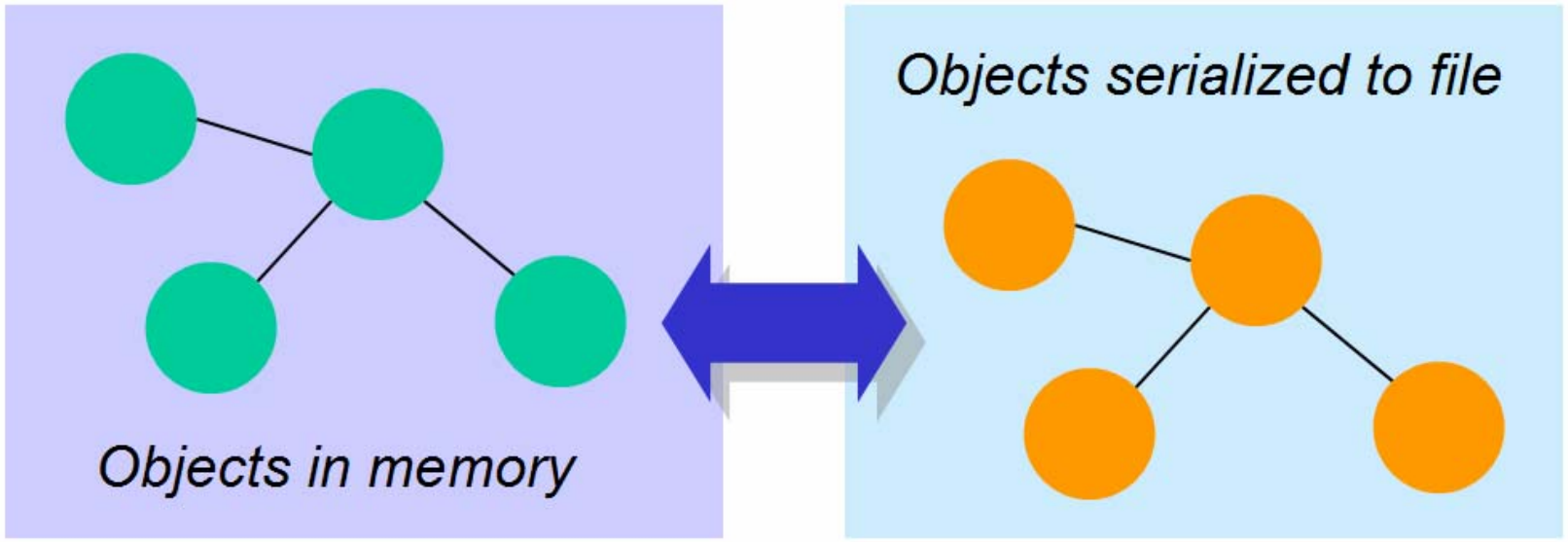
Object Persistence

Introduction

- One of the most critical tasks that applications have to perform is to save and restore data
- **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

- Simple persistence method which provides a program the ability to read or write a whole object to and from a stream of bytes
- Allows Java objects to be encoded into a byte stream suitable for streaming to a file on disk or over a network
- The class must implement the *Serializable* interface (*java.io.Serializable*), which does not declare any methods, and have accessors and mutators for its attributes

Object Serialization

```
// 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);
```

Using Databases

- Most Client-Server applications use a RDBMS as their data store while using an object-oriented programming language for development
- Objects must be mapped to tables in the database and vice versa
- Applications generally require the use of SQL statements embedded in another programming language
- “Impedance mismatch”

Using Databases

- Goal of object-oriented design is to model a process
- 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
  - object inheritance
- The resulting tables may store data inefficiently, or access to data may be inefficient

Using Databases

- There are essentially three approaches which have been developed for the management of object storage in databases:

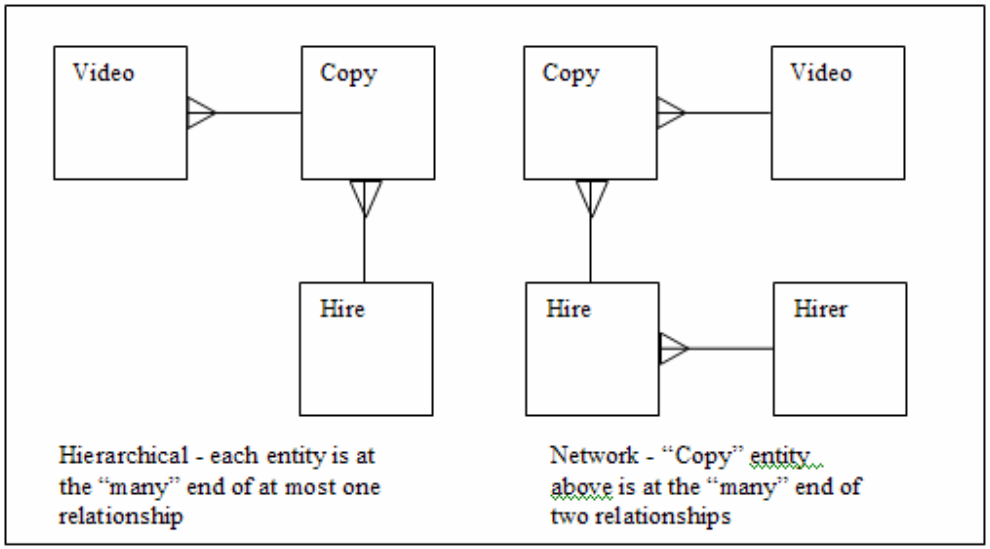
- **the Object-Oriented Database Management System** (OODBMS)
- **the Object-Relational Database Management System** (ORDBMS)
- **Object Relational Mapping**

Data Models: 1st 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.

Data Models: 1st Generation

- **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



Data Models: 2nd 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
- does not have the full power of a programming language
- Applications generally require the use of SQL statements embedded in another programming language.

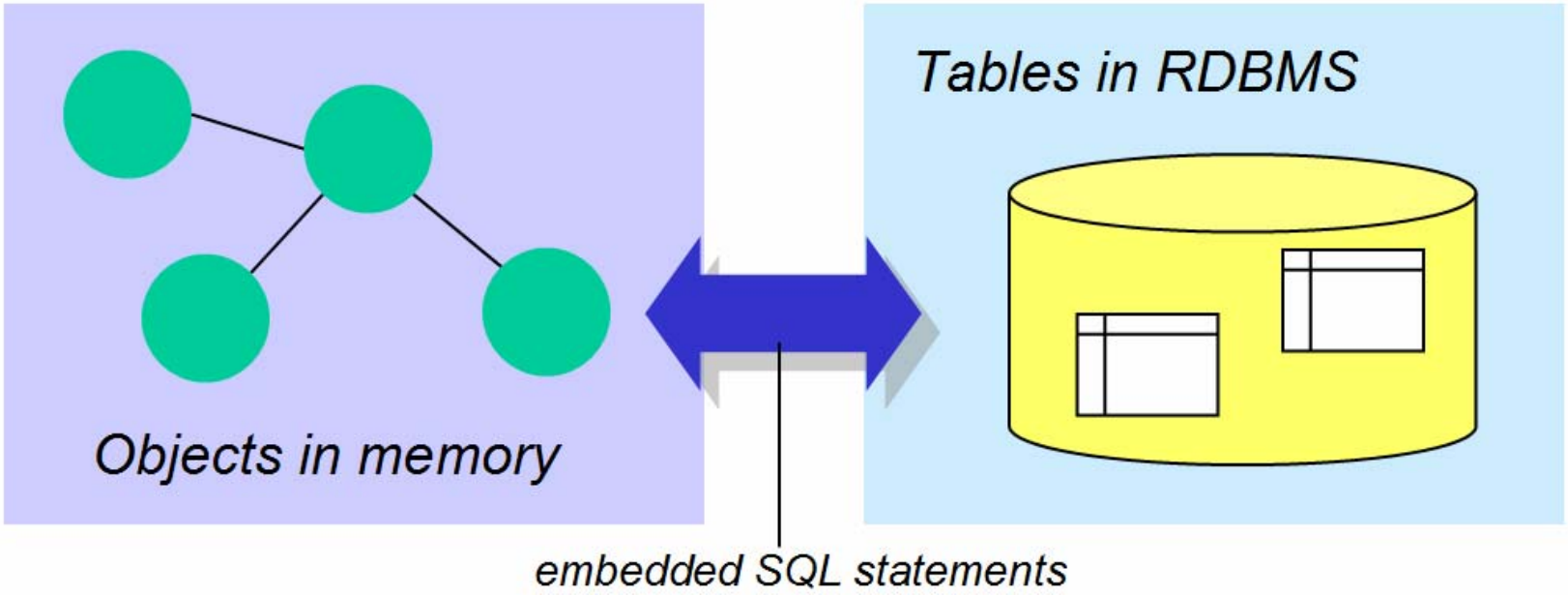
Data Models: 3rd Generation

- most programming languages are now object-oriented
- **Object Data Model**
  - offers **persistence** to objects, including their associations, attributes and operations
  - 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
- **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

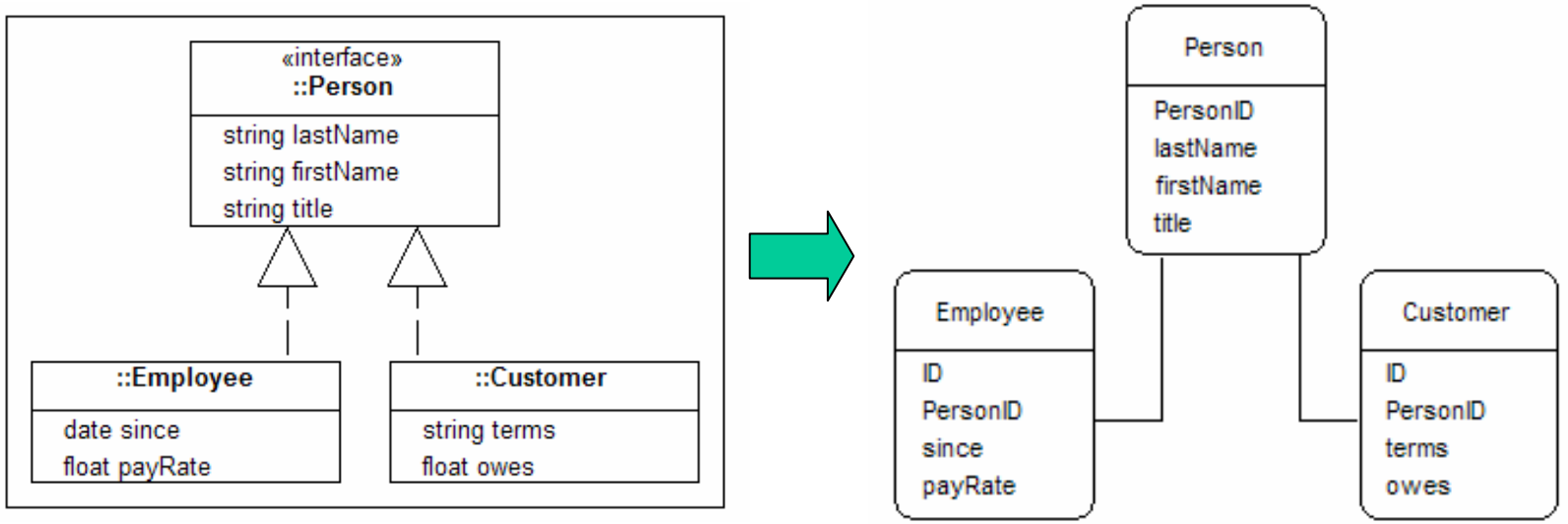
- Most business database applications use relational databases
- Need to map the objects in the application to tables in the database
- Sometimes be a simple matter of mapping individual classes to separate database tables
- However, if the class structure is more complex, then the mapping must be carefully considered to allow data to be represented and accessed as efficiently as possible

Object Relational Mapping



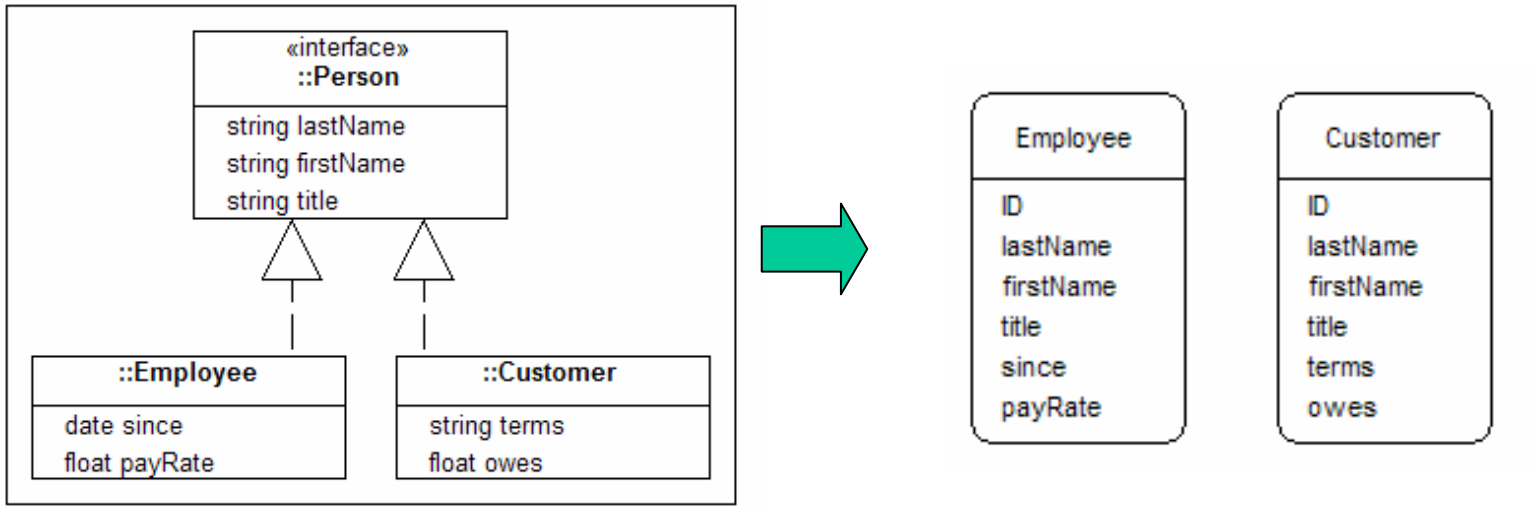
OR Mapping: Inheritance

- Vertical mapping



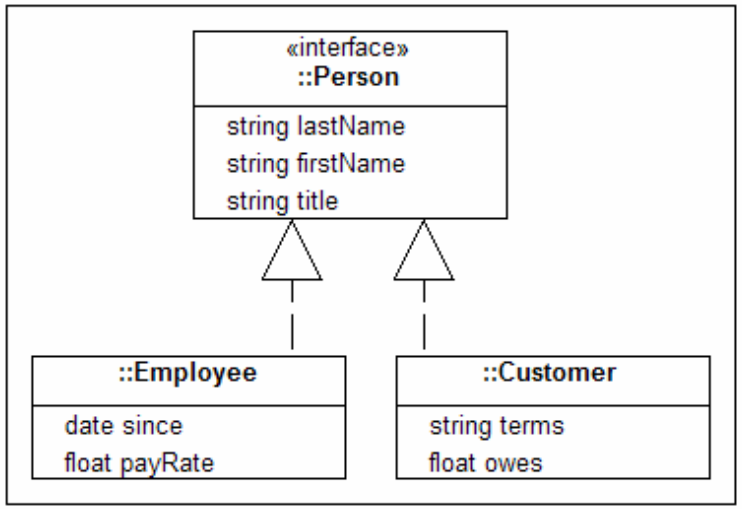
OR Mapping: Inheritance

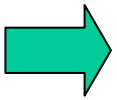
- Horizontal mapping

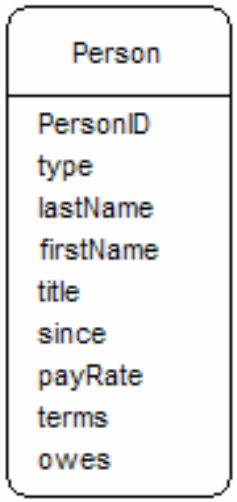


OR Mapping: Inheritance

- Filtered mapping



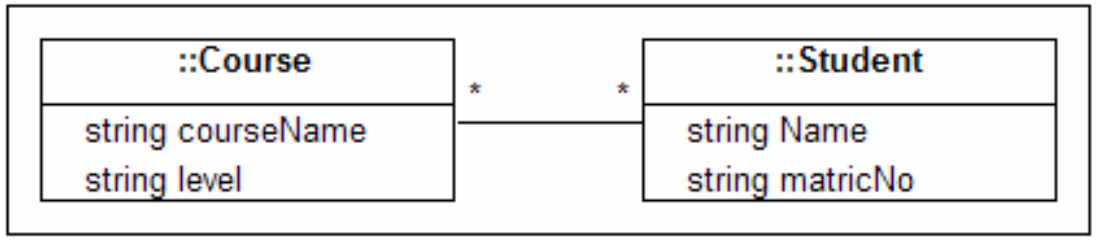




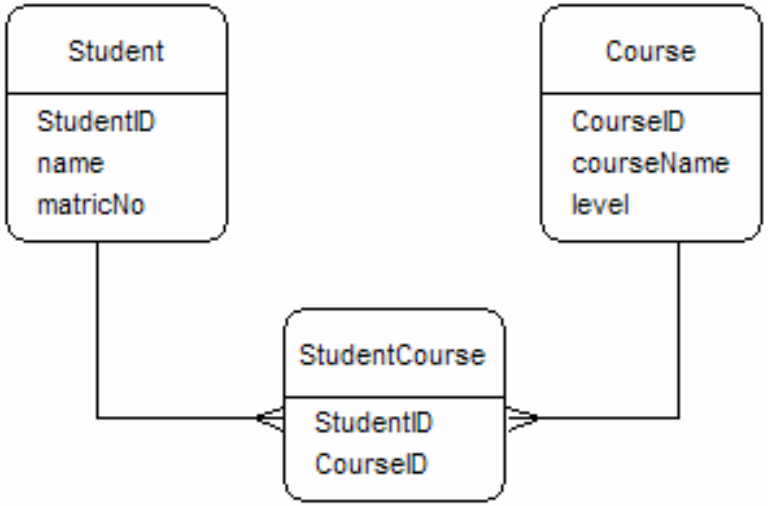
OR Mapping: Guidelines

- 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

OR Mapping: Many-to-Many



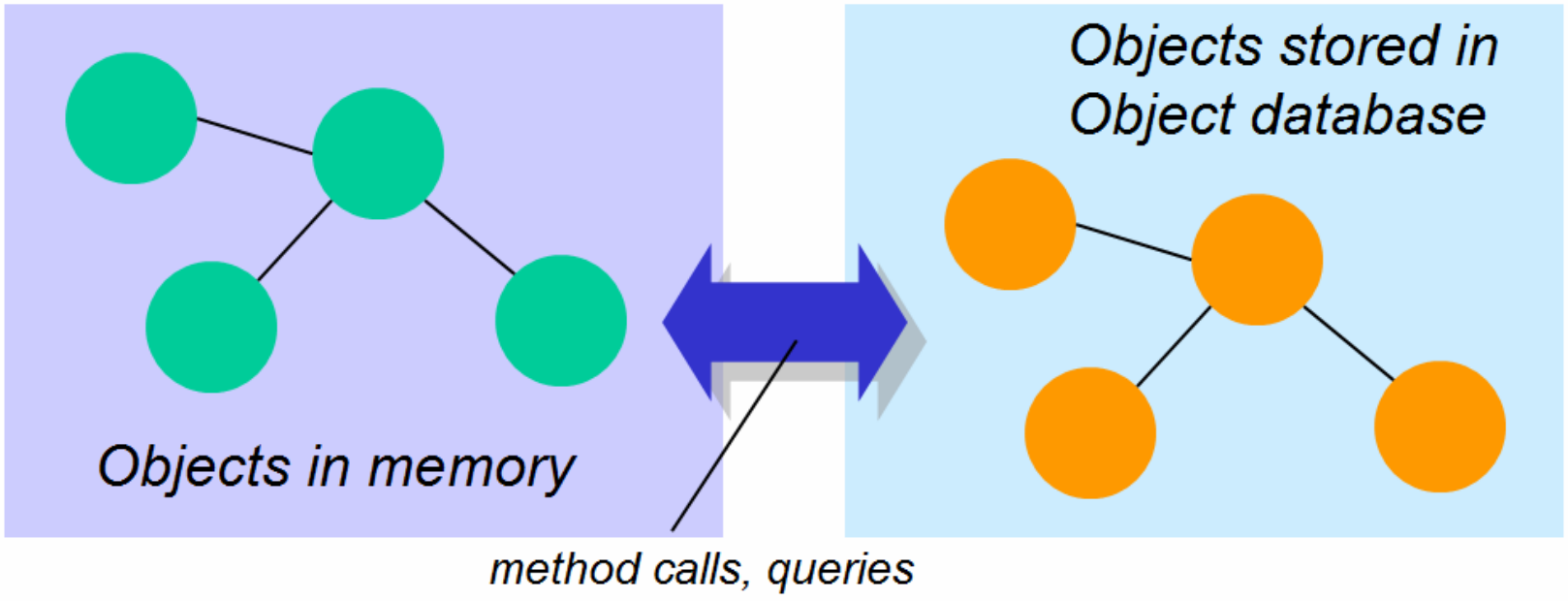
In a relational database a join table is required to represent this relationship:



Object Data Model

- Object-oriented database systems are based on the concept of **persistent objects**
- Use class declarations similar to those used by object-oriented programming languages
- Class declarations should additionally indicate relationships between objects.
- The system must be able to represent traditional database relationships and also relationships unique to the object-oriented model, such as inheritance

Object Data Model



Object Identifiers

- RDBMS
  - entities are uniquely identified by primary keys
  - relationships are represented by matching primary key-foreign key data
  - Identification *depends on the values in the key fields*
- 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*

Object Identifiers

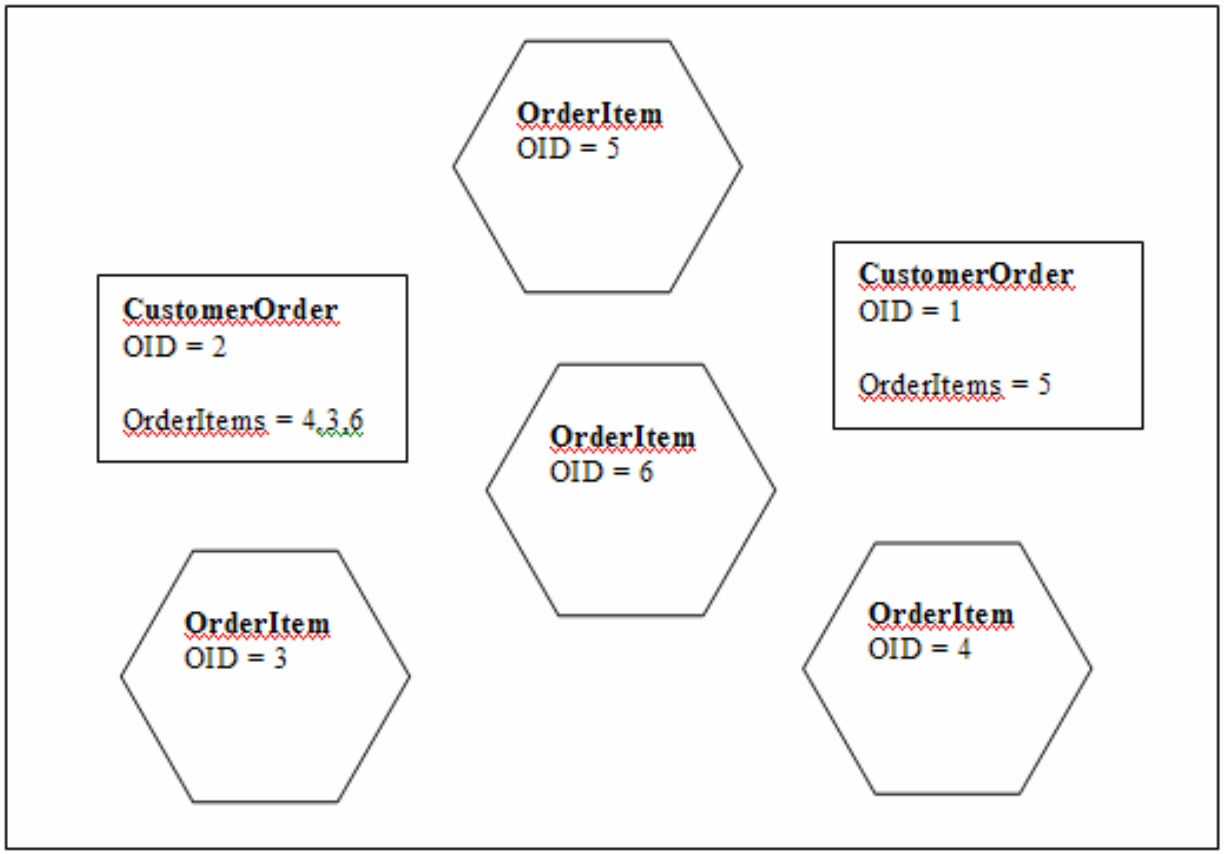
- The fact that an object's identity is distinct from its values means that the concepts of equivalence and equality are different:
- **Equivalent:** same OID
- **Equal:** same state values

- 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



Representation of Relationships

- In the diagram all relationships have **inverse relationships**.
- 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*
- The inverse ensures that an *OrderItem* is associated with only one *Order*.

Representation of Relationships

- Only relationships predefined by storing OIDs can be used to query or traverse the database
- The database is therefore *navigational*
- 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*

Relationships: One-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)

Relationships: Many-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

Relationships: “Is A” & “Extends”

- These relationships can be represented because the object-oriented paradigm supports inheritance
- 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
- 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;
   }
```

Relationships: “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

Object Database Standards

- **The Object Oriented Database Manifesto (1989)**

- **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

Object Database Standards

- ***The ODMG Proposed Standard***
- One of the crucial factors in the commercial success of RDBMSs is the relative standardisation of the data model
- The Object Data Management Group (ODMG) was formed by a group of industry representatives to define a proposed standard for the object data model.
- It is still far from being as widely recognised as the relational database standards.
- The ODMG proposed standard defines the following aspects of an OODBMS:
  - 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)

Advantages of OODBMS

- **Complex objects and relationships**
- **Class hierarchy**
- **No impedance mismatch**
- **No need for primary keys**
- **One data model**
- **One programming language**
- **No need for query language**
- **High performance for certain tasks**

Disadvantages of OODBMS

- **Schema changes**
- **Lack of agreed standards**
- **Lack of ad-hoc querying**

- 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)

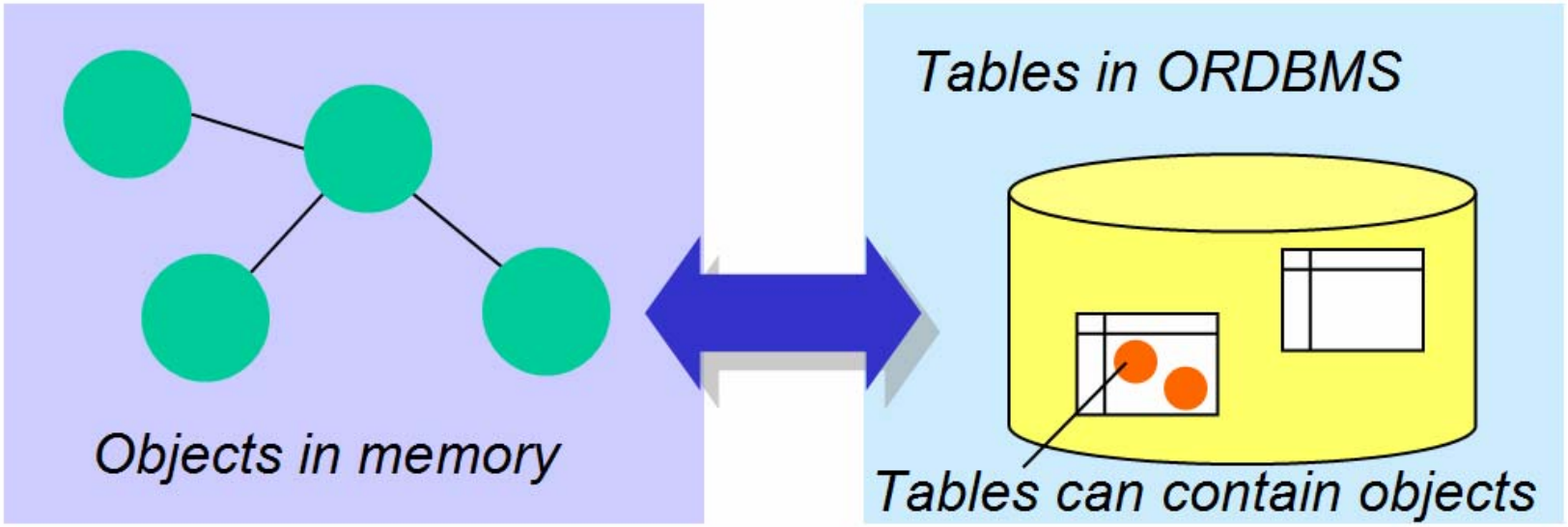
OODBMS Users

- 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



The Object Relational Model

- 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

ORDB Example - Oracle

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

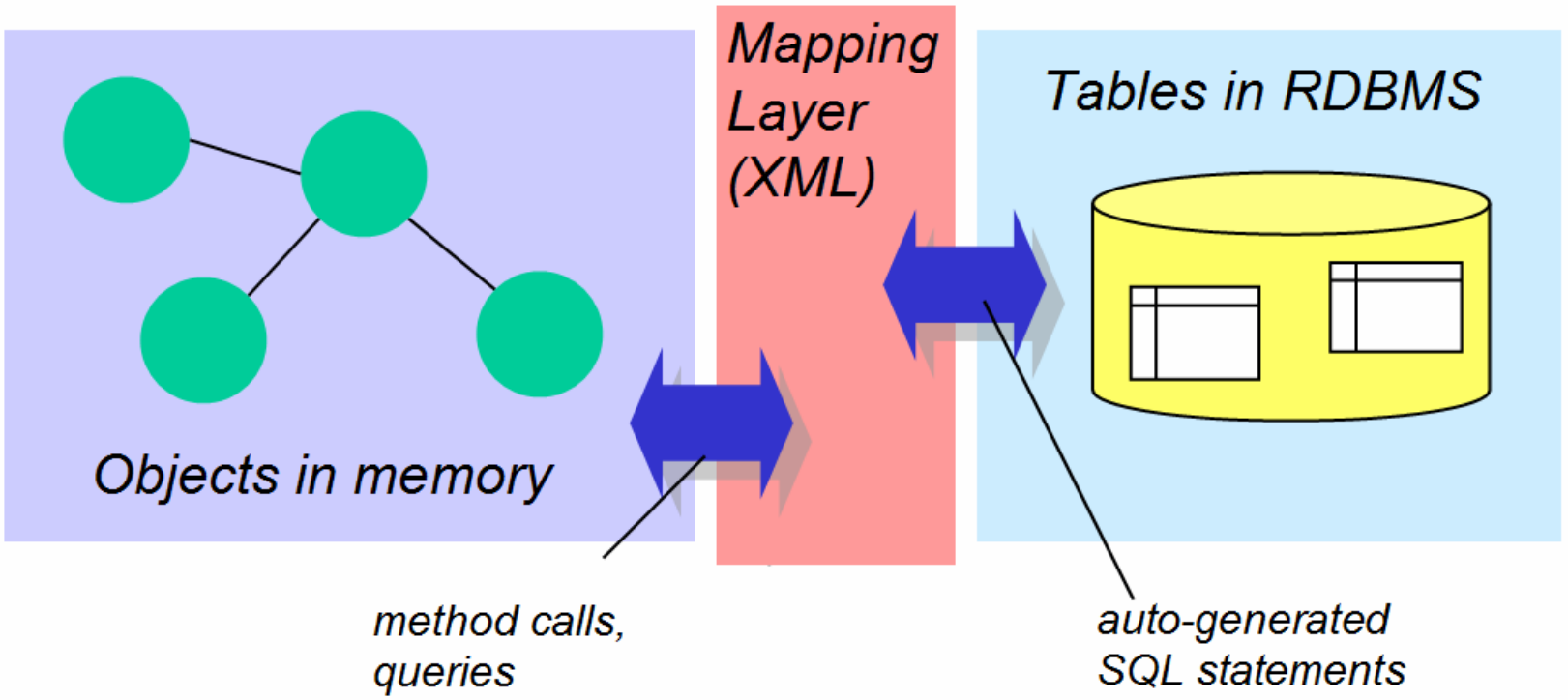
Code to define operations – in this case simply a class constructor

```
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));
```

OR Mapping Frameworks



OR Mapping Frameworks

- Most databases are relational
- Much effort has been put in recently to making OR mapping more convenient
- Transparent persistence

OR 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

OR Mapping Frameworks

- **Java Data Objects (JDO**
  - 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.

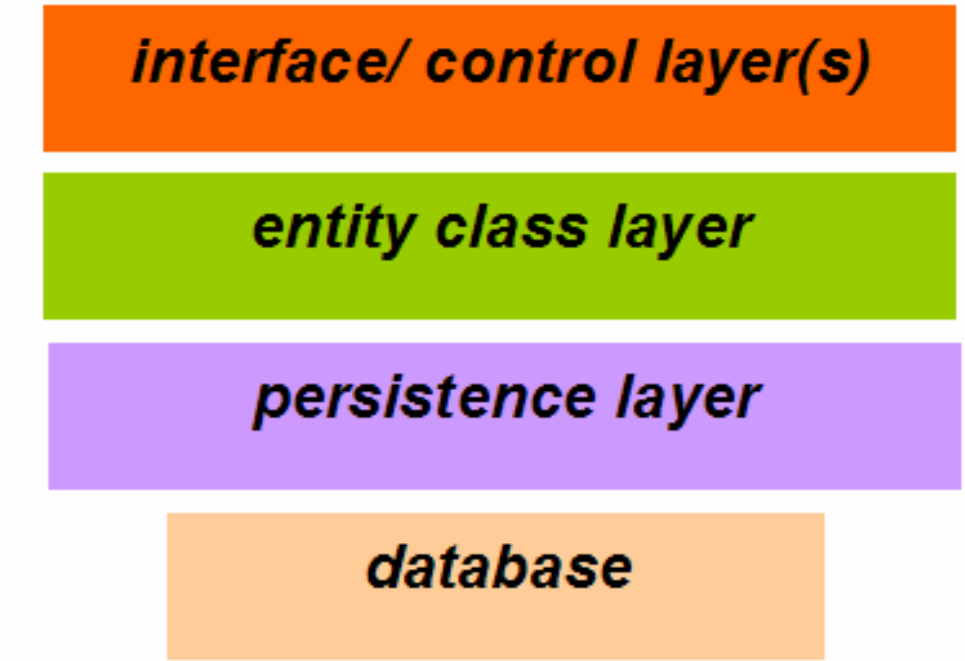
OR Mapping Frameworks

- Sun's Enterprise JavaBeans (EJB), an advanced server-side Java component architecture, has its own persistence mechanism, **Container Managed Persistence (CMP)**
- There are also open-source OR mapping frameworks which work in a similar way to JDO, including **Hibernate**, **ObJectRelationalBridge (OJB)** and **Castor**.
- Commercial products such as **Toplink** make it easier to define mappings.
- Some OR frameworks, including Hibernate and OJB, are compliant with the **ODMG 3.0 standard** for interfacing with a database.

Hibernate Mapping Example

```
<?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>
```

Writing a Persistence Layer



a typical application architecture

Advantages of Persistence Layer

- 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

Persistence Layer Example

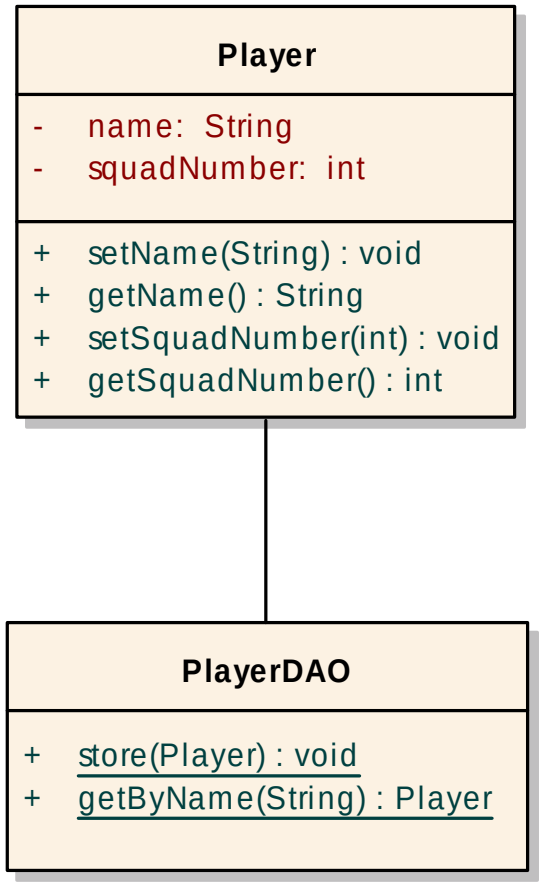
- Class Player
- Has attributes name and squadNumber
- The application needs to store information about players in an ODBC database.
- How do we do this?

Persistence Layer Example

- 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

Persistence Layer Example

- 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



Persistence Layer Example

```
 public static void store(Player p) throws Exception {

…

try{
   stmt = con.createStatement();

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

…

}
```

Changing the Database

- Since only the PlayerDAO class contains any reference to the database
- So 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
- Note that there is no SQL in this code – objects are stored and retrieved using *set* and *get* methods

Changing the Database

```
public static void store(Player p) throws Exception {
   …
   ObjectContainer db = Db4o.openFile(fullPath);

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

   …

}
```

Further Reading