Java™ Persistence API: Portability Do’s and Don’ts

Mike Keith
Oracle Corp.
http://otn.oracle.com/jpa

TS-4568

Goal

Goal

To learn more about Java^{TM} Persistence API (JPA), what the portability issues are, and what you need to know to write more portable code.

Agenda

Background and Status

The Portability Struggle

Built-in Strategies

Other Strategies

Voice of Warning (A Case Study)

Review and Summary

Agenda

Background and Status

The Portability Struggle

Built-in Strategies

Other Strategies

Voice of Warning (A Case Study)

Review and Summary

Background

- Unifying POJO persistence technology into a standard enterprise API
- Part of Enterprise JavaBeans™ (EJB™) 3.0 specification, but is separately documented
- May be used in either Java Platform, Enterprise Edition (Java EE platform) or Java Platform, Standard Edition (Java SE platform)
  - Superior ease of use within host container
  - Client API with local transactions in Java SE platform
- Service Provider Interface (SPI) for container/persistence provider pluggability

Primary Features

- POJO-based persistence model
  - Simple Java class files—not components
- Supports traditional O-O modelling concepts
  - Inheritance, polymorphism, encapsulation, etc.
- Standard abstract relational query language
- Standard O/R mapping metadata
  - Using annotations and/or XML
- Portability across providers (implementations)

Where Are We Now?

- JPA 1.0 finalized in May 2006
  - Released as part of Java EE 5 platform
- All major vendors have implemented or are working towards offering EJB 3.0 specification/JPA
- Developer interest and adoption proving to be extremely strong
- 80–90% of useful ORM features specified
  - Additional features will be added to JPA 2.0

Implementations

- Persistence provider vendors include:
  - Oracle, Sun/TopLink Essentials (RI)
  - Eclipse JPA—EclipseLink Project
  - BEA Kodo/Apache OpenJPA
  - RedHat/JBoss Hibernate
  - SAP JPA
- JPA containers
  - Sun, Oracle, SAP, BEA, JBoss, Spring 2.0
- IDEs
  - Eclipse, NetBeans™ IDE, IntelliJ, JDeveloper

Agenda

Background and Status

The Portability Struggle

Built-in Strategies

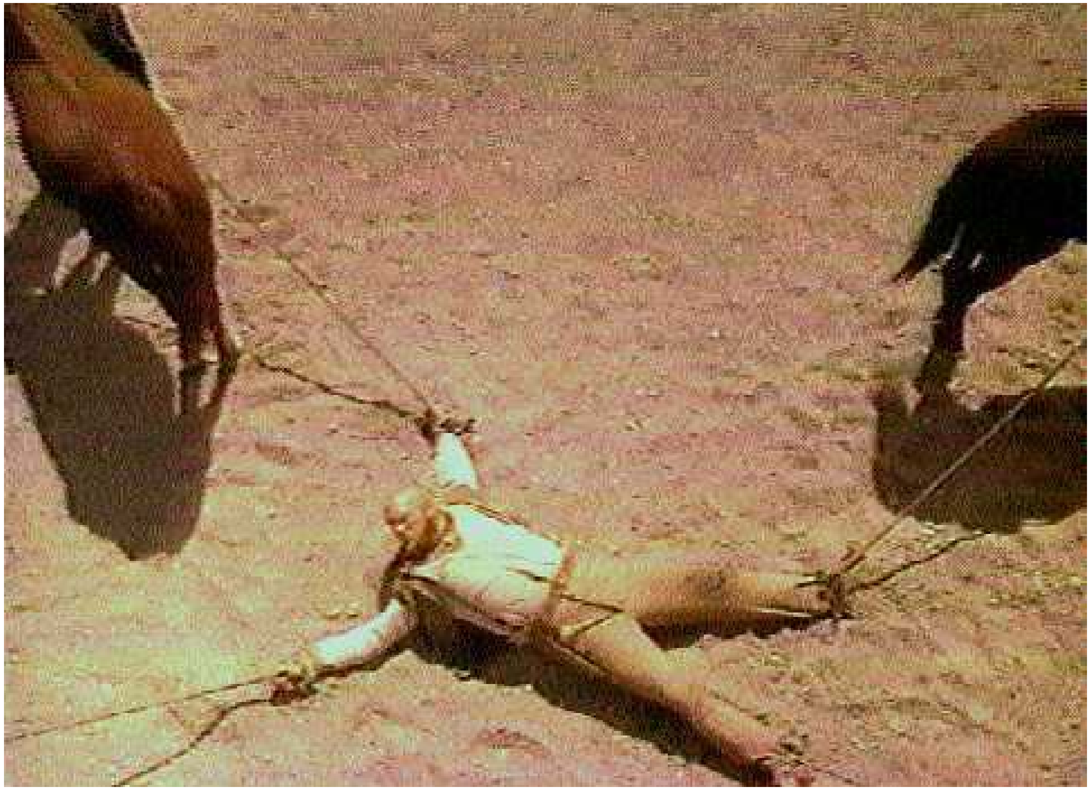
Other Strategies

Voice of Warning (A Case Study)

Review and Summary

Forces Acting Upon Us

Portability



Features

Simplicity

Portability vs. Added Value

Innovation Is Good!

- Vendors are expected to add features their customers ask for and need
  - Popular features will be moved into the JPA spec
  - Less used features shouldn't clutter the API

Corollary 1: We will always have to live with the presence of non-standard features

Corollary 2: If you are ever in the position of needing a feature that is not in the spec then you will be glad Corollary 1 is true

Accessing Vendor Features

- Vendor features show up in different forms
  - Persistence properties
  - Query hints
  - Casting to vendor-specific class
  - Customization code
  - Vendor-specific annotations
  - Additional proprietary XML descriptors

Agenda

Background and Status

The Portability Struggle

Built-in Strategies

Other Strategies

Voice of Warning (A Case Study)

Review and Summary

Integrating the Proprietary

- Hooks are built into JPA to support vendor-specific features at two different levels
  - Persistence unit properties
  - Query hints
- Unrecognized options must be ignored by the provider
- Provides source code and compile-time portability
  - Not necessarily semantically portable

Persistence Unit Properties

- Set of optional key-value properties specified in persistence.xml file
- Apply to the entire persistence unit
- May have multiple vendor properties specifying the same or different things
- Property only has meaning to the vendor that defines and interprets it

Persistence Unit Properties

```
<persistence>
  <persistence-unit name=“HR”>
    <properties>
      <property
        name=“toplink.logging.thread”
        value=“false”/>
      <property
        name=“toplink.cache.shared.default”
        value=“false”/>
    </properties>
  </persistence-unit>
</persistence>
```

Query Hints

- Vendor directives may be defined statically in named query metadata (annotations or XML)
- Applied at query execution time

```
@NamedQuery(name=“Trade.findBySymbol”,
    query=“SELECT t FROM Trade t “ +
          “WHERE t.symbol = :sym”,
    hints={
     @QueryHint(
          name=“toplink.pessimistic-lock”,
          value=“Lock”),
     @QueryHint(
          name=“openjpa.ReadLockLevel”,
          value=“write”) } )
```

Query Hints

- May be defined dynamically using the Query API
  - More flexible because any Java object may be passed in as the value
  - Lose source-code portability if object is vendor-specific

```
Query query = em.createQuery(
   “SELECT t FROM Trade t WHERE t.symbol = :sym”);
query.setHint(“toplink.pessimistic-lock”,“Lock”);
     .setHint(“openjpa.ReadLockLevel” “write”);
     .setParameter(“sym”,“ORCL”)
     .getResultList();
```

Pessimistic Transactions

- Optimistic concurrency is built into JPA, but no support for pessimistic locking is specified
- Will likely be addressed in a future JPA release
- All credible JPA implementations support pessimistic locks in some way or another
- No completely portable way to pessimistically lock, but many provide query hints like those shown in previous slides
- EntityManager lock() method can be used with optimistic locking, and error handling

Java DataBase Connectivity (JDBC™) Connection Settings

- Resource-level JDBC technology settings are vendors responsibility
- Need to specify the four basic JDBC technology properties to obtain driver connections
  - Driver class, URL, username, password
- The property keys will be different, but the values for a given JDBC technology data source will be the same for all vendors
- Used when not in a container, or when managed data sources are not available or not desired

JDBC Technology Connection Settings

```
<properties>
  ...
  <!-- TopLink -->
  <property name=“toplink.jdbc.driver”
      value=“oracle.jdbc.Driver”/>
  <property name=“toplink.jdbc.url”
     value=“jdbc:oracle:thin:@localhost:1521:XE”/>
  <property name=“toplink.jdbc.user
      value=“scott”/>
  <property name=“toplink.jdbc.password”
      value=“tiger”/>
```

JDBC Technology Connection Settings

```
...
  <!-- OpenJPA -->
  <property name="openjpa.ConnectionDriverName"
      value=“oracle.jdbc.Driver”/>
  <property name="openjpa.ConnectionURL"
     value="jdbc:oracle:thin:@localhost:1521:XE"/>
  <property name="openjpa.ConnectionUserName"
      value="scott"/>
  <property name="openjpa.ConnectionPassword"
      value=“tiger"/>
  ...
</properties>
```

DDL Generation

- Standard enables it but does not currently dictate that providers support it
- Mapping metadata specifies how DDL should be generated
- Vendors may offer differing levels of support, including:
  - Generating DDL to a file only
  - Generating and executing DDL in DB
  - Dropping existing tables before creating new ones

DDL Generation

```
<properties>
  ...
  <!-- TopLink -->
  <property
      name="toplink.ddl-generation"
      value="create-tables"/>
  <!-- OpenJPA -->
  <property
      name="openjpa.jdbc.SynchronizeMappings"
      value=“buildSchema"/>
  ...
</properties>
```

Database Platform

- No standard way to define the database platform being used at the back end
- If provider knows the database then it can:
  - Generate corresponding SQL
  - Make use of db-specific features and types
  - Make adjustments for db-specific constraints and limitations
- Implementations usually automatically discover database platform

Database Platform

```
<properties>
  ...
  <!-- TopLink -->
  <property
      name="toplink.target-database"
      value="Derby"/>
  <!-- OpenJPA -->
  <property
      name="openjpa.jdbc.DBDictionary"
      value="derby"/>
  ...
</properties>
```

Logging

- Users want to control over logging, but vendors use different logging APIs
- Can usually configure to use one of the well-known logging APIs
  - java.util.logging, log4J, etc.
- Common requirement is to configure the logging level to show the generated SQL

Logging

```
<properties>
  ...
  <!-- TopLink -->
  <property
      name="toplink.logging.level"
      value="FINE"/>
  <!-- OpenJPA -->
  <property
      name="openjpa.Log"
      value=“Query=TRACE, SQL=TRACE"/>
  ...
</properties>
```

Agenda

Background and Status

The Portability Struggle

Built-in Strategies

Other Strategies

Voice of Warning (A Case Study)

Review and Summary

Casting to Implementation Artifacts

- Cast specification-defined interface to a vendor implementation type

```
public Employee pessimisticRead1(int id) {
  Employee emp =
             em.find(Employee.class, id);
  UnitOfWork uow = (TopLinkEntityManager)
                        em.getUnitOfWork();
  uow.refreshAndLockObject(emp, LOCK);
  return emp;
}
```

Casting to Implementation Artifacts

```
public Employee pessimisticRead2(int id) {
   Query q = em.createQuery(
      “SELECT e FROM Employee e “ +
      “WHERE e.id = :e_id”);
   q.setParameter(“e_id”, id);
    ((ObjectLevelReadQuery)
      ((TopLinkQuery)q.getDatabaseQuery()))
                            .acquireLocks();
   return q.getSingleResult();
  }
```

Customization

- Customization opens the door to any amount of twiddling
- Can change or set additional vendor metadata
- Customization class has compile-time dependencies, but limits the scope of them
- Convenient place to stash vendor-specific feature code—if you change providers you know exactly where to look first
- Write “default” code, if possible, so that even if the vendor code is not present the application will still work

Customization Using Properties

```
<properties>
  ...
  <!-- TopLink -->
  <property
    name="toplink.session.customizer"
    value="acme.MySessionCustomizer"/>
  <property
    name="toplink.descriptor.customizer.Employee"
    value="acme.MyDescriptorCustomizer"/>
  ...
</properties>
```

Customization Using Properties

```
public class MySessionCustomizer
                implements SessionCustomizer {

  public void customize(Session session) {
   session.setProfiler(new PerformanceProfiler());
  }
}

public class MyDescriptorCustomizer
                implements DescriptorCustomizer {

  public void customize(ClassDescriptor desc) {
    desc.disableCacheHits();
  }
}
```

Customizing Queries

- May have lots of pre-existing queries in proprietary vendor query format
- May want to access functionality in a custom or vendor-specific query language
- Once they are added to the vendor EntityManager then they are accessible as normal JPA named queries
- Can migrate them to JPQL or port them to a different vendor when/as required

Customizing a Query

```
public class MySessionCustomizer
                  implements SessionCustomizer {

  public void customize(Session session) {
    DatabaseQuery query =
        session.getQuery(“Employee.findAll”);
    StoredProcedureCall call =
        new StoredProcedureCall();
    call.setProcedureName("Read_All_Employees");
    query.setCall(call);
  }
}
```

Customizing a Query

In entity code

```
@Entity
@NamedQuery(name="Employee.findAll",
    query="SELECT e FROM Employee e")
public class Employee {  ... }
```

In component code:

```
...
  return
  em.createNamedQuery("Employee.findAll")
           .getResultList();
...
```

Vendor Annotations

```
import javax.persistence.Entity;
import oracle.toplink.annotations.Cache;
import org.apache.openjpa.persistence.DataCache;

@Entity
@Cache(disable-hits=TRUE) // TopLink annotation
@DataCache(enabled=false) // OpenJPA annotation
public class Employee {
  ...
}
```

Agenda

Background and Status

The Portability Struggle

Built-in Strategies

Other Strategies

Voice of Warning (A Case Study)

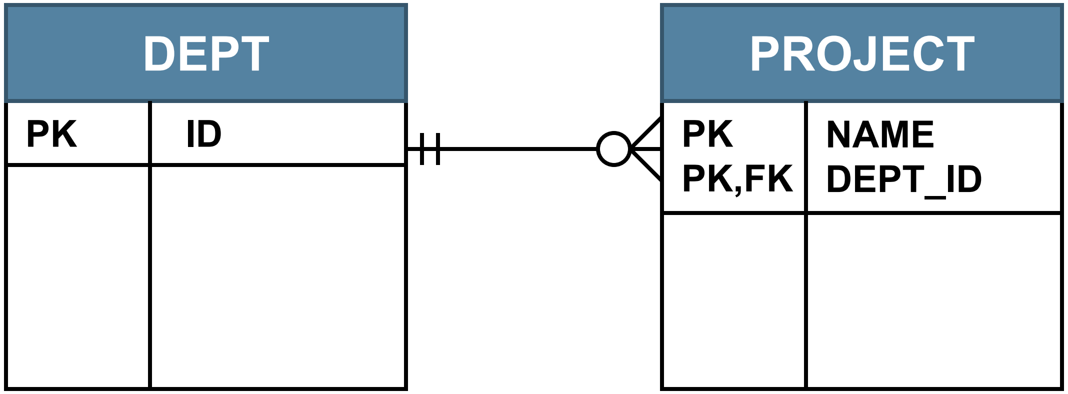
Review and Summary

PK With Relationship

- Sometimes in the data model the primary key includes one or more foreign key columns
- In the object model this means the identifier includes the identifier of a related entity
- Relationship must exist when the entity is first created
- Relationship may not change over the lifetime of the entity

PK With Relationship

- Each department may have many projects, but they must all have different names
- Many projects may have the same name, but only if they belong to different departments



DEPT	
PK	ID

PROJECT	
PK PK,FK	NAME DEPT_ID

PK With Relationship

```
/* Compound PK class */
public class ProjectId implements Serializable {
  int deptId;
  String name;

  public ProjectId() {}

  public ProjectId(int deptId, String name) {
    this.deptId = deptId;
    this.name = name;
  }
```

PK With Relationship

```
  /* PK class (cont’d) */
  public int getDeptId() { return deptId; }
  public String getName() { return name; }

  public boolean equals(Object o) {
    return ((o instanceof ProjectId) &&
      name.equals(((ProjectId)o).getName()) &&
        deptId == ((ProjectId)o).getDeptId());
  }
  public int hashCode() {
    return name.hashCode() + deptId;
  }
}
```

PK With Relationship

```
/* The Project entity class */
@Entity @IdClass(ProjectId.class)
public class Project {
   @Column(name="DEPT_ID",
              insertable="false",
              updatable="false")
  @Id private int deptId;
  @Id private String name;

  @ManyToOne @JoinColumn(name="DEPT_ID")
  private Department department;
   ...
}
```

Do we make the Id mapping (@Column) read-only or the relationship (@JoinColumn) mapping?

PK With Relationship

- Depends on:
  - The vendor
  - How you use the entity
- Some vendors support one or the other, or both
- If you set the relationship when creating a Project and persist it without filling in the dept id then you might make the dept id read-only
- If you set the dept id and then persist the Project then you might make the relationship read-only

Agenda

Background and Status

The Portability Struggle

Built-in Strategies

Other Strategies

Voice of Warning (A Case Study)

Review and Summary

Review

- Persistence properties and query hints normally offer compile-time and runtime portability
- Class casts introduce compile time and runtime dependencies
- Vendor annotations introduce compile-time dependencies
- Customization provides a “pluggable” dependency that can be easily removed
- All of these may and often will result in subtle runtime dependencies

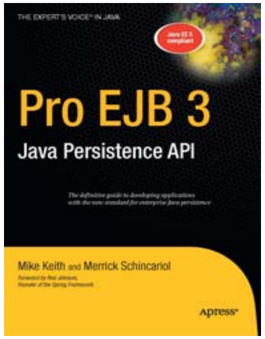
Summary

- No spec can or ever will offer everything to everyone
- JPA must (and does) provide ways for vendors to add value and support features for their users
- Vendors may also use other approaches to make features available
- Developers should be aware of non-portable features, and consequences of using them
- Spec is well-positioned to add new features as requested by the community

For More Information

- Technical Sessions
  - **TS-4902:** Java Persistence API: Best Practices & Tips Friday, 10:50AM
- Resources
  - http://otn.oracle.com/jpa
- Books
  - **Pro EJB 3: Java Persistence API**

    Mike Keith & Merrick Schincariol (Foreword by Rod Johnson)


THE EXPERT'S VOICE® IN JAVA
Pro EJB 3
Java Persistence API
Mike Keith and Merrick Schincariol
Apress®

Q&A

Java$^{\text{TM}}$ Persistence API: Portability Do’s and Don’ts

Mike Keith
Oracle Corp.
http://otn.oracle.com/jpa

TS-4568