Java™ Persistence 2.0

Linda DeMichiel

Sun Microsystems, Inc.

TS-4945

Goal of This Talk

Learn where we are heading with the Java™ Persistence API—and why.

Agenda

Background

Proposed Functionality

- What and Why (including some gotchas)

Summary and Roadmap

Where to Learn More

Java Persistence Today

Background

- Java Persistence API introduced in JSR 220 (Enterprise JavaBeans™ (EJB™) 3.0)
  - A great start!
  - Standalone use for Java Platform, Standard Edition (Java SE platform) environments
  - Pluggable implementations for Java Platform, Enterprise Edition (Java EE platform) environments
  - Strong uptake in the community
- However: Still a 1.0 release
  - Open issues, ambiguities, and a few bugs
  - Optional functionality left as vendor extensions
  - Some missing pieces

Java Persistence 2.0

- Purpose of Java Persistence 2.0 is to solidify the standard
  - Clarify open issues
  - Reduce non-portability aspects
  - Standardize optional functionality
  - Address requests from the community for some needed features
- Will be new Java Specification Request (JSR) under the Java Community Process℠ (JCP℠) program

Proposed Functionality

- More flexible modeling capabilities
- Expanded object/relational mapping functionality
- Additions to the Java Persistence query language
- API for criteria queries
- Standardization of sets of configuration hints
- Standardization of additional contracts for handling detached entities
- Expanded pluggability contracts for container-managed extended persistence contexts
- Support for validation

Agenda

Background

Proposed Functionality

- **More Flexible Modeling**

Summary and Roadmap

Where to Learn More

More Flexible Modeling

- Improved support for embeddable classes
- Collections of strings and other basic types
- Ordered lists
- More flexible use of access types

Multiple Levels of Embeddables

```
// Strawman syntax

@Embeddable public class Address {
   protected String street;
   protected String city;
   protected String state;
   @Embedded protected ZipCode zipcode;
   ...
}

@Embeddable public class ZipCode {
   @Length(5) protected String zip;
   @Length(4) protected String plusFour;
}
```

Multiple Levels of Embeddables (Cont.)

```
@Entity public class Customer {
@Id protected Integer id;
protected String name;
protected Address address;
...
}
```

CUSTOMER

ID	NAME	STREET	CITY	STATE	ZIP	PLUSFOUR

Collections of Basic Types, Embeddables, Etc.

```
// Strawman syntax

@Entity public class Person {
    @Id protected String ssn;
    protected String name;
    protected Address primaryResidence;
    @Basic protected Set<String> nickNames = new HashSet();
    ...
}

@Entity public class WealthyPerson extends Person {
    @Embedded
    protected Set<Address> vacationHomes = new HashSet();
    ...
}
```

Mapped Superclasses for Embeddables

- Mapped superclasses designed to support factorization of entity state/behavior
  - State is applied to inheriting entities
  - Don't define entities themselves
- Extension to embeddables
  - State is applied to inheriting embeddable classes
  - Don't define embeddables themselves
  - i.e., can't be value of field or property

Example

```
@MappedSuperclass public class Address {
   protected String street;
   protected String city;
   protected String state;
   @Embedded protected ZipCode zipcode;
   ...
}

@Embeddable public class BusinessAddress extends Address {
   protected String building;
   protected String mailStop;
   ...
}

@Embeddable public class HomeAddress extends Address {
   protected String apartment;
   ...
}
```

What About

```
@Embeddable public class Address {
   protected String street;
   protected String city;
   protected String state;
   @Embedded protected ZipCode zipcode;
   ...
}

@Embeddable public class BusinessAddress extends Address {
   protected String building;
   protected String mailStop;
   ...
}

@Embeddable public class HomeAddress extends Address {
   protected String apartment;
   ...
}
```

Ordered Lists

- @OrderBy metadata specifies sort order when a collection is retrieved
  - Doesn't apply to updating of collection
  - Database-centric point of view
- However: Many developers want ordering to be persistent

Example

```
// Strawman syntax

@Entity public class Employee {
    @Id protected Integer empId;
    protected String name;
    protected String ssn;
    ...
    @OneToMany
    @Ordered @OrderColumn(name="REVIEW_INDEX")
    protected List<Review> reviews = new ArrayList();
    ...
}
```

Access Type

- Defines whether provider uses fields or properties
- Spec currently states that only a single access type applies to an entity hierarchy
  - Unclear what this really means
    - Implementations may (non-portably) support more, but not defined how
- Issues
  - Current lack of portability
  - Single access type too inflexible
  - Allowing multiple access types to be defined within a single class is the interesting case

Consider

```
@Embeddable public class Address {
   protected String street;
   protected String city;
   protected String state;
   protected String zip;
   ...
   public String getStreet() {return street;}
   public void setStreet(String street) {
     this.street = street;
   }
   ...
   public String getCity() {return city;}
   public void setCity(String city) {this.city = city;}
   ...
}
```

Does This Work?

```
@Entity public class Customer {
@Id protected Integer id;
protected String name;
@Embedded protected Address address;
...
}

@Entity public class SalesRep {
   protected Integer id;
   protected String name;
   protected Address address;
   ...
   @Id public String getName() {return name;}
   ...
   @Embedded public Address getAddress() {return address;}
   ...
}
```

Example: Combining Access Types

```
@Entity public class Customer {
@Id protected Integer id;
protected String name;
protected Address address;
...
}

// Strawman syntax
@AccessType(PROPERTY)
@Embeddable public class Address {
   protected String street;
   protected String city;
   protected String state;
   protected String zipcode;
   ...
   public String getStreet() {return street;}
   public void setStreet(String street) {this.street = street;}
   ...
}
```

Example: Combining Access Types

```
@Entity public class Employee {
@Id protected Integer empId;
protected String name;
protected String ssn;
...
}


@AccessType(PROPERTY)
@Entity public class Contractor extends Employee {
   protected Float hourlyRate;
   protected String agency;
   ...
   @Basic public Float getHourlyRate() {return hourlyRate;}
   public void setHourlyRate(Float rate) {hourlyRate = rate;}

   public String getAgency() {return agency;}
   public void setAgency(String agency) {this.agency = agency;}
}
```

Example: Combining Access Types

```
//  Not so obvious


@Entity public class Customer {
@Id protected Integer id;
protected String name;
protected Address address;
...
protected Integer rating;
...
@AccessType(PROPERTY)
public Integer getCreditRating() {
   return rating;
}
public void setCreditRating(Integer rating) {
   this.rating = rating;
}
}
```

Agenda

Background

Proposed Functionality

- **Expanded O/R Mapping Functionality**

Summary and Roadmap

Where to Learn More

Expanded O/R Mapping Functionality

Relationship mappings

- Unidirectional one-to-many relationships using foreign key mappings currently not supported
- However:
  - This is the obvious database modeling strategy
  - Shouldn't have to make one-to-many relationships bidirectional to use it

Unidirectional One-to-Many

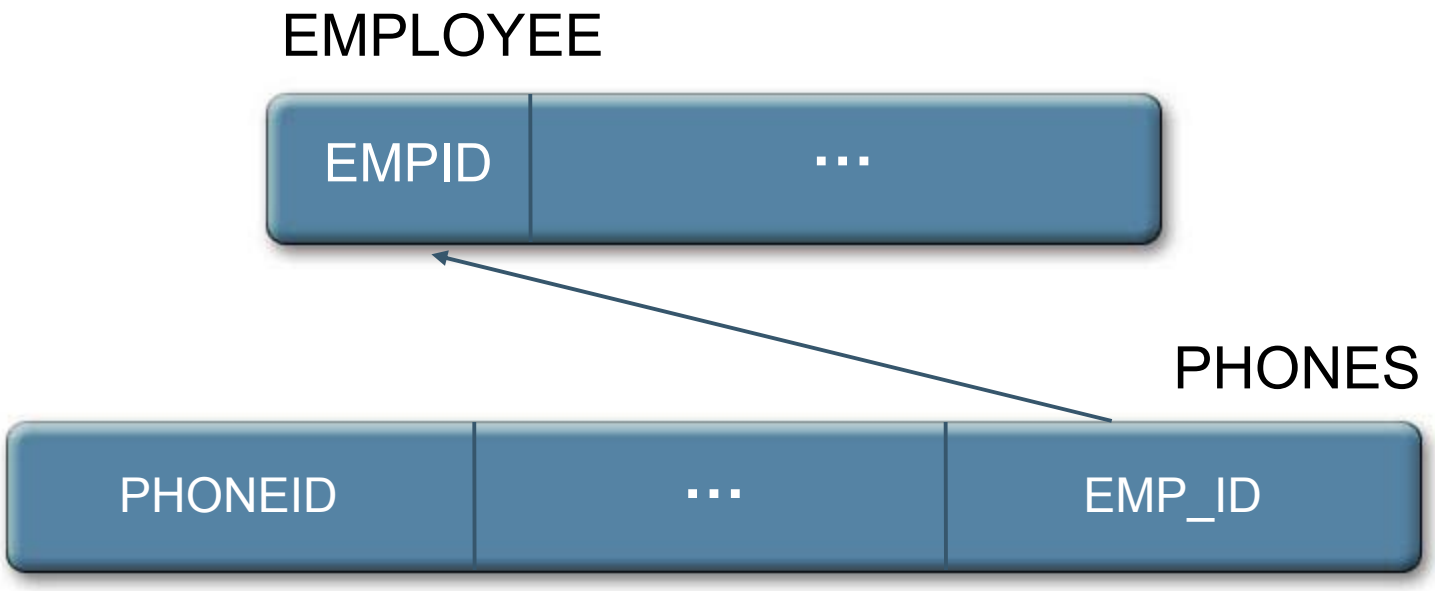
```
@Entity public class Employee {
    @Id protected Integer empId;
    ...
    @OneToMany
    protected Set<Phones> phones = new HashSet();
    ...
}

@Entity public class Phone {
  @Id protected int phoneId
  protected Float currentCharges;
  protected String vendor;
  ...
  // Don't want phone to have to know about the employee !
}
```

Using Join Table

```
// Default implementation
```

Using Foreign Key Mapping



Unidirectional One-to-Many

```
// Using foreign key mapping

@Entity public class Employee
    @Id protected Integer empId;
    ...
    @OneToMany @JoinColumn(name="EMP_ID")
    protected Set<Phones> phones = new HashSet();
    ...
}

@Entity public class Phone {
   @Id protected int phoneId
   protected Float currentCharges;
   protected String vendor;
   ...
}
```

Expanded O/R Mapping Functionality

Inheritance mappings

- Two currently supported mapping strategies
  - Single table per class hierarchy
    - Non-normalized (nulls!)
    - Good support for polymorphic queries, relationships
  - Joined subclass strategy
    - Subclass-specific state stored in separate table(s)
    - Normalized
    - Performance an issue for moderately deep hierarchies
- Table per concrete class strategy left as optional
  - However: legacy databases do model this way

Example

```
@Inheritance(TABLE_PER_CLASS)
@Entity public class Employee {
  @Id protected Integer empId;
  protected String name;
  protected String ssn;
  ...
}

@Entity public class RegularEmployee extends Employee {
  protected Float salary;
  @Column(name="VAC_HRS") protected Integer vacation;
}

@Entity public class Contractor extends Employee {
  @Column(name="HR_RATE") protected Float hourlyRate;
  protected String agency;
}
```

Using Table Per Class

EMPLOYEE

EMPID	NAME	SSN

REGULAR

EMPID	NAME	SSN	SALARY	VAC_HRS
25	Joe	123-45-6789	98000.00	48
58	Max	234-56-7891	56000.00	80
19	Bill	567-89-1234	125000.00	92

CONTRACTOR

EMPID	NAME	SSN	HR_RATE	AGENCY
97	Ann	345-67-8912	100.00	XYZ
82	Rob	456-78-9123	80.00	ITemps

Table Per Class Strategy

- Each concrete class mapped to separate table
- Pluses
  - Normalized
  - Good for non-polymorphic queries
  - OK for non-polymorphic relationships
- Minuses
  - Poor for polymorphic queries
  - Very poor for polymorphic relationships

Agenda

Background

Proposed Functionality

- **Expanded Query Capabilities**

Summary and Roadmap

Where to Learn More

Java™ Persistence Query Language

Some current limitations

- SELECT clause still too constrained
  - Only aggregate functions supported in SELECT clause
  - Use of additional operators and functions important, especially for report queries
- Some unnecessary restrictions on parameter usage
- Queries are always polymorphic

Examples: SELECT Clause

```
SELECT CONCAT(p.lastname, CONCAT(', ', p.firstname)) AS n
FROM Person p
ORDER BY n
```

```
SELECT e.name, e.salary + e.bonus
FROM Employee e
WHERE e.dept.name = 'Engineering'
```

```
SELECT d.name, SUM(c.hourlyRate * c.hoursWorked * 52)
FROM Contractor c JOIN c.dept d
GROUP BY d.name
```

Example: Restricted Polymorphism

```
//Strawman syntax

SELECT e.name
FROM Employee e JOIN e.dept d
WHERE d.name = 'Engineering'
AND CLASS(e) IN ('Contractor', 'PartTime')
```

Dynamic Queries

Java Persistence dynamic queries currently entail string construction

```
    ...
    @PersistenceContext EntityManager em;
    ...
Query q = em.createQuery(
  "SELECT c" +
  "FROM Customer c" +
  "WHERE c.status = 'preferred'" +
  "AND c.address.city = 'New York'" +
  "ORDER BY c.name"
  );
    ...
```

Criteria Queries

Criteria APIs allow “node-wise” query construction

```
   ...
   @PersistenceContext EntityManager em;
   ...

   // Strawman syntax
CriteriaQuery cq = em.createCriteria(Customer.class)
  .add(Restrictions.eq(“status”, “preferred”))
  .add(Restrictions.eq(“address.city”, “New York”))
  .addOrder(Order.asc(“name”));
...
```

Criteria Queries

- Considerable set of criteria APIs and expression APIs already in existence for us to learn from
  - Hibernate
  - OJB
  - Cayenne
  - TopLink
  - …

Agenda

Background

Proposed Functionality

- **Configuration Hints**

Summary and Roadmap

Where to Learn More

Standardized Hints and Properties

- Hints and properties used in configuration of:
  - Entity manager factory
  - Entity manager/persistence context
  - Queries
- Many candidates for standardization
  - JDBC driver, user, password, connection pool
  - Caching, cache size
  - Timeouts
  - Logging
  - DDL handling
  - Etc., etc.

JDBC driver = a driver supporting the JDBC™ API (JDBC driver)

Agenda

Background

Proposed Functionality

Better Contracts for Handling Detached Objects

Summary and Roadmap

Where to Learn More

Unfetched State

- Detached entities often have unfetched state and/or relationships
  - What is fetched is determined by fetch elements, defaults, queries, and access
  - Should consider fetch plans for greater flexibility
- Unfetched state access issue left as part of application contract
  - e.g., “don’t access x, y, z”
- What happens on access to unfetched state is currently undefined
  - Implementations place different burdens on clients

Example: Detached Access

```
@NamedQuery(
    name="findBySSN",
    query="SELECT e FROM Employee e WHERE e.ssn = :ssn"
)
@Entity public class Employee {
    @Id protected Integer empId;
    protected String name;
    protected String ssn;
    ...
    @ManyToOne(fetch=LAZY)
    protected Department dept;
    ...
}
```

Example: Detached Access (Cont.)

```
@Stateless @Remote
public class HRInfoBean implements HRInfoService {
   ...
   @PersistenceContext EntityManager em;

   public Employee findEmployeeBySSN(String ssn) {
        return em.createNamedQuery("findBySSN")
          .setParameter("ssn", ssn)
          .getSingleResult();
      }
    ...
}
```

Example: Detached Access (Cont.)

```
// In client

   @EJB HRInfoService HRInfo;
   ...
   Employee e = HRInfo.findEmployeeBySSN("123-45-6789");
   ...
   Department d = e.getDepartment();
```

Questions:

What do you have to do to deploy this client?

What happens when you access the unfetched department?

Extended Persistence Contexts

- Application-managed persistence contexts are always extended
  - Application manages their lifecycle
    - Persistence context exists until closed
  - Application manages transaction association
    - Requirements for joinTransaction() a source of bugs

Example: (buggy)

```
public class BookBuyerServlet extends HttpServlet {
  @PersistenceUnit EntityManagerFactory emf;
  @Resource UserTransaction utx;
  protected void doPost(HttpServletRequest req,
                        HttpServletResponse res) throws ... {
    Integer custId = Integer.parseInt(req.getParameter("customerId"));
    String bookName = req.getParameter("bookName");
    EntityManager em = emf.createEntityManager();
    utx.begin();
    Customer c = em.find(Customer.class, custId);
    Book b = em.find(Book.class, bookName);
    Order o = new Order(b);
    c.addOrder(o);
    em.persist(o);
    utx.commit();
    em.close();
  }
}
```

Example: (fixed)

```
public class BookBuyerServlet extends HttpServlet {
  @PersistenceUnit EntityManagerFactory emf;
  @Resource UserTransaction utx;
  protected void doPost(HttpServletRequest req,
                        HttpServletResponse res) throws ... {
    Integer custId = Integer.parseInt(req.getParameter("customerId"));
    String bookName = req.getParameter("bookName");
    utx.begin();
    EntityManager em = emf.createEntityManager();
    Customer c = em.find(Customer.class, custId);
    Book b = em.find(Book.class, bookName);
    Order o = new Order(b);
    c.addOrder(o);
    em.persist(o);
    utx.commit();
    em.close();
  }
}
```

Example: (fixed)

```
public class BookBuyerServlet extends HttpServlet {
  @PersistenceUnit EntityManagerFactory emf;
  @Resource UserTransaction utx;
  protected void doPost(HttpServletRequest req,
                        HttpServletResponse res) throws ... {
    Integer custId = Integer.parseInt(req.getParameter("customerId"));
    String bookName = req.getParameter("bookName");
    EntityManager em = emf.createEntityManager();
    utx.begin();
    em.joinTransaction();
    Customer c = em.find(Customer.class, custId);
    Book b = em.find(Book.class, bookName);
    Order o = new Order(b);
    c.addOrder(o);
    em.persist(o);
    utx.commit();
    em.close(); }
}
```

Container-Managed Extended Persistence Contexts

- Provide ease-of-use in Java EE application environments
  - Stateful session bean is perfect fit for management
  - Automatic coupling of lifecycles
- Becoming increasingly important to support “conversations”
  - Stateful web services (EJB 3.1 specification)
  - First-class conversational scopes (Web Beans)
- Issue: stateful session bean “passivation”
  - Needed for scaling, failover/replication
- Spec needs to further define pluggability contracts

Agenda

Background

Proposed Functionality

- **Validation**

Summary and Roadmap

Where to Learn More

Validation

- JSR 303 (Bean Validation)
  - Goal is to define metadata model and API for validation
  - For general use in Java SE and Java EE platforms
- Would like to leverage this for Java Persistence
  - Whether this is possible depends on rate of progress of JSR 303

Validation Example

```
// Strawman syntax

@Entity public class Employee {
    @Id @GeneratedValue protected Integer empId;

    @Required protected String name;

    @Length(max=5) protected String locationCode;

    @Max(240) protected Integer vacationAccrued;

    @AdequatelyCompensated protected Float salary;

    ...
}
```

Agenda

Background

Proposed Functionality

Summary and Roadmap

Where to Learn More

Summary

Java Persistence 2.0

- Proposed functionality to support
  - More flexible modeling
  - Expanded O/R mapping functionality
  - Query language extension
  - Greater portability across implementations
  - Alignment with emerging JSRs

Roadmap

- Java Persistence 2.0 JSR to be posted shortly
- Expert Group formation in June
- Goal is completion in Java EE platform v.6 time-frame
  - Desirable to complete Maintenance Release (1.1) as first phase
- Input alias
  - persistenceNoSpam-feature-requests@sun.com
    - Will go to Expert Group (once formed)
    - Will be reincarnated when too much spam

For More Information

Related Sessions and BOFs

- TS-4856: Architecture of Popular Object/Relational Mapping Providers (Today)
- TS-4568: Java Persistence API: Portability Do's and Don'ts (Thursday)
- TS-4902: Java Persistence API: Best Practices and Tips (Friday)
- TS-4112: EJB 3.0 and JSR 303 Beans Validation (Friday)
- BOFs 4641, 4612: Java EE 6 Meet the Experts (Tonight)

Q&A

Java™ Persistence 2.0

Linda DeMichiel

Sun Microsystems, Inc.

TS-4945