

8. Object Persistence

Contents

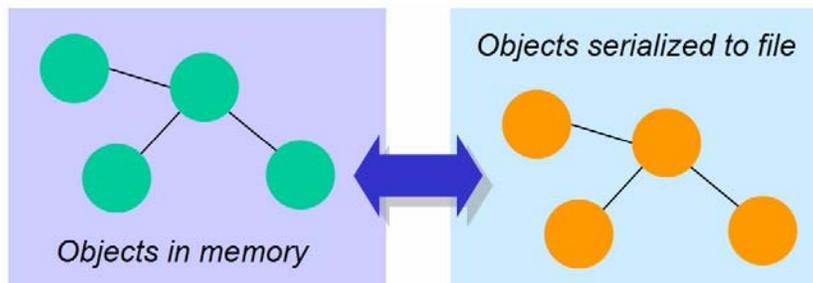
INTRODUCTION.....	2
OBJECT SERIALIZATION.....	2
PERSISTENCE WITH DATABASES.....	3
DATA MODELS.....	4
FIRST GENERATION.....	4
SECOND GENERATION.....	5
THIRD GENERATION.....	5
OBJECT-RELATIONAL MAPPING	6
INHERITANCE	6
MANY-MANY RELATIONSHIPS.....	8
OBJECT DATA MODEL	10
OBJECT IDENTIFIERS	10
OBJECTS AND LITERALS.....	11
REPRESENTATION OF RELATIONSHIPS.....	11
TYPES OF RELATIONSHIP	12
ORTHOGONAL PERSISTENCE.....	14
OBJECT-ORIENTED DATABASE STANDARDS.....	16
THE OBJECT-ORIENTED DATABASE SYSTEM MANIFESTO.....	16
THE ODMG PROPOSED STANDARD.....	17
COMPARING OODBMS AND RDBMS	18
ADVANTAGES OF OODBMS.....	18
DISADVANTAGES OF OODBMS.....	18
EXAMPLES OF REAL-WORLD OODBMS USERS	19
OODBMS PRODUCTS	19
THE OBJECT RELATIONAL MODEL AND ORDBMS	20
OBJECT RELATIONAL MAPPING FRAMEWORKS.....	21
WRITING A PERSISTENCE LAYER	23
EXAMPLE	23
CHANGING THE DATABASE	27
FURTHER READING	29

Introduction

One of the most critical tasks that applications have to perform is to save and restore data. For example, a word processing application saves documents to disk, a game can save its state so that the player can take a break, and an ecommerce web site saves information about what its customers have ordered.

Persistence is the storage of data from working memory so that it can be restored when the application is run again. In object-oriented systems, there are several ways in which objects can be made persistent. The choice of persistence method is an important part of the design of an application.

Object Serialization



Object **serialization** is a simple persistence method which provides a program the ability to read or write a whole object to and from a stream of bytes. It allows Java objects to be encoded into a byte stream suitable for streaming to a file on disk or over a network. serializing an object requires only that it meets one of two criteria. The class must implement the *Serializable* interface (*java.io.Serializable*), which does not declare any methods, and have accessors and mutators for its attributes.

For example, to serialize an object *t1* of serializable class *Team* to a file:

```
// create output stream
File file = new File("teams_serialize.ser");
String fullPath = file.getAbsolutePath();
fos = new FileOutputStream(fullPath);

// open output stream and store team t1
out = new ObjectOutputStream(fos);
out.writeObject(t1);
```

You can download the full code for this example from your course web site.

Persistence with Databases

Most business applications use a Relational Database Management System (RDBMS) as their data store while using an object-oriented programming language for development. This means that objects must be mapped to tables in the database and vice versa, instead of the data being stored in a way that is consistent with the programming model. Applications generally require the use of SQL statements embedded in another programming language. The difference in data definitions, leading to frequent mappings from tables to objects and back, is known as the *"impedance mismatch"*.

Further, object-oriented application design attempts to use classes and objects which describe real-world entities. The goal of object-oriented design is to model a process, while the goal of relational database design is normalisation. The mapping from objects to tables can be difficult if the model contains complex class structures, large unstructured objects, or object inheritance. The resulting tables may store data inefficiently, or access to data may be inefficient.

The increasing importance of object-oriented design and programming has spurred on research and development in new approaches to the management of object storage in databases. There are essentially three approaches which have been developed:

- the Object-Oriented Database Management System (OODBMS)
- the Object-Relational Database Management System (ORDBMS)
- Object Relational Mapping

Relational databases have achieved commercial dominance, and are supported by major companies such as Oracle, IBM and Microsoft. At present it is not clear which, if any, of the above approaches will become widely accepted ways to replace or extend the relational database model for object-oriented applications.

Data Models

Before the development of the first DBMS, application programs accessed data from flat files. These did not allow representation of logical data relationships or enforcement of data integrity. Several data models have been developed since the 1960s to provide these features.

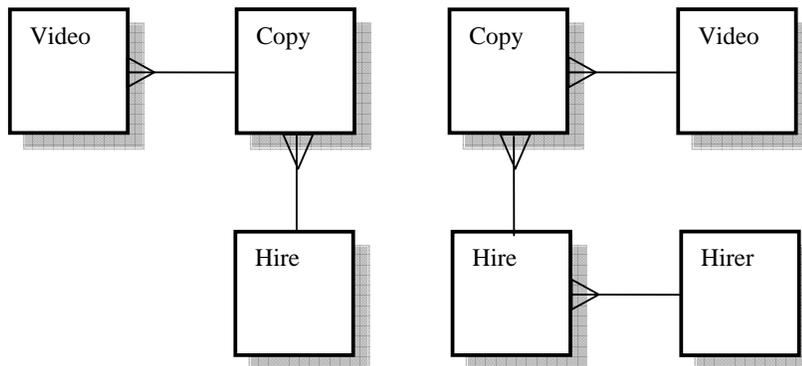
First Generation

Hierarchical Data Model

- implemented primarily by IBM's Information Management System (IMS)
- allows one-to-one or one-to-many relationships between entities
- an entity at a "many" end of a relationship can be related to only one entity at the "one" end
- this model is *navigational* – data access is through defined relationships, efficient if searches follow predefined relationships but performs poorly otherwise, e.g for ad-hoc queries.

Network Data Model

- standard developed by the Committee on Data Systems Languages (CODASYL)
- allows one-to-one or one-to-many relationships between entities
- allows multiple parentage – a single entity can be at the "many" ends of multiple relationships
- navigational



Hierarchical - each entity is at the "many" end of at most one relationship

Network - "Copy" entity above is at the "many" end of two relationships

Second Generation

Relational Data Model

- developed by Edgar Codd (1970)
- data represented by simple tabular structures (relations)
- relationships defined by primary keys and foreign keys
- data accessed using high-level non-procedural language (SQL)
- separates logical and physical representation of data
- highly commercially successful (Oracle, DB2, SQL Server, etc)

SQL is not computationally complete. This means that it does not have the full power of a programming language. For example, while any SQL code could be rewritten as a Java program, not all Java programs could be rewritten in SQL. Applications generally require the use of SQL statements embedded in another programming language.

Third Generation

As most programming languages are now object-oriented, there is a growing impetus to store data in the form of objects. Two competing approaches have emerged. Both approaches are described in these notes.

Object Data Model

- offers **persistence** to objects, including their associations, attributes and operations, i.e. they continue to exist after the program run finishes
- requirements specified by Atkinson in 1989, standards proposed by Object Data Management Group (ODMG)
- navigational – a step backwards from the relational model in some ways, making OODBMs relatively unsuited to ad-hoc querying

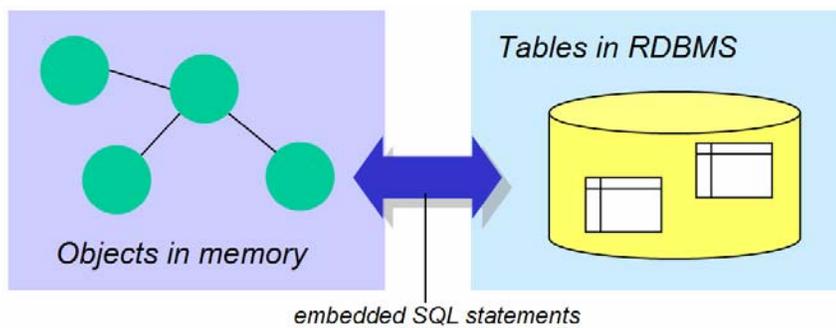
Object-Relational Model

- hybrid (or “post-relational”) model – objects can be stored within relational database tables
- proposed by Stonebraker (1990)
- no accepted standards, but ORDBMS features supported by major commercial RDBMSs such as Oracle

Object-Relational Mapping

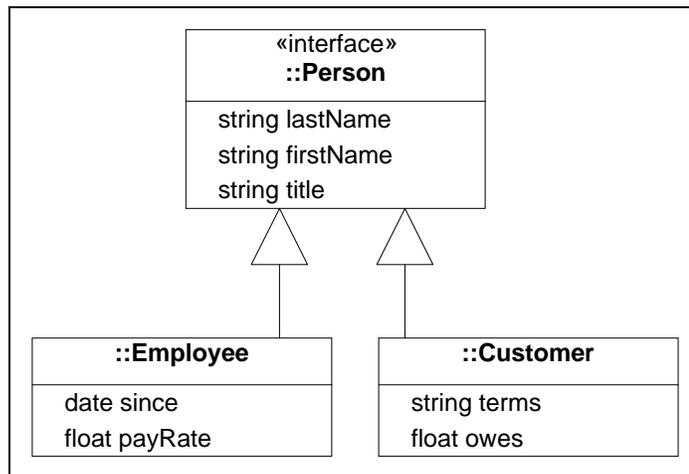
Most business database applications use relational databases. The job of creating object-oriented applications which deal with the data involves mapping the objects in the application to tables in the database, and using the JDBC classes and embedded SQL statements to move information between objects in memory and database tables.

This can sometimes be a straightforward matter of mapping individual classes to separate database tables. However, if the class structure (created using object-oriented design techniques to represent the real world of the application) is more complex, then the mapping must be carefully considered to allow data to be represented and accessed as efficiently as possible.



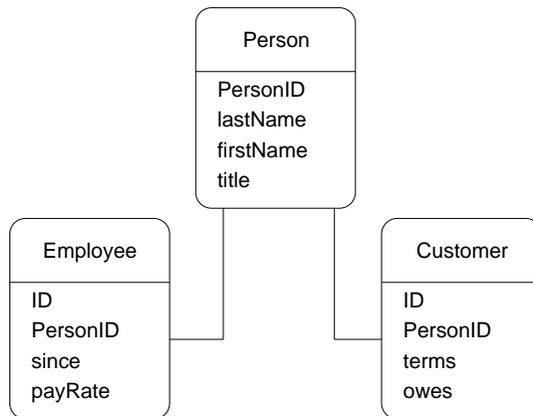
Inheritance

The UML class diagram shows a simple inheritance tree with an abstract class *Person* and two subclasses.



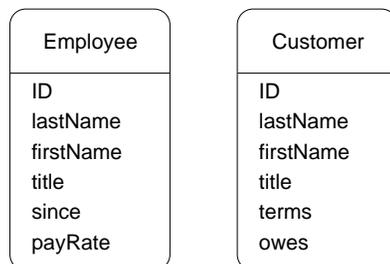
This can be mapped to a relational database in three ways, illustrated by the following examples:

Vertical Mapping



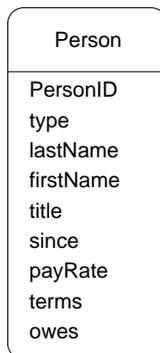
- separate table for each class, including abstract class
- subclass tables related by foreign keys to superclass tables
- need additional key fields (PersonID, ID) to create relationships
- creating an *Employee* object in the application involves joining *Person* and *Employee* tables in the database
- can result in complex queries for deeper levels inheritance than this example

Horizontal Mapping



- each concrete class mapped to a different table
- each table contains columns for all attributes of class, including inherited ones
- simple to design, but changes in abstract class design require changes in *all* tables

Filtered Mapping



- all concrete classes mapped to one table
- contains attributes of all concrete and abstract classes
- a filter column (type) is included to distinguish between subclasses
- violates normalisation rules
- wasteful – e.g the payRate column is not relevant to a *Customer*

Guidelines for the choice of which way to map an inheritance tree:

Use **Vertical** mapping when:

- there is significant overlap between types
- changing types is common

Use **Horizontal** mapping when:

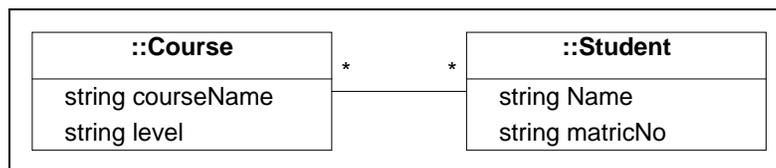
- there is little overlap between types
- changing types is uncommon

Use **Filtered** mapping for:

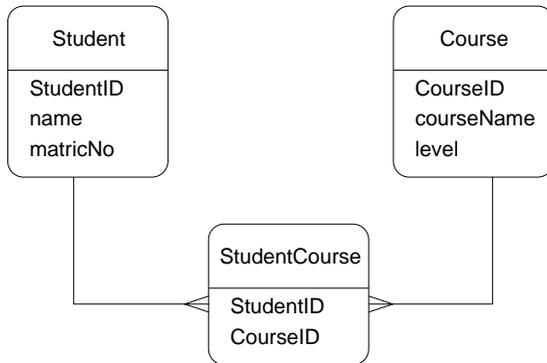
- simple or shallow hierarchies with little overlap between types

Many-Many Relationships

The class diagram below shows a many-many relationship between *Course* and *Student* classes. A *Student* can be on many *Courses*, while the one *Course* has many *Students* on it.

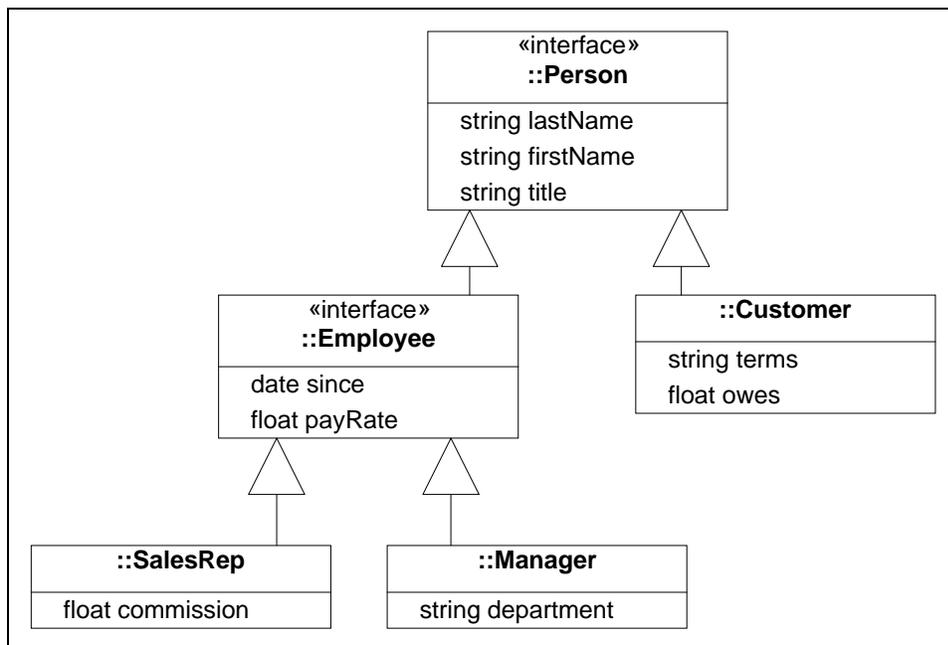


In a relational database a join table is required to represent this relationship:



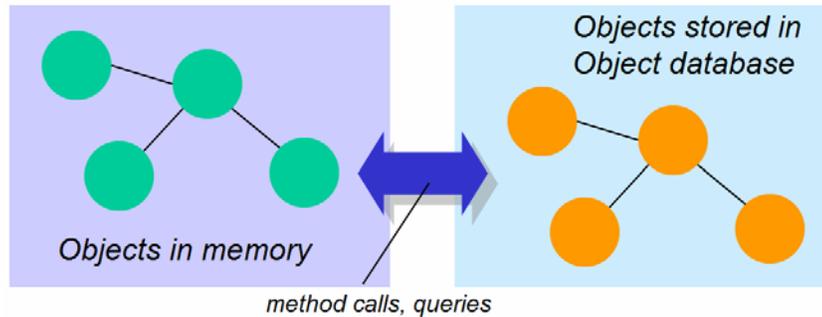
EXERCISE 1.

This class diagram is an extension of the inheritance tree described above. Create relational tables for object-relational mappings using vertical, horizontal and filtered mappings. Discuss how the increased depth of inheritance affects these mappings compared to those in the example.



Object Data Model

Object-oriented database systems are based on the concept of **persistent objects**. They use class declarations similar to those used by object-oriented programming languages. The class declarations should additionally indicate relationships between objects.



The system must be able to represent traditional database relationships (one-to-many, many-to-many) and also relationships unique to the object-oriented model, such as inheritance.

Object Identifiers

In RDBMS, entities are uniquely identified by primary keys, and relationships are represented by matching primary key-foreign key data. The relationships are used in queries as required to join tables. Identification *depends on the values in the key fields*.

In contrast, an object-oriented database stores object identifiers (OIDs) within an object to indicate other objects to which it is related. The object identifier is not visible to the user or database programmer. An object *remains the same object even when its state takes on completely new values*.

The fact that an object's identity is distinct from its values means that the concepts of equivalence and equality are different:

Equivalent: same OID
Equal: same state values

Equality can exist at different levels:

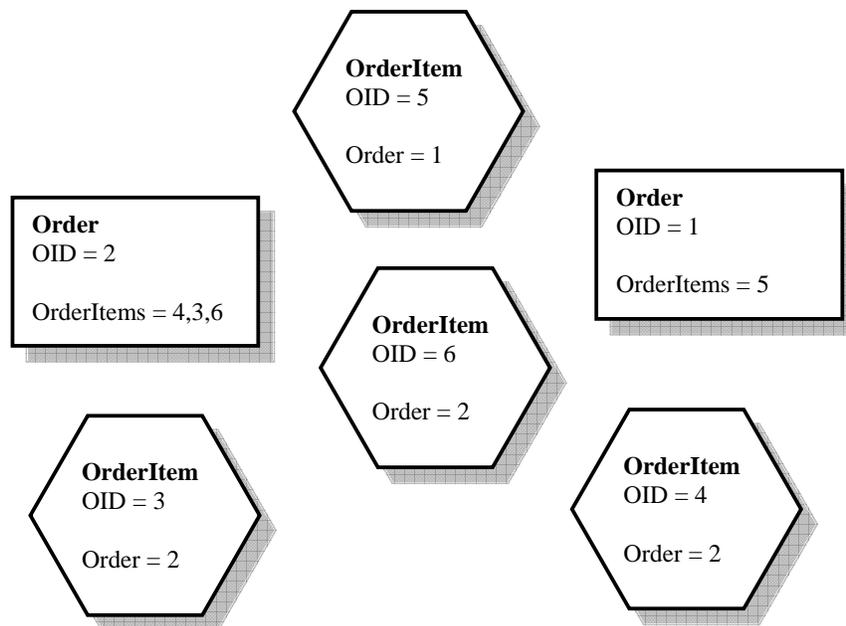
Shallow equality same state values (e.g. two *CustomerOrder* objects have same values)
Deep equality same state values and related objects also contain same state values (e.g. two *CustomerOrder* objects have same values and all their related *OrderItem* objects have same values)

Objects and Literals

Objects can change their state values, and are described as being *mutable*. Another component of the object data model is the **literal**, which represents a value or set of values which cannot be changed. A literal is described as being *immutable*. Literals do not have OIDs.

Representation of Relationships

The figure shows the use of OIDs to represent relationships:



- the rectangles represent objects which are instances of a *Order* class
- the hexagons represent objects which are instances of an *OrderItem* class
- each *Order* is related to one or more *OrderItems*
- each OID is unique
- the OID is not related to the class

In the diagram all relationships have **inverse relationships**. For example, *Order 2* has a reference to *OrderItem 4*, while *OrderItem 4* has a reference to *Order 2*. Inverse relationships are optional but can be used to enforce **relationship integrity**. Without an inverse relationship, there is nothing to stop an *OrderItem* being referenced by more than one *Order*, but the inverse ensures that an *OrderItem* is associated with only one *Order*.

Note that only relationships predefined by storing OIDs can be used to query or traverse the database. For example, if an *OrderItem* did not store the OID for an *Order*, it would not be possible to find the related *Order* in the database.

An object database is therefore **navigational**. This implies that the object data model is a step backward compared to the relational model as it limits the flexibility of the user or programmer. In a relational database, data in related tables can *always* be found using suitable SQL joins.

Object-oriented databases are generally not as well suited to ad-hoc querying as relational databases. However, *performance can be better than a relational database for predictable access patterns which follow predefined relationships*.

Types of Relationship

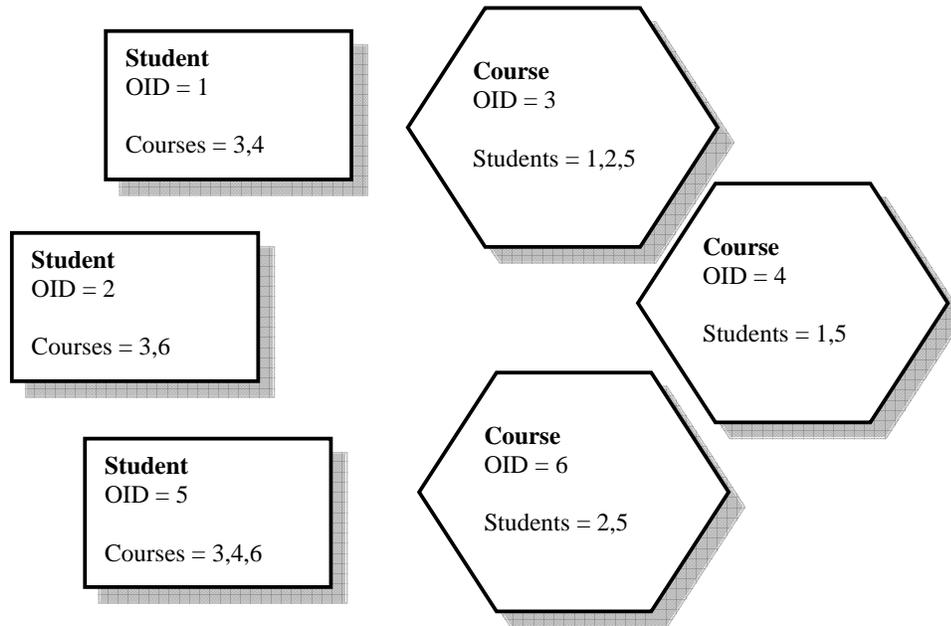
One-to-Many

- unlike relational model, the object data model allows multi-valued attributes (known as *sets* and *bags*)
- class at “many” end has attribute to hold OID of parent (see *OrderItem* in the figure above)
- class at “one” end has attribute to hold a *set* of related OIDs (see *CustomerOrder* in the figure)

Many-to-many

- object data model allows direct many-to-many relationships
- in contrast, relational model requires the use of composite entities to remove many-to-many relationships
- each class has an attribute to hold a set of OIDs.

For example, in the diagram below there is a many-to-many relationship between *Student* and *Course*— each *Student* studies many *Courses*, while each *Course* has many *Students*. (In the relational model, a join table *StudentCourse* would be required).



“Is A” and “Extends”

These relationships can be represented because the object-oriented paradigm supports inheritance. Inheritance can be difficult for a relational database to deal with.

For example, a company needs to store information about *Employees*. There are specific types of employee, such as *SalesRep*. A *SalesRep* has an attribute to store a sales area, e.g. North, West. Other types of employee also have attributes specific to them.

This situation can be modelled easily in an object-oriented system. For example in Java:

```
public class Employee {
    String name;
    String address;
    ...
}

public class SalesRep extends Employee {
    String area;
}
```

with similar definitions for other employee types.

Inheritance is more difficult to represent in the relational model (see Object Relational Mapping).

“Whole-Part”

A whole-part relationship is a many-to-many relationship with a special meaning. It is useful in a manufacturing database which is used to track parts and subassemblies used to create products. A product can be made up of many part and subassemblies. Also, the same part or subassembly can be used in many products. This type relationship is represented as a many-many relationship using sets of OIDs in two classes. This type of relationship is very awkward for a relational database to represent.

Orthogonal persistence

Orthogonality can be described by saying that feature A is orthogonal to feature B if you don't have to care about feature A while thinking about feature B. Orthogonal persistence is a form of object persistence which adheres to the following principles (Atkinson & Morrison, 1995):

- **principle of persistence independence**
programs look the same whether they manipulate long-term or short-term data
- **principle of data type orthogonality**
all data objects are allowed to be persistent irrespective of their type
- **principle of persistence identification**
the mechanism for identifying persistent objects is not related to the type system

An orthogonally persistent programming language allows a programmer to work with objects and to store any object of any type as required.

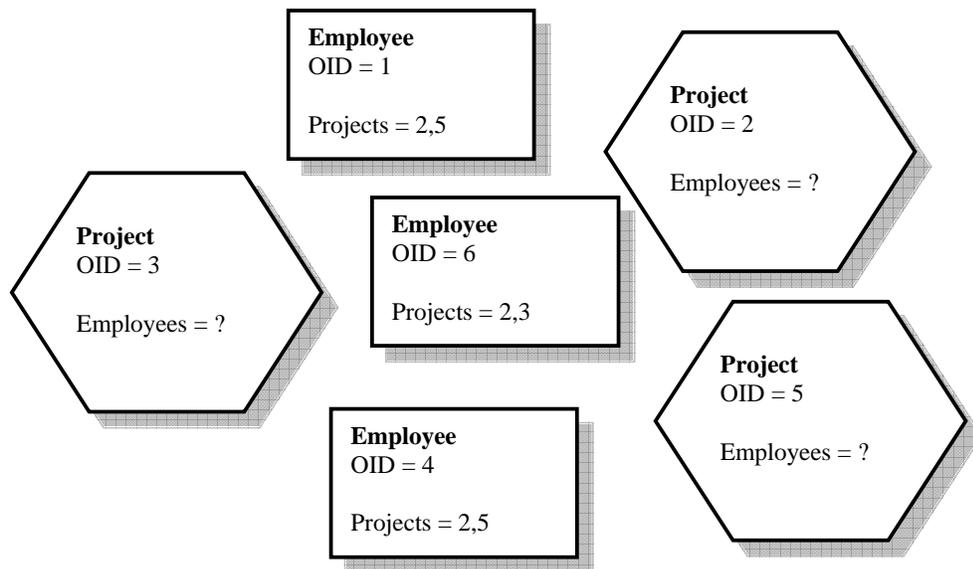
EXERCISE 2.

1. A company has Sales, Engineering and Admin departments. Its employees include John Jones (Sales), Lucy Smith (Engineering), Ken Brown (Admin), Bill Clark (Sales), Jan McDonald (Admin) and Jill Johnson (Admin). The departments and employees are to be represented by objects in an object database.

Assign an OID to each object, and list the OID references which each object must have to ensure relationship integrity.

2. A company assigns employees to projects. Each employee can be assigned to one or more projects. Each project can have one or more employees assigned to it. The diagram shows how this is represented in an object database.

Copy the diagram and complete the object references to ensure relationship integrity.



Object-Oriented Database Standards

The Object-Oriented Database System Manifesto

The basic specification for a relational database system was defined in 1970 by Codd. Atkinson attempted to define a similar specification for an object-oriented database system in a manifesto in 1989. This paper provided a focus for research and development on OODBMSs.

The manifesto considered features of database systems and object-oriented systems to define a set of mandatory and optional features of an object-oriented database.

Mandatory features:

- **Complex objects** (OO feature)
objects can contain attributes which are themselves objects.
- **Object identity** (OO)
- **Encapsulation** (OO)
- **Classes** (OO)
- **Inheritance** (OO): class hierarchies
- **Overriding, Overloading, Late Binding** (OO)
- **Computational completeness** (OO)
- **Persistence** (DB)
data must remain after the process that created it has terminated
- **Secondary Storage Management** (DB)
- **Concurrency** (DB)
- **Recovery** (DB)
- **Ad hoc query facility** (DB)
not necessarily a query language – could be a graphical query tool

Optional features:

Multiple inheritance, Type checking, Inferencing, Distribution, Design Transactions and Versions.

The ODMG Proposed Standard

One of the crucial factors in the commercial success of RDBMSs is the relative standardisation of the data model, the data definition language (DDL) and SQL. DDL commands for the major RDBMSs are very similar, while SQL standards are set by recognised standards bodies (ANSI and ISO).

The Object Data Management Group (ODMG) was formed by a group of industry representatives to define a proposed standard for the object data model. The ODMG standard specifies how objects are stored in database systems (persistence). The Java part of the standard specifies how database systems communicate with the Java programming language.

The first version of the ODMG standard was published in 1993 and updated versions have been published since (ODMG 2.0 in 1997, ODMG 3.0 in 2000). Many OODBMS vendors have committed to adhering to the standard, but it has not been universally adopted. It is still far from being as widely recognised as the relational database standards.

The ODMG proposed standard defines the following:

- basic terminology
- data types
- classes and inheritance
- objects
- collection objects (including sets, bags, lists, arrays)
- structured objects (Date, Interval, Time, Timestamp – similar to SQL)
- relationships
- object definition language (ODL)
- object query language (OQL)

Although the standard originally focused on object-oriented database systems, **relational databases can also be the basis of an ODMG 3.0 implementation** (see page 23). The standard just specifies the interface with the programming language and does not say anything about the details of data storage.

Note also that some object oriented database products do not adhere to the ODMG standard.

Comparing OODBMS and RDBMS

Advantages of OODBMS

- **Complex objects and relationships**
objects can include other objects
- **Class hierarchy**
real-world situations are often best represented by hierarchy and inheritance, which is easier to represent in the object model
- **No impedance mismatch**
no mapping between data models
- **No need for primary keys**
objects are uniquely identified by OIDs
- **One data model**
Object data model can deal with complete system including dynamic aspects
- **One programming language**
no embedded query code. SQL statements embedded as strings are a common source of error, and cannot be checked at compile time.
- **No need for query language**
interaction with database by directly accessing objects (although query language is possible)
- **High performance for certain tasks**
if data access follows predefined relationships

Disadvantages of OODBMS

- **Schema changes**
in RDBMS tables are independent of application and can often be changed without recompiling the application
- **Lack of agreed standards**
ODMG is the closest, but nowhere near as universal as relational standard
- **Lack of ad-hoc querying**
Object model does not lend itself well to ad-hoc queries, although query languages do exist (OQL). Important as in many users want to access data in forms not conceived of at design time.

In general, RDBMSs are probably more suitable for databases with a variety of query and user interface requirements (i.e. most mainstream business applications), while OODBMSs are

appropriate for applications with complex, irregular data, where data access will follow predictable patterns (e.g CAD/CAM systems, manufacturing databases).

Examples of real-world OODBMS users

- The Chicago Stock Exchange - managing stock trades
- CERN in Switzerland - large scientific data sets
- Radio Computing Services – automating radio stations (library, newsroom, etc)
- Adidas – content for web site and CD-ROM catalogue
- Federal Aviation Authority – passenger and baggage traffic simulation
- Electricite de France – managing overhead power lines

OODBMS products

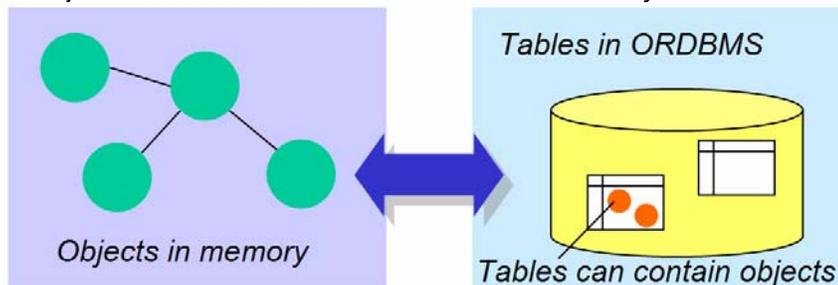
- Versant
- ObjectStore and PSE Pro from eXcelon
- Objectivity/DB
- Intersystems Cache
- POET fastObjects
- db4o
- Computer Associates Jasmine
- GemStone

The Object Relational Model and ORDBMS

A limitation of the relational model is that it requires that each column in a table must contain atomic data, i.e. single values. The object relational model is an extension of the relational model, with the following features:

- a field may contain an object with attributes and operations.
- complex objects can be stored in relational tables
- the object relational model offers some of the advantages of both the relational and object data models
- has the commercial advantage of being supported by some of the major RDBMS vendors

An object relational DBMS is sometimes referred to as a *hybrid* DBMS.



Some commercial RDBMSs support ORDBMS features, for example Oracle. In Oracle, it is possible to declare a new data type which is basically a class, and to set this new type as the data type for a table column.

Example:

Type declaration – specifies attributes and operations

```
CREATE TYPE Name AS OBJECT (
    first_name CHAR (15),
    last_name CHAR (15),
    middle_initial CHAR (1);
    MEMBER PROCEDURE initialize;
```

Code to define operations – in this case simply a class constructor

```
CREATE TYPE BODY Name AS
    MEMBER PROCEDURE initialize IS
    BEGIN
        first_name := NULL;
        last_name := NULL;
        middle_initial := NULL;
    END initialize;
END;
```

Using the new type in a table

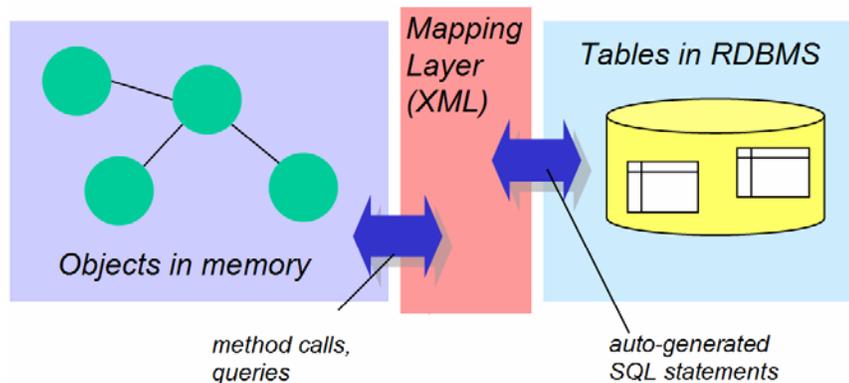
```
CREATE TABLE person(
    person_ID NUMBER;
    person_name Name,
    PRIMARY KEY (person_ID));
```

Object Relational Mapping Frameworks

Recognising that most databases are relational, and that this is likely to remain the case for some time to come, much effort has been put in recently to object relational mapping frameworks.

Key features:

- the programmer can work only with objects – no SQL statements in the code
- selected objects are initially marked as being persistent – thereafter, changes in those objects are transparently changed in the database, and the programmer does not have to write code specifically to update the database
- the framework handles the mapping of the objects to relational database tables where they are actually stored
- mapping of objects to database tables is usually defined in XML descriptor files



The Java Community, including Sun Microsystems, who originally created Java, have developed **Java Data Objects (JDO)**, an object relational mapping API for Java. The features of JDO include:

- Applications written to use JDO for persistence can be used with any database for which a JDO implementation is available.
- Queries are written in a Java-like language JDO Query Language (JDOQL).
- Mapping of objects to database tables is defined in XML descriptor files

Some OODBMS vendors, including POET and Versant have released products which are based on JDO.

Sun's Enterprise JavaBeans (EJB), an advanced server-side Java component architecture, has its own persistence mechanism, **Container Managed Persistence (CMP)** which works in a similar way but is part of a more complex system.

There are also open-source OR mapping frameworks which work in a similar way to JDO, including **Hibernate**, **ObjectRelationalBridge (OBJ)** and **Castor**. Commercial products such as **Toplink** make it easier to define mappings.

Note that some OR frameworks, including Hibernate and OBJ, are compliant with the **ODMG 3.0 standard** for interfacing with a database.

The following example shows a Hibernate mapping file for a class called *Team* which has attributes *name*, *stadium* and a collection of objects of class *Player*. *Team* objects will be stored in a database table called *teams*. The *name* attribute of the class will be stored in a column called *team_name*, and so on.

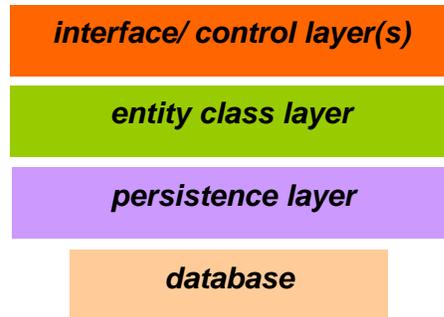
```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-
    2.0.dtd">

<hibernate-mapping>
    <class name="Team" table="teams">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name" column="team_name"
            type="string"/>
        <property name="stadium" column="stadium"/>
        <bag name="players" lazy="true" inverse="true"
            cascade="save-update">
            <key column="team"/>
            <one-to-many class="Player"/>
        </bag>
    </class>
</hibernate-mapping>
```

You can download the full code for this example from your course web site.

Writing a Persistence Layer

It is usually a good idea to **separate the entity classes**, which model the real world objects in a system, **from the persistence code** which stores and retrieves them from a database. Persistence code can be placed in separate classes which are closely associated with their entity classes. These classes are the **persistence layer** of an application.



a typical application architecture

The advantages of using a persistence layer include:

- Entity classes can be re-used in other applications which use different databases
- Entity classes are easier to read and understand without any database access code in them
- You can easily change the database you are using without changing the entity classes

Example

An application needs a *Player* class to represent a football player. The *Player* class has attributes *name* and *squadNumber*. The class could look like this:

```
public class Player extends Persistent {  
  
    private String name;  
    private int squadNumber;  
  
    public void setName(String name){  
        this.name = name;  
    }  
  
    public String getName(){  
        return this.name;  
    }  
}
```

```
public void setSquadNumber(int squadNumber){
    this.squadNumber = squadNumber;
}

public int getSquadNumber(){
    return this.squadNumber;
}

}
```

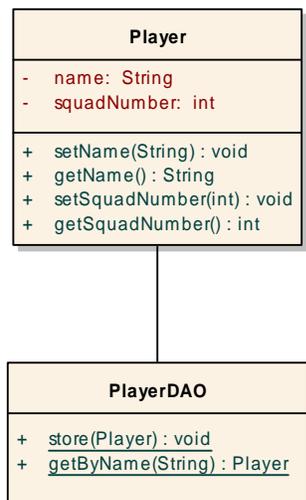
The application needs to store information about players in an ODBC database. How do we do this?

Firstly, we **do not add database access code to this class.**

Secondly, we must decide what information will need to be stored or retrieved. In this case, we probably would want to be able to do the following database actions:

- Store a *Player* object
- Retrieve the *Player* with a specified name.

To do this, we write a Data Access Object (DAO) class which can perform these actions. Each entity class in the system which needs to be persistent should have a DAO associated with it. The DAO for *Player* must have a method to perform each database action.



The code for the *PlayerDAO* class is as follows:

```
import java.util.*;
import java.sql.*;
import java.io.*;

/**
 * PlayerDAO - data access object for Player business object
 */
public class PlayerDAO {

    public static void store(Player p) throws Exception {

        Statement stmt = null;
        Long id = null;
        Connection con = dbConnection();

        try{
            stmt = con.createStatement();

            // create INSERT query string
            String insertQuery =
                "INSERT INTO PLAYERS (name, squadnumber) " +
                "VALUES ('" + p.getName() + "', " +
                p.getSquadNumber() + ")";
            // execute INSERT query
            stmt.executeUpdate(insertQuery);

            con.close();

        }
        catch (SQLException exc) {
            System.out.println("Error performing query: " +
                exc.toString());
        }

    }

    public static Player getByName(String name) {
        Player result = null;

        Statement stmt = null;
        Connection con = dbConnection();

        try{
            stmt = con.createStatement();

            // create SELECT query string and execute query on
            // teams table
            ResultSet rs = stmt.executeQuery("SELECT * FROM
                PLAYERS " +
                "WHERE name = '" + name + "'");
```

```
        // only expect one result - use first entry in
        // ResultSet
        Player player = new Player();
        if (rs.next()){
            player.setName(rs.getString(2));
            player.setSquadNumber(rs.getInt(3));
            result = player;
        }

        con.close();
    }
    catch (SQLException exc) {
        System.out.println("Error performing query: " +
            exc.toString());
    }

    return result;
}

private static Connection dbConnection(){
    Connection con = null;
    try{
        //Make sure the JdbcOdbcSriver class is loaded
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

        //Try to connect to database
        con =
            DriverManager.getConnection("jdbc:odbc:student");
    }
    catch (SQLException exc) {
        System.out.println("Error making JDBC connection: "
            + exc.toString());
    }
    catch (ClassNotFoundException exc) {
        System.out.println("Error loading driver class: "
            + exc.toString());
    }
    //Return Connection
    return con;
}
}
```

dbConnection is a method which sets up the database connection, to an ODBC data source. Note that the widely used term Data Access Object is not strictly correct for this example as all the methods are static so no instance of the *PlayerDAO* class needs to be created.

An application can then use the *PlayerDAO* class to store and retrieve player objects, for example:

storing...

```
Player p1 = new Player();
p1.setName("Zinedine Zidane");
p1.setSquadNumber(10);
```

```
PlayerDAO.store(p1);
```

retrieving...

```
Player player = PlayerDAO.getByName("Zinedine Zidane");
if (player != null)
    System.out.println("Squad number for " +
        player.getName() + " is " + player.getSquadNumber());
else
    System.out.println("Player not found");
```

Changing the database

Since only the *PlayerDAO* class contains any reference to the database, we can change the application to use a completely different persistence method without changing any other classes.

The following version of *PlayerDAO* uses **db4o**, an object database. *db4o* stores data using the object model in a single database file. To use it, the *db4o.jar* class need to be included in the project. *db4o* is an example of an object database which is not ODMG 3.0 compliant. It can retrieve data using the “query-by-example” method shown in this code. Note that there is no SQL in this code – objects are stored and retrieved using *set* and *get* methods.

```
import java.util.*;
import com.db4o.*;
import java.io.*;

/**
 * PlayerDAO - data access object for Player business object
 */
public class PlayerDAO {

    public static void store(Player p) throws Exception {
        // open database file in current project directory
        Db4o.licensedTo("bell.ac.uk");
        File file = new File("teams_db4o.yap");
        String fullPath = file.getAbsolutePath();
        ObjectContainer db = Db4o.openFile(fullPath);

        // store player - also stores associated players
        db.set(p);

        // close database
        db.close();
    }
}
```

```
public static Player getByName(String name) {
    Player result = null;

    // open database file in current project directory
    Db4o.licensedTo("bell.ac.uk");
    File file = new File("teams_db4o.yap");
    String fullPath = file.getAbsolutePath();
    ObjectContainer db = Db4o.openFile(fullPath);

    // query by example - create example player object
    Player examplePlayer = new Player();
    examplePlayer.setName(name);

    // retrieve object
    ObjectSet set = db.get(examplePlayer);
    if (set.hasNext()){
        result = (Player)set.next();
    }

    // close database
    db.close();

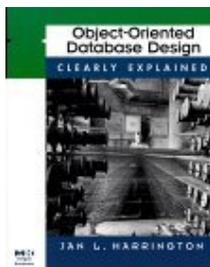
    return result;
}
}
```

EXERCISE 3.

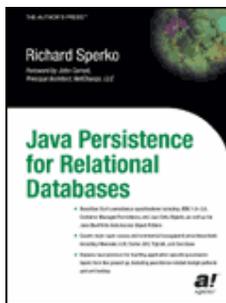
1. Download the file *player_jdbc_dao.exe* from your course web site and extract the contents. Create a new NetBeans project called *player_jdbc_dao* and mount the folder you extracted.
2. The database file *student.mdb* needs to be associated with a Windows DSN *student*. On a college PC this can be done by copying the file to the folder *My Databases* inside *My Documents* (overwrite the existing file if there is one).
3. Execute the class *Create*. Open the database in Access and look at the contents of the *players* table.
4. Execute the class *Retrieve*. Compare the results with the database.
5. A new requirement is added to the application to retrieve a list of all the players in the database. Add a new method *getAll()* to *PlayerDAO* which returns a *List* of *Player* objects. Add code to *Retrieve* to call this method and display the results. Test your updated project.

6. Download the file *player_jdb4o_dao.exe* from your course web site and extract the contents. Create a new NetBeans project called *player_db4o_dao* and mount the folder you extracted. Mount the file *db4o.jar* which is in this folder as an archive.
7. Execute the class *Create* and then execute the class *Retrieve*. Check that a *Player* object is retrieved.
8. Again, add a new method *getAll()* to *PlayerDAO* which returns a *List* of *Player* objects. Add the same code as before to *Retrieve* to call this method and display the results. Test your updated project.

Further Reading



Object-Oriented Database Design Clearly Explained by Jan L. Harrington (Morgan Kaufmann 2000)



Java Persistence for Relation Databases by Richard Sperko (APress 2003)

The Object-Oriented Database System Manifesto

M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, S. Zdonik

1989 Proceedings of the First International Conference on Deductive and Object Oriented Databases