

# Java Persistence Architecture

For more details see the note JPA.pdf on website

<http://www.uow.edu.au/~nabg/399/JPA/JPA.pdf>  
And try the exercise  
<http://www.uow.edu.au/~nabg/399/E5.html>

# JPA

- Java Persistence Architecture
  - Extra level in “software stack” needed to access data in relational tables
    - JPA
    - JDBC
    - Database driver
  - Hides messy implementation detail of JDBC access
  - Allows more uniform “object model”
  - Facilitates automatic code generation

# Evolution-1 : EJB

- Automated persistence
  - “Entity” classes – correspond to entities in database model
  - Enterprise Java Beans (1998..) introduced “Entity Beans”
    - You could code these manually, writing JDBC code
    - They could be partially generated from meta-data describing your database tables
  - EJB-1/EJB-2 model for entity beans
    - Not popular
    - Requires that the beans are based on classes and interfaces defined in EJB model
      - Complex
      - Intrusive

# Evolution-2 : others

- From ~2000, lots of independent projects developing alternative object-relational models
  - JDO, Kodo, Hibernate, ...
  - Spring framework using Hibernate
- JPA
  - Part of official Java standard
  - Draws on this independent work

# What's wrong with JDBC

# JDBC problems-1: the code

- Too low level
  - Load driver
  - Get connection
  - Create statement object
  - Assemble SQL query as text string
  - Run query
  - Foreach row in resultset
    - Unpack data for each column
    - Insert into some bean structure

## Setting up code

```
Class.forName(dbDriverName);
dbConnection =
    DriverManager.getConnection(
        dbURL, userName, userPassword);

// Create prepared statements as well
pstmtSpecificTeacher =
    dbConnection.prepareStatement(getSpecificTeacher());
```

## Get data item given primary key

```
private static Teacher getTeacher(Long id) {
    Teacher t = null;

    // Attempt to retrieve Teacher record given employment
    // number
    try {
        pstmtSpecificTeacher.setLong(1, id);
        ResultSet rs = pstmtSpecificTeacher.executeQuery();
        if (rs.next()) {
            t = new Teacher(
                rs.getString("EMPLOYEENUMBER"),
                rs.getString("TITLE"),
                rs.getString("SURNAME"),
                rs.getString("FIRSTNAME"),
                rs.getString("INITIALS"),
                rs.getString("SCHOOLNAME"),
                rs.getString("ROLE"));
        }
        rs.close();
    } catch (SQLException ex) { ... }
    return t;
}
```

## Writing data

```
pstmtAddTeacher.setString(1, empNumber);
pstmtAddTeacher.setString(2, surName );
pstmtAddTeacher.setString(3, firstName );
pstmtAddTeacher.setString(4, initials);
pstmtAddTeacher.setString(5, title );

pstmtAddTeacher.executeUpdate();
```

## JDBC problems-2: the model

- Typically, "bean" classes invented by programmer to represent database entities do not really capture the relationships amongst the entities
- Example
  - Simple entity
    - Teacher associated with school
      - Teacher table has foreign key schoolname referencing School table
    - Java class
      - Teacher object has String data member for schoolname: if need details of School for teacher then will need to perform another access to database to load School record
  - More complex entity
    - Conceptually School "owns" collection of teachers
    - Not represented at class level

## Can JPA do better?

Can we fix it?  
Yes we can.



## That code ...

- The code needed for JDBC is very stereotyped
  - Read: grab data from column of returned row, place in member of newly created object
  - Multiple rows in ResultSet? OK, create instance of collection class and add each new object to collection
- Relatively easy to generate such code automatically
  - Need a class defined that matches structure of row and accessor/mutator functions that provide read/write access to data members
  - "Reflection" can be used to invoke member functions in uniform manner
- Just need to read meta data describing table

## Mapping restrictions?

- Objects are “unique” – need to match unique rows
  - Rows must have primary key
    - Natural primary key – e.g. employee-number
    - Composite primary key – e.g. (date-of-match, stadium)
    - Faked primary key – e.g. shopping cart items can have key generated as cartseq.nextval for some sequence associated with table

## Entity relationships - 1

- Teacher references school
  - Class Teacher
    - Not String schoolname
    - Instead School schoolname
  - When load instance of Teacher, also automatically load instance of School
  - Then can traverse object references and easily print address where Teacher works

## Entity relationships - 2

- School “owns” collection of Teachers (and collection of Pupils).
- OK, define in class:

```
public class School implements Serializable {  
    private String schoolName;  
    private String address;  
    private Collection<Teacher> staff;  
    private Collection<Pupil> pupils;  
    public String getSchoolName() { return schoolName; }  
    ...  
}
```

## Wait a moment!

- Load specific Teacher employee number 2003110672
  - Teacher works at School “Wendell Public”, so load School record
    - School Wendell Public has 35 teachers and 702 pupils;  
so load all 34 colleagues of Teacher 2003110672 and all 702 pupils she might see
  - This doesn't seem sensible.

## Lazy load

- Collections don't need to be loaded until actually used
  - Effective definition of School's `Collection<Teacher> getStaff()` is a little more elaborate than the apparent “return staff” code.  
Mechanism can be hidden by using some auto-generated subclass of class Collection that does the loading when needed.

Occasionally, though pretty rarely, it may be appropriate to “eager” load collections.

## Changing objects?

- You load Teacher object
- You change School where employed – by changing field in object
- What about the data tables?
- Well, if **something** watches over the objects that were loaded from tables it can notice changes and synchronize persistent store



## JPA fixes it

- EntityManager
  - "Intelligent" connection to database
    - Intelligence?
      - E.g. methods like
        - *"find an object of class X by searching XTable for row with primary key aaaaaa"*
        - *"return a collection of Teacher objects that correspond to rows in TeacherTable where School foreign key column contains null"*
  - Responsible management of entities
    - *"Object x loaded from table t has been changed, I'll run an update on database when I next get a chance."*



## How does fix work? 1

- Need to define matching classes and tables
  - *Easiest approach (for most developers)* :
    - define your tables
    - use a utility that will interpret table meta-data to generate class definitions

You can define classes, then generate tables



## How does fix work? 2

- Run time needs to know
  - *What is primary key?*
  - *What collection members and how should they be treated (lazy/eager)?*
  - *What are the actual column names? (Naming conventions for your classes and your tables may differ, so you may want to use similar but distinct names.)*
- So, arrange for necessary meta-data to be available at run time
  - Supplementary configuration file?
  - Extra supplementary data in the class definitions themselves?
  - Both schemes? Select whichever more convenient for particular application



## How does fix work? 3

- Runtime's connection to the database
  - *Need: URL, driver classes, user name and password*
- So, JNDI and datasource maybe?
  - Well, not that convenient
    - Remember how these tend to be rather container specific (e.g. name was jdbc/unionacle or java.comp/env/jdbc/unionacle)
- How about a little configuration file that will get read in by runtime;
  - put it somewhere standard – fixed name, stored along with codebase files



## How does fix work? 4

- Synchronizing with database
  - Reading data?
    - No problems here, use "read committed", multiple concurrent readers allowed
  - Writing data?
    - Better use database transaction system properly
    - Do updates when "commit" transaction.

## Examples

Database tables and classes

## Soccerleague example

Simple entity

## League - Oracle

```
create sequence leagueseq increment by 1 start with 1;
create table soccerleague (
  gameid number primary key,
  played date,
  location varchar(64),
  team1 varchar(32),
  team2 varchar(32),
  score1 number,
  score2 number );
```

```
insert into soccerleague values ( leagueseq.nextval,
  TO_DATE('2007-08-26','YYYY-MM-DD'),
  'Members Equity Stadium', 'Perth', 'Newcastle', 0, 0);
```

## League – generated Java class

```
@Entity
@Table(name = "SOCCERLEAGUE")
@NamedQueries({
  @NamedQuery(name = "Soccerleague.findByGameid",
    query = "SELECT s FROM Soccerleague s WHERE s.gameid = :gameid"),
  ...})
public class Soccerleague implements Serializable {
  private static final long serialVersionUID = 1L;
  @Id
  @Column(name = "GAMEID", nullable = false)
  private BigDecimal gameid;
  @Column(name = "PLAYED")
  @Temporal(TemporalType.DATE) private Date played;
  ...
  public Soccerleague() { }
  public Soccerleague(BigDecimal gameid) { this.gameid = gameid; }
  public BigDecimal getGameid() { return gameid; }
  ...
}
```

## Generated class

- Class Soccerleague **implements Serializable**
  - Remember Serializable – the empty interface
    - Soccerleague has no extra methods? !
  - **Misuse of inheritance (implements) tag really**
- **"implements Serializable"**
  - Interpreted by Java compiler as flagging need to generate some extra methods for this class
    - readObject, writeObject etc

Why implements Serializable?  
Because often need to transfer entity objects across net from server back to client

## Annotations – better than ad hoc compilation tags!

- Annotations introduced much later
- Same idea really
  - Remind compiler that it has to generate some extra code (or modified code)
- "Annotation tag" classes provide the information that compiler will need
  - Many standard annotations defined
  - You can define your own

## Generated class - annotations

- Annotations used (all classes defined in javax.persistence package):
  - @Entity
  - @Table
  - @NamedQuery
  - @Id
  - @Column
- Compiler uses class definitions and data supplied in tags to create meta-data and otherwise modify code generated for the class.

## Generated class - annotations

- Examples
  - @Table
    - Definition really says that can specify "catalog", "schema" etc
    - These to appear as attributes of @Table class
  - @Id
    - Tag the data member that represents primary key
  - @Column
    - Attributes allow definition of column name, flag if it is "nullable", give size for string (varchar) types etc

## League – generated Java class

```

@Entity
@Table(name = "SOCCERLEAGUE")
@NamedQueries( ...)
public class Soccerleague implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Column(name = "GAMEID", nullable = false)
    private BigDecimal gameid;
    @Column(name = "PLAYED")
    @Temporal(TemporalType.DATE)
    private Date played;
    @Column(name = "LOCATION")
    private String location;
    @Column(name = "TEAM1")
    private String team1;
    ...
    
```

Definition of Entity class starts  
Maps to table SOCCERLEAGUE

Primary key is a Number, column  
name GAMEID  
"PLAYED" is a date

"LOCATION" is a varchar

## League – generated Java class

```

public class Soccerleague implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Column(name = "GAMEID", nullable = false)
    private BigDecimal gameid;
    @Column(name = "PLAYED")
    ...
    }
    
```

Accessor and mutator functions generated for  
data members

```

public Date getPlayed() { return played; }
public void setPlayed(Date played) { this.played = played; }
    
```

## Generated queries

- Will obviously want to run queries against table
  - Return all games played at stadium X
- Queries will be similar to SQL queries but they are queries about objects not collections of column values; so syntax should be similar to but not identical to SQL
- Java Persistence Query Language

## League – generated query

```

@NamedQueries({
    @NamedQuery(
        name = "Soccerleague.findByGameid",
        query = "SELECT s FROM Soccerleague s WHERE s.gameid = :gameid"),

    @NamedQuery(
        name = "Soccerleague.findByPlayed",
        query = "SELECT s FROM Soccerleague s WHERE s.played = :played"),
    )
    
```

## Generated query

- These query things will can be used by EntityManager to retrieve objects.
- Utility that generates class definition from table creates a set of simple queries
- You can define your own queries in JPQL
  - Want the drawn games?
    - SELECT s FROM Soccerleague s WHERE s.score1 = s.score2
- You can even define and use queries in SQL but that is rare

## Code using a generated Entity

- You want all games involving a team that you are following (team may appear as team1 or team2 in table)
- So ask entity manager to find list of games with team1="myteam", and merge results with list got when ask for team2="myteam"

```
EntityManager em = emf.createEntityManager();
Query q = em.createNamedQuery("Soccerleague.findByTeam1");
q.setParameter("team1", myTeam);
List liz1 = q.getResultList();
q = em.createNamedQuery("Soccerleague.findByTeam2");
q.setParameter("team2", myTeam);
List liz2 = q.getResultList();
liz1.addAll(liz2);
```

emf – instance of EntityManagerFactory

## That code

- Used "EntityManagerFactory" (explained shortly) to create an EntityManager
- Used the EntityManager to prepare a query – based on information in the NamedQuery generated automatically and defined as part of class in the annotations
- Set the parameters for the query (actual name of team of interest)
- Ran the query
  - Getting back a Collection<SoccerLeague> objects

## EntityManagerFactory and EntityManager

- EntityManager
  - "Intelligent" database connection
  - Runs the queries
  - Watches over the objects loaded looking for changes
  - If given opportunity, will reflect changes back into data tables
  - Used for a "session" then disposed of
- EntityManagerFactory
  - Knows about how to connect to database
  - Creates EntityManager objects when asked

## Getting connection data

- EntityManagerFactory
  - Needs information about database – URL, user-name/password, driver class, ...
  - Needs to have these data loaded before it gets asked for them
- The data
  - Stored in that little configuration file mentioned earlier (let's give it a name: persistence.xml)
  - Need to read in this file and use data to initialize a factory object that will then be used by program

## Intialization

- Intialization section of program will have to create an EntityManagerFactory and get it to load details of "persistence unit" from the xml configuration file

```
public class Main {
    private static EntityManagerFactory emf;

    private static void setupDataPersistence()
    {
        emf = Persistence.createEntityManagerFactory(
            "persistenceunitname");
        em = emf.createEntityManager();
    }
}
```

## Virtues of a programmer

- Remember **Larry Wall's** characterization of the virtues of a programmer:
  - *Most of you are familiar with the virtues of a programmer. There are three, of course: laziness, impatience, and hubris.*
- Programmer:
  - *Do we really need to write out stock standard initialization code? Couldn't the system just know we want initialization done and do it for itself?*

## "Injection"

- Injection
  - Means use an annotation tag that is interpreted as **Get the "container" to initialize this data member**
  - Of course, can only use injection if code will be running in a "container" that can be asked to perform initialization
    - No "container" for standard Java applications
    - Servlet container (or EJB container) can be asked to do such work

## CheckTeamServlet

```
public class CheckTeamServlet extends HttpServlet {
    ...
    @PersistenceUnit(unitName="JSPJPAApp1PU")
    private EntityManagerFactory emf;
    protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException
    {
        String myTeam = request.getParameter("myteam");
        ...
        EntityManager em = emf.createEntityManager();
        Query q = em.createNamedQuery("Soccerleague.findByTeam1");
        q.setParameter("team1", myTeam);
        List liz1 = q.getResultList();
        ...
    }
}
```

## javax.persistence.Query

- Useful methods like:
  - Object getResult
  - List getResultList
- Tidying up all that JDBC junk with its statements and ResultSets that gave back rows rather than objects

## "Seven a side soccer" example

Related entities!

## Oracle

- Two tables
  - One defines teams –
    - Not much data, just a teamid and a string name
  - Other defines people who participate –
    - Name, gender, etc
    - Also a foreign key referencing teamid of team table – "What team does this person play for?"
- Implicitly, team owns collection of players

## Table definition

```
create sequence side7teamseq increment by 1 start with 1;
create sequence personseq increment by 1 start with 1;
create table side7team (
    teamid number primary key,
    teamname varchar(64) );
create table persondata (
    personid number primary key,
    name varchar(32),
    gender varchar(8),
    myteam number,
    constraint persondata_gender_check check (gender in ('Male', 'Female')),
    constraint person_team foreign key(myteam)
    references side7team(teamid)
);
```

## Generated class

```
@Entity
@Table(name = "SIDE7TEAM")
@NamedQueries(...)
public class Side7team implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id @Column(name = "TEAMID", nullable = false)
    @GeneratedValue(generator = "MySeq7")
    @SequenceGenerator(name = "MySeq7", sequenceName = "SIDE7TEAMSEQ",
        allocationSize = 1)
    private BigDecimal teamid;
    @Column(name = "TEAMNAME")
    private String teamname;
    @OneToMany(mappedBy = "myteam")
    private Collection<Persondata> persondataCollection;
    ...
    public Collection<Persondata> getPersondataCollection() { return persondataCollection; }
```

## Collection data member

- Utility has interpreted the foreign key relation defined in table meta-data and inferred that "a side7team owns a collection of persondata objects"
- Annotations and data members in generated class express this relation.

## Using the class

- Can now easily get listings of all players in a team
  - Ask EntityManager to load side7team object
  - Ask object for its collection of persondata objects (this triggers the lazy loading of persondata objects)
  - Use them

## Code from servlet

```
String myTeam = request.getParameter("myteam");
if(myTeam==null || myTeam.length()<1) { ... }
EntityManager em = emf.createEntityManager();
Query q =
    em.createNamedQuery("Side7team.findByTeamname");
q.setParameter("teamname", myTeam);
Side7team s7t = (Side7team) q.getSingleResult();
...
// forward s7t object onto a JSP that prints contents
```

## Schools example

### Updating an entry

For more details see the note JPA.pdf on website  
<http://www.uow.edu.au/~nabq/399/JPA/JPA.pdf>

## Changes to the tables

- These require transactions – change to table made when commit transaction.
- Transactions
  - Can use classes defined in javax.persistence; this appropriate for stand alone applications
  - Can use JTA transaction; this appropriate for sevlets (and is done for you with EJB sessions etc)

## Changes

- Insert new record
- Update a record
- Remove a record

## NewTeacherServlet

*"Inject" factory and transaction context*

```
public class NewTeacherServlet extends HttpServlet {  
  
    @PersistenceUnit(unitName="WebSchoolDemo2PU")  
    private EntityManagerFactory emf;  
    @Resource  
    private UserTransaction utx;
```

## NewTeacherServlet

```
protected void doPost(HttpServletRequest request,  
    HttpServletResponse response)  
    throws ServletException, IOException {  
    // Read and validate input data  
    String empNumStr = request.getParameter("enumber");  
    ...  
  
    // Create and initialize Teacher object  
    Teacher t = new Teacher();  
    t.setFirstname(firstName);  
    t.setEmployeeNumber(empNum);  
    ...
```

## NewTeacherServlet

```
...  
try {  
    // Establish transactional context  
    utx.begin();  
    EntityManager em = emf.createEntityManager();  
    em.persist(t);  
    utx.commit();  
    em.close();  
} catch(EntityExistsException eee) { ... }
```

Since it is a new Teacher must explicitly add to the collection (\*persist()\*)

## Updating

- If changing an object loaded from database:
  - em.find() or some other query to load it
  - Change fields
  - Start, and commit a transaction
- Don't need to do changes within the apparent transaction context – but it makes the code clearer

## Update example

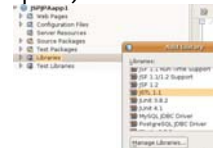
- Changing pupil record in servlet

```
EntityManager em = null;  
Pupil p = null;  
try {  
    utx.begin();  
    em = emf.createEntityManager();  
    p = em.find(Pupil.class, enrINum);  
    if(p==null) {  
        badData(response, "no such Pupil"); utx.rollback(); return;  
    }  
    p.setClassvl(classname);  
    if(schoolname==null)  
        p.setSchoolname(null);  
    else  
        p.setSchoolname(em.find(School.class, schoolname));  
    utx.commit();  
} catch(...) { ... }  
finally { em.close(); }
```

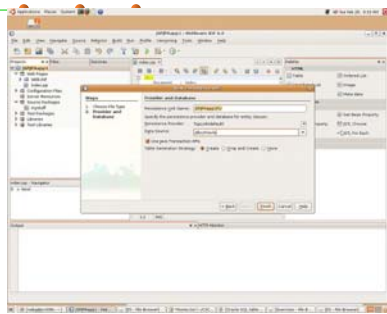
## Doing it in Netbeans

## Netbeans

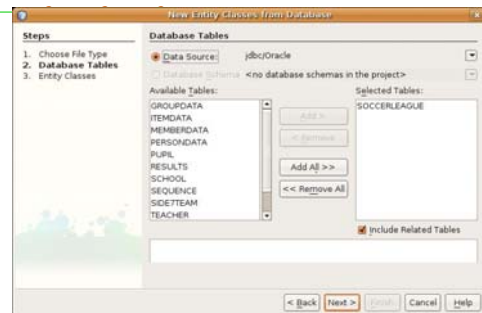
- Create new project, type = web application
- Add required libraries
  - Database driver
  - JPA implementation (Oracle toptlink)



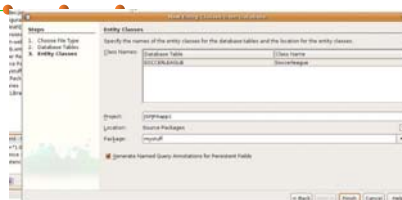
## Define a "persistence unit"



## Create new entity from database



## Create new entity from database



## Fix up entity definition

- Typically, the generated entity class needs some minor editing
  - An effective toString method
  - Additional accessors that return strings with subsets of data etc