed by '**Query**' interface which is created by using **'PersistenceManager**' interface.
- The JDO Query allows us to **filter out** instances from set of instances specified by an **Extent** or a **Collection**.
- A **Filter** consists of a Boolean expression and it is applied to the instances. The query result includes all the instances for which the boolean expressions are true.

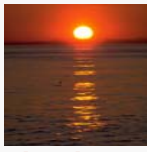


# JDOQL

- Example:

```
Extent extent = manager.getExtent (player.class,true);
String filter = "name == a ";
Query query = manager.newQuery(extent,filter);
```

- In filter we can use a number of operators like **equality** operator(==), **inequality** operator(!=), **comparison** operators like greater than, greater than or equal to (<, >, <=, >=), **Boolean** operators like conditional AND, logical AND (&, &&, |, ||, !) and **arithmetic** operators like addition, subtration, multiplication, division (+, -, *, %, ~).
- The JDOQL also supports string expressions inside the filter. For this purpose **startsWith(..) and endsWith()** methods are provided. For example, we can write the filter as **name.startsWith("a%")**.

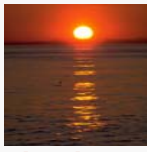


# JDO in Action

- First we should create a **property file** as shown:

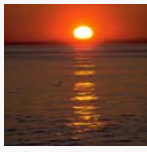
c:\demojdo\jdopack\jdo.properties

```
javax.jdo.PersistenceManagerFactoryClass=
com.sun.jdori.fostore.FOStorePMF
javax.jdo.option.ConnectionURL=fostore:dbdemo
javax.jdo.option.ConnectionUserName=biba
javax.jdo.option.ConnectionPassword=
javax.jdo.option.Optimistic=false
```



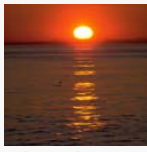
# FOStore

- The JDO reference implementation has its own storage facility called '**File Object Store'(FOStore)**.
- We can create a new datastore by using **FOStore** and test our application.



# Creating the database

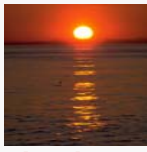
```
import java.io.*;
import java.util.*;
import javax.jdo.*;
public class jdocreatedb
{
   public static void main(String args[])
   {
      try{
  InputStream   fis = new FileInputStream("jdo.properties");
        Properties    props= new Properties();
         props.load(fis);
         props.put("com.sun.jdori.option.ConnectionCreate","true");
        PersistenceManagerFactory    factory =
                    JDOHelper.getPersistenceManagerFactory(props);
         PersistenceManager  manager =  factory.getPersistenceManager();
         Transaction       tx =    manager.currentTransaction();
         tx.begin();
         tx.commit();
      }
      catch(Exception e1) { System.out.println(""+e1); }
   }}
```



# JDO in Action

```
package jdopack;

public class player
{
String  name;
String  game;

public   player()  {  }
   public   player(String a,String b)
   {
      name = a;
      game = b;
   }
//--------------------------------
```

```
 public String getName()
  {
     return name;
  }
  public void setName(String b)
  {
     name = b;
  }
//--------------------------------
  public String getGame()
  {
     return  game;
  }
  public void setGame(String c)
  {
     game = c;
  }
}
```

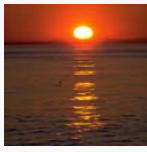


# JDO

JDO uses this metadata file to identify the classes that need to be persisted and specify the persistence-related information that cannot be expressed in Java.

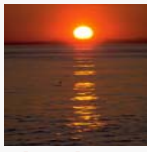
C:\demojdo\jdopack\package.jdo

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE jdo PUBLIC
"-//Sun Microsystems, Inc.//DTD
Java Data Objects Metadata 1.0//EN"
"http://java.sun.com/dtd/jdo_1_0.dtd" >

<jdo>
<package name="jdopack">
<class name="player" />
</package>
</jdo>
```



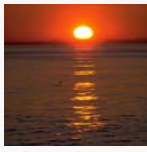
# makePersistent()

```
PersistenceManagerFactory  factory=null;
PersistenceManager         manager=null;
Transaction                tx;
InputStream     fis = new FileInputStream("jdo.properties");
Properties    props = new Properties();
    props.load(fis);
    factory = JDOHelper.getPersistenceManagerFactory(props);
    manager =  factory. getPersistenceManager();
    tx = manager.currentTransaction();
    tx.begin();
    String a = “Name”:
    String b = “Surname”
    player    player1  =  new player(a,b);
    manager.makePersistent(player1);
    tx.commit();
```



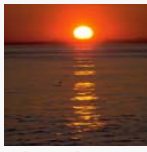
# JDO Extent

```
PersistenceManagerFactory  factory=null;
PersistenceManager         manager=null;
InputStream     fis = new FileInputStream("jdo.properties");
Properties    props = new Properties();
        props.load(fis);
        factory = JDOHelper.getPersistenceManagerFactory(props);
        manager =  factory. getPersistenceManager();

Extent extent = manager.getExtent(player.class,true);
Iterator iter = extent.iterator();
while(iter.hasNext())
{
    Object ob = (Object)iter.next();
    System.out.println(ob.getName());
    System.out.println(ob.getGame());
}
```



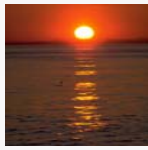
# JDO Query

```
Extent extent = manager.getExtent(player.class,true);
String filter = "name == a ";
Query query = manager.newQuery(extent,filter);
query.declareParameters("String a");

Collection list1 = (Collection)query.execute(a);
Iterator i = list1.iterator();
while(i.hasNext())
{
    Object ob = (player)i.next();
    System.out.println(ob.getName());
    System.out.println(ob.getGame());
}
query.close(list1);
```



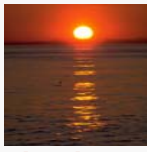
# O-R mapping

- Object-relational mapping (ORM, O/RM, and O/R mapping) is a programming technique for converting data between incompatible type systems in relational databases and object-oriented programming languages.
- This creates, in effect, a "virtual object database" that can be used from within the programming language.
- There are both free and commercial packages available that perform object-relational mapping, although some programmers opt to create their own ORM tools.



# O-R impedance mismatch

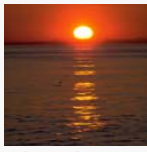
- The **object-relational impedance mismatch** is a set of conceptual and technical difficulties that are often encountered when a relational database management system (RDBMS) is being used by a program written in an object-oriented programming language or style;
  - Particularly when objects or class definitions are mapped in a straightforward way to database tables or relational schemata.
- This practice has been recommended and documented by some object-oriented literature as a way to use databases in object-oriented programs.



# JPA: Java Persistence API

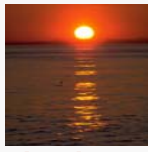
- The Java Persistence API, sometimes referred to as JPA, is a Java programming language framework that allows developers to manage relational data in applications using Java Platform, Standard Edition and Java Platform, Enterprise Edition.

- Persistence in this context covers three areas:
  - the API itself, defined in the javax.persistence package
  - the Java Persistence Query Language (JPQL)
  - object/relational metadata



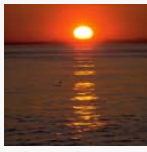
# Relation to JDO

- The Java Persistence API was developed in part to unify the Java Data Objects API, and the EJB 2.0 Container Managed Persistence (CMP) API.
- As of 2009 most products supporting each of those APIs support the Java Persistence API.
- The Java Persistence API specifies relational persistence (ORM: Object relational mapping) only for RDBMS (although providers exists who support other datastores).
- The Java Data Objects specification(s) provides relational persistence (ORM), as well as persistence to other types of datastores.



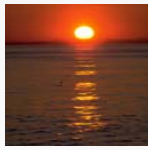
# Relational Persistence for Java and .NET: Hibernate

- Hibernate is an object-relational mapping (ORM) library for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database.
- Hibernate solves object-relational impedance mismatch problems by replacing direct persistence-related database accesses with high-level object handling functions.
- Hibernate lets you develop persistent classes following object-oriented idiom - including association, inheritance, polymorphism, composition, and collections.
- The LGPL open source license allows the use of Hibernate and NHibernate in open source and commercial projects.



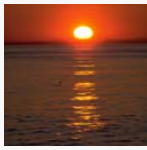
# Hibernate

- Hibernate's primary feature is mapping from Java classes to database tables (and from Java data types to SQL data types).
- Provides data query and retrieval facilities.
- Generates the SQL calls and relieves the developer from manual result set handling and object conversion, keeping the application portable to all supported SQL databases, with database portability delivered at very little performance overhead.
- Hibernate allows you to express queries in its own portable SQL extension (HQL), as well as in native SQL, or with an object-oriented criteria.
- https://www.hibernate.org/



# Comparison of O-O and O-R Databases

- **Relational systems**
  - simple data types, powerful query languages, high protection, high-level optimization (such as reducing I/O) is easy, low performance.
- **Persistent-programming-language-based OODBs**
  - complex data types, integration with programming language, **high performance**, more susceptible to data corruption by programming errors.
- **Object-relational systems**
  - complex data types, powerful query languages, high protection.
- Note: Many real systems blur these boundaries
  - E.g. persistent programming language built as a wrapper on a relational database offers first two benefits, but may have poor performance.

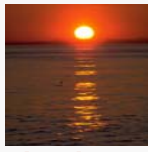


# O-R database management systems

- The following Database Management Systems (DBMSs) have at least some object-relational features. They vary widely in their completeness and the approaches taken.

| | Type | Method | Type inheritance | Table inheritance |
|---|---|---|---|---|
| CUBRID | Yes | Yes | Yes | Yes |
| Oracle | Yes | Yes[1] | Yes | Yes |
| OpenLink Virtuoso | Yes | Yes | Yes | Yes |
| PostgreSQL | Yes | Yes | Yes | Yes |

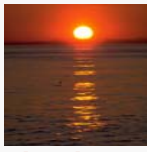
| | Array | List | Set | Multiset | Object reference |
|---|---|---|---|---|---|
| CUBRID | Yes | Yes | Yes | Yes | Yes |
| Oracle | Yes | Yes | Yes | Yes | Yes |
| OpenLink Virtuoso | Yes | Yes | Yes | Yes | Yes |
| PostgreSQL | Yes | Yes | Yes | Yes | Yes |



# O-R in Oracle

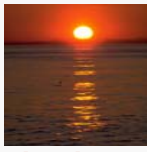
**create type** ADDRESS_TY **as object**
(Street VARCHAR2(50),
City VARCHAR2(25),
State CHAR(2),
Zip NUMBER);

**create type** PERSON_TY **as object**
(Name VARCHAR2(25),
Address ADDRESS_TY);



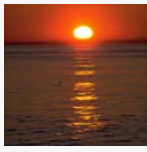
# Derivation

```
create or replace type base_type as object (
  a number,
  constructor function base_type return self as result,
  member function  func return number,
  member procedure proc (n number)
) instantiable not final;

create or replace type deriv_type under base_type (
  m number,
  overriding member function func return number
);
```



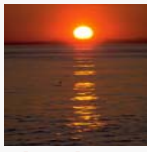
# Implementation of Base Type

```
create or replace type body base_type as
  constructor function base_type return self as result is
  begin
    a:=0;
    return;
  end base_type;

  member function func return number is
  begin
    return a;
  end func;

  member procedure proc (n number) as
  begin
    a:=n;
  end proc;
end;
```



# Implementation of Derived Type

```
create or replace type body deriv_type as
  overriding member function func return number is
  begin
    return m*a;
  end;
end;
```

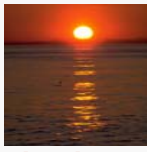


# Creating Tables with Defined Types

- create table table_base (
  b base_type
);

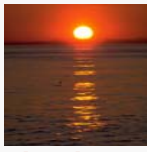
- declare
  base  base_type := base_type();
  deriv deriv_type:= deriv_type(8,9);
begin
  insert into table_base values(base);
  insert into table_base values(deriv);
end;

- select t.b.func() from table_base t;



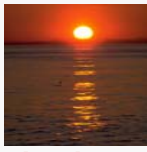
# Loop

```
begin
  for r in (select b from table_base) loop
    dbms_output.put_line(r.b.func());
  end loop;
end;
```



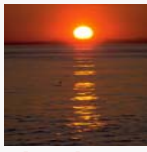
# Clean up

```
drop table table_base;

drop type deriv_type;
drop type base_type;
```



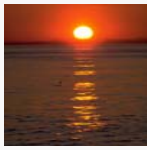
# not instantiable

```
create or replace type math_type as object (
  num  number(4),
  not instantiable member function  func(n number) return number,
) not instantiable not final;
```



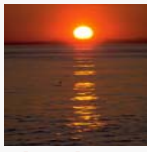
# Derive from math_type

```
create or replace type add_type under math_type (
  overriding member function func(n number) return number,
) instantiable final;

create or replace type body add_type as
  overriding member function func(n number) return number is begin
    return n + num;
  end;
end;
```



# Derive another type from math_type

```
create or replace type mult_type under math_type (
  overriding member function func(n number) return number,
);

create or replace type body mult_type as
  overriding member function func(n number) return number is begin
    return n * num;
  end;
```



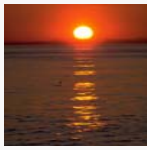
# Creating tables

- create table t_(m math_type, o number(3));

- declare

```
  a add_type  := add_type (49);
  m mult_type := mult_type(15);
begin
  insert into t_ values(a, 100);
  insert into t_ values(m,  10);
end;
```

- select t.m.func(6) from t_ t;



# End of Chapter

**Database System Concepts**