Java is a trademark of Sun Microsystems, Inc.

JavaOne℠

Java™ Persistence 2.0 What's New?

Linda DeMichiel
Sun Microsystems

Java Persistence 2.0
Current Status

- Launched in Fall 2007 as JSR 317
- Proposed Final Draft published in March 2009
- Reference Implementation under completion through EclipseLink project
-

Java Persistence 2.0 Features

- More flexible modeling capabilities
- Expanded O/R mapping functionality
- Additions to Java Persistence query language
- Metamodel API
- Criteria API
- Pessimistic locking
- Standardization of many configuration options
- Support for validation

More flexible modeling and mapping

- Collections of basic types and embeddables
  - @ElementCollection, @CollectionTable
- Multiple levels of embeddable classes
- Embeddable classes with relationships
- Real (generalized) map support
  - @MapKeyClass, @MapKeyColumn, @MapKeyJoinColumn, ...
- Persistently ordered lists
  - @OrderColumn

More flexible modeling and mapping (cont)

- Orphan deletion
  - orphanRemoval element (of @OneToOne, @OneToMany)
- Combinations of access types
  - @Access
- Derived identities
  - @MappedById
- Unidirectional one-many foreign key mapping
- One-one, many-one/one-many join table mappings

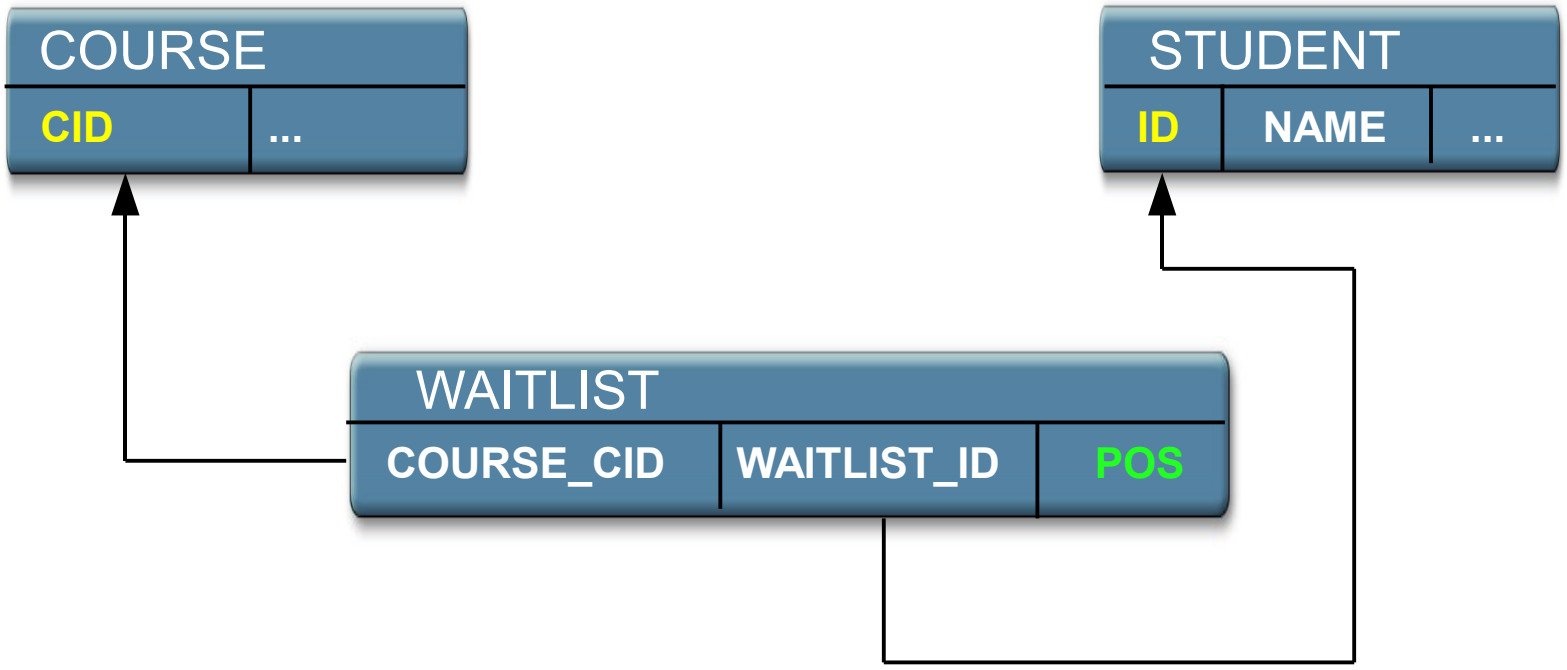
Example
Ordered lists (@OrderBy)

```
@Entity public class Course {
  @Id Integer cId;
  String name;

  // ordered upon retrieval (Java Persistence 1.0)
  @OrderBy("name")
  @ManyToMany List<Student> enrolled;
  ...
}

@Entity public class Student {
  @Id Integer id;
  String name;
  ...
}
```

Example
Ordered lists (@OrderColumn)

```
@Entity public class Course {
  @Id Integer cId;
  String name;

  // ordered upon retrieval
  @OrderBy("name")
  @ManyToMany List<Student> enrolled;

  // ordering stored in database
  @OrderColumn(name="pos")
  @JoinTable(name="waitlist")
  @ManyToMany List<Student> waitlist;
  ...
}

@Entity public class Student {...}
```

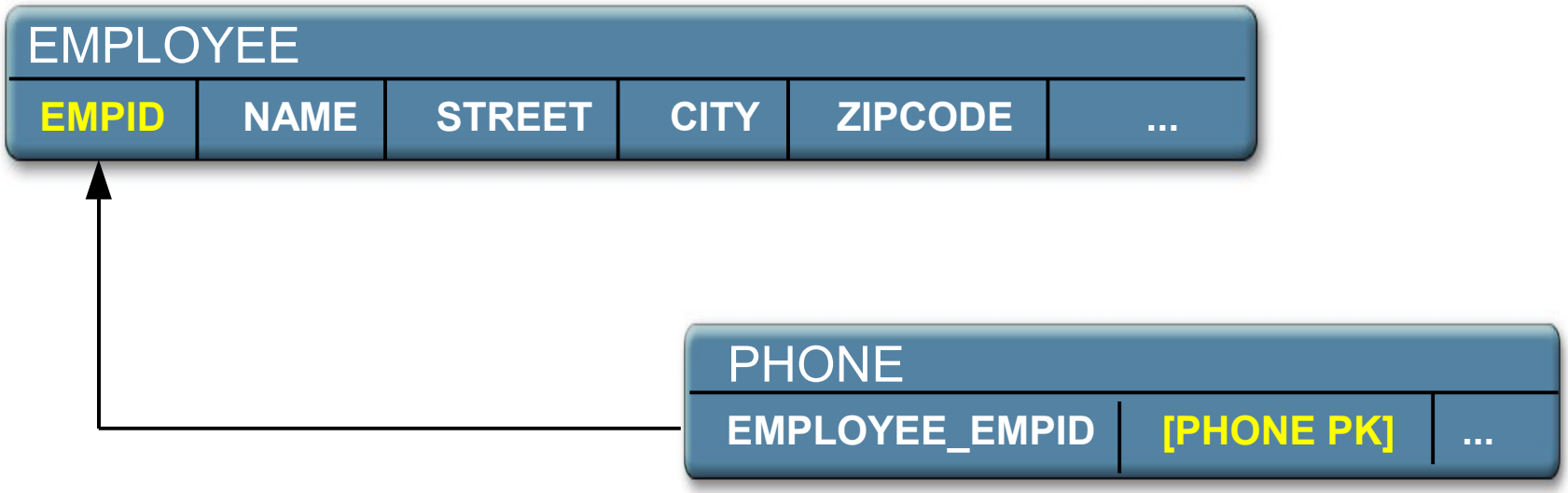
Example
Ordered lists (@OrderColumn)



Example
Extensions for embeddables

```
>
> @Entity public class Employee {
>   @Id Integer empId;
>   String name;
>   ContactInfo info;
>   ...
> }
>
> @Embeddable public class ContactInfo {
>   Address address;
>   @OneToMany @JoinColumn Set<Phone> phones;
> }
>
> @Embeddable public class Address {
>   String street;
>   String city;
>   String zipcode;
> }
>
>
```

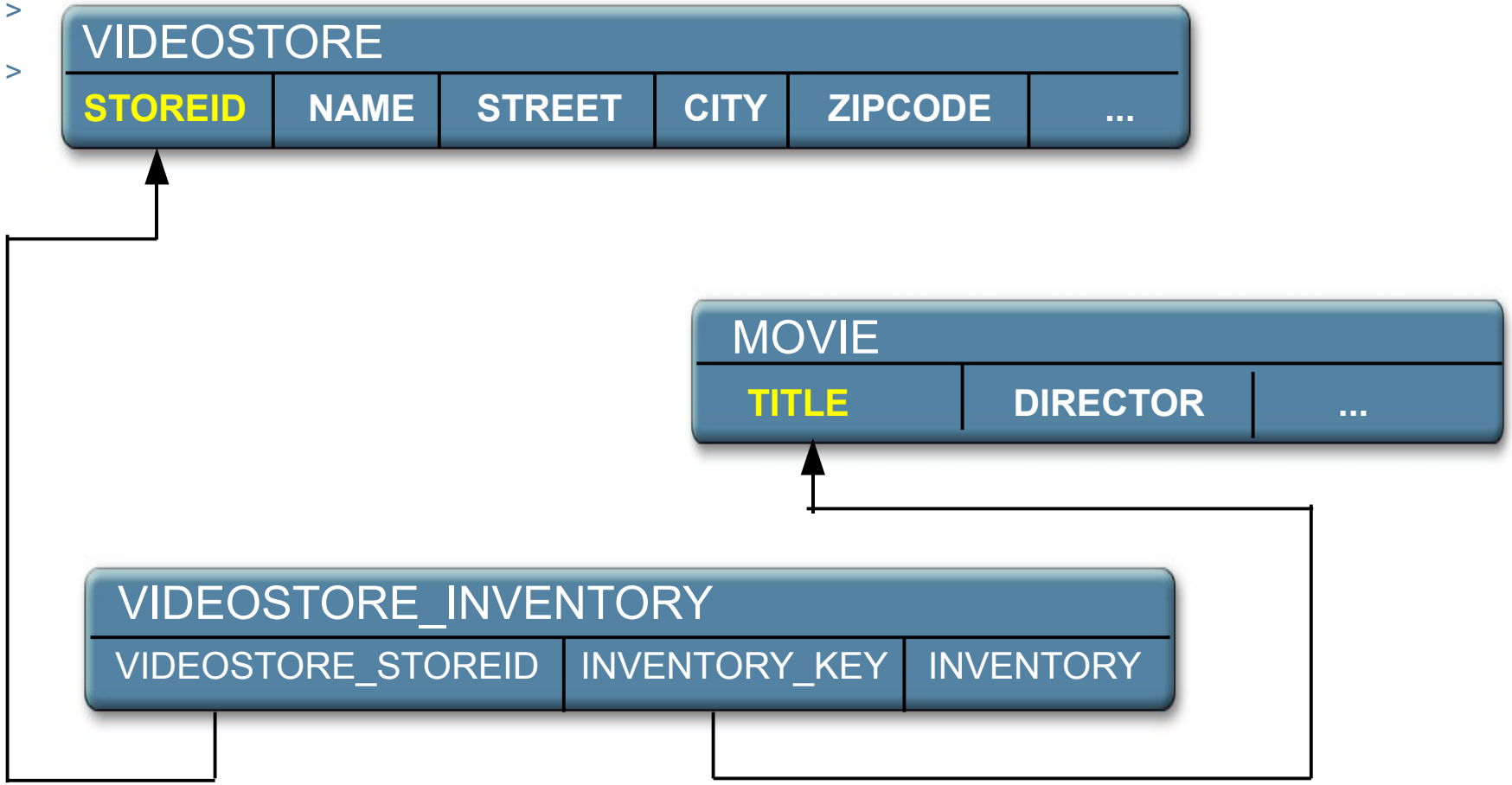
Example
Embeddables



Example
Maps

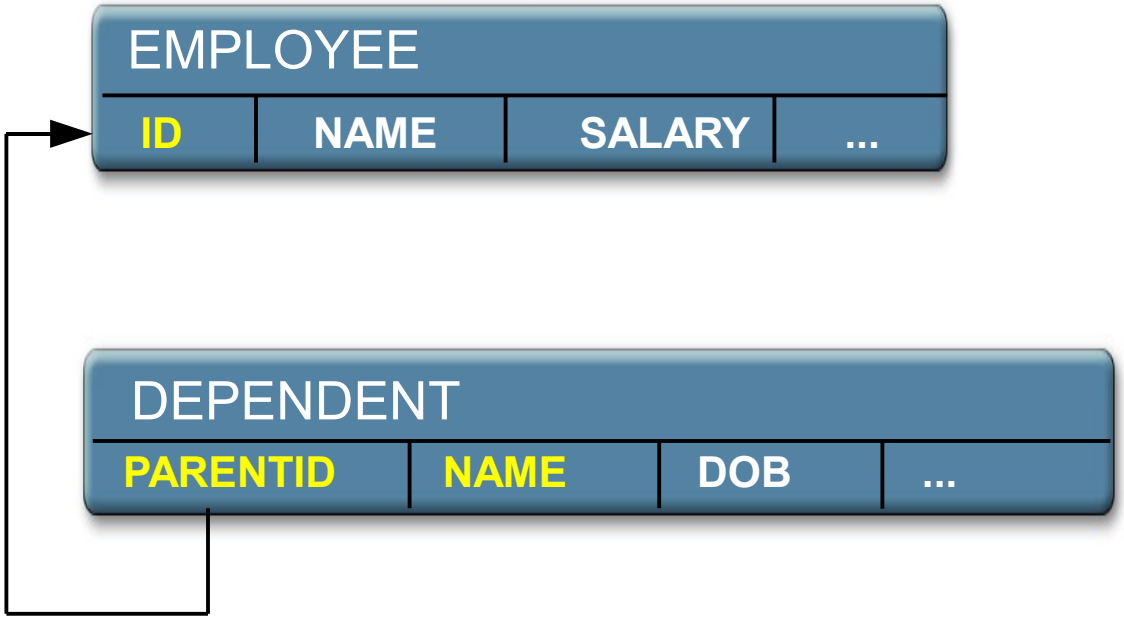
```
@Entity public class VideoStore {
  @Id Integer storeId;
  Address location;
  @ElementCollection Map<Movie, Integer>
  inventory;
  ...
}

@Entity public class Movie {
  @Id String title;
  String director;
  @OneToMany Set<Actor> stars;
  ...
}
```

Example
Maps



Example (Use Case)
Derived identities

>
>
>



Example
Derived identities

```
@Entity public class Employee {
  @Id Integer empId;
  String name;
  Float salary;
  @OneToMany(mappedBy="parent") Set<Dependent>
  dependents;
  ...
}
@Entity public class Dependent {
  @EmbeddedId DependentId id;
  @Temporal(DATE) Date dob;
  @ManyToOne @MappedById("parentId") Employee parent;
  ...
}
@Embeddable public class DependentId {
  Integer parentId;
  String name;
}
```

Java™ Persistence Query Language

- Support for use of all new modeling and mapping features
- Operators and functions in select list
- Case, coalesce, nullif expressions
- Restricted polymorphism
- Collection-valued parameters
-

Java™ Persistence Query Language

New operators

- INDEX
  - for ordered lists
- KEY, VALUE, ENTRY
  - for maps
- CASE, COALESCE, NULLIF
  - for case expressions and friends
- TYPE
  - for entity type expressions

Example
Ordered lists

```
@Entity public class Course {
  @Id Integer cId;
  String name;
  @OrderBy("name) @ManyToMany List<Student> enrolled;

  @OrderColumn
  @ManyToMany List<Student> waitlist;
  ...
}

@Entity public class Student {
  @Id Integer id;
  String name;
  ...
}
```

Example
Ordered lists

```
>
> SELECT w.name
> FROM Course c JOIN c.waitlist w
> WHERE c.name = 'Calculus'
>   AND INDEX(w) < 5
```

Example
Maps

```
@Entity public class VideoStore {
  @Id Integer storeId;
  Address location;
  @ElementCollection Map<Movie, Integer>
  inventory;
  ...
}

@Entity public class Movie {
  @Id String title;
  String director;
  @OneToMany Set<Actor> stars;
  ...
}
```

Example
Maps

```
> SELECT v.location.street, KEY(i).title, VALUE(i)
> FROM VideoStore v JOIN v.inventory i
> WHERE v.location.zipcode = '94301'
>   AND KEY(i).director LIKE '%Hitchcock%'
>   AND VALUE(i) > 0
>
>
>
>
```

Example
Case expressions

```
@Entity public class Employee {
  @Id Integer empId;
  String name;
  Float salary;
  Integer rating;
  ...
}

UPDATE Employee e
SET e.salary =
          CASE WHEN e.rating = 1 THEN e.salary *
  1.05
               WHEN e.rating = 2 THEN e.salary *
  1.02
               ELSE e.salary * .95
          END
```

Example
Case expressions

```
> SELECT e.name, CASE e.rating
>                     WHEN 1 THEN e.salary * 1.05
>                     WHEN 2 THEN e.salary * 1.02
>                     ELSE e.salary * .95
>                     END
> FROM Employee e
>
```

Example
Restricted polymorphism

```
> SELECT e
> FROM Employee e
> WHERE TYPE(e) IN (PartTime, Contractor)
>
> SELECT e
> FROM Employee
> WHERE TYPE(e) IN :empTypes
```

Criteria API

- Object-based API for building queries
- Designed to mirror JPQL semantics
- Strongly typed
  - Based on type-safe metamodel of persistence unit
- Supports string-based navigation as well
-
-
  -
  -

Metamodel

- Abstract, “schema-level” model over managed classes of persistence unit
  - Entities, mapped superclasses, embeddables
  -
- Accessed dynamically
  - EntityManagerFactory.getMetamodel()
  - EntityManager.getMetamodel()
  -
- And/or statically materialized as metamodel classes
  - Use annotation processor with javac

Example
Entity class

```
>
> @Entity
> public class Customer {
>    @Id Integer custId;
>    String name;
>    ...
>    Address address;
>    @ManyToOne SalesRep rep;
>    @OneToMany Set<Order> orders;
> }
>
>
>
```

Example
Metamodel class

```
>
> import javax.persistence.metamodel.*;
>
> @StaticMetamodel(Customer.class)
> public class Customer_ {
>   public static SingularAttribute<Customer, Integer>
    custId;
>   public static SingularAttribute<Customer, String> name;
>   public static SingularAttribute<Customer, Address>
    address;
>   public static SingularAttribute<Customer, SalesRep> rep;
>   public static SetAttribute<Customer, Order> orders;
> }
>
>
>
```

Criteria API
Core interfaces

- QueryBuilder
  - Factory for CriteriaQuery objects, predicates, expressions
    - Obtained from EntityManager or EntityManagerFactory
- CriteriaQuery
  - Used to add / replace / browse query elements
    - from, select, where, orderBy, groupBy, having, ... methods

Criteria API
Interfaces (cont.)

- Root
  - query roots
- Join, ListJoin, MapJoin, ...
  - joins from a root or existing join
- Path
  - navigation from a root, join, or path
- Subquery
  - subqueries
-

Example
How to Build a Criteria Query

```
>
> EntityManager em = ...;
> QueryBuilder qb = em.getQueryBuilder();
> CriteriaQuery cq = qb.create();
>
> Root<SomeEntity> e = cq.from(SomeEntity.class);
> Join<SomeEntity,RelatedEntity> j = e.join(...);
      •
> cq.select(...)
>   .where(...)
>   .orderBy(...)
>   .groupBy(...);
>
> Query q = em.createQuery(cq);
>
      •
```

Example
Navigation: Joins

```
>
> SELECT c
> FROM Customer c JOIN c.orders o
>
>
> QueryBuilder qb = ...;
> CriteriaQuery cq = qb.create();
> Root<Customer> c = cq.from(Customer.class);
> Join<Customer, Order> o =
    c.join(Customer_.orders);
> cq.select(c);
>
```

Example
Paths and Predicates

```
>
> SELECT c
> FROM Customer c JOIN c.orders o
> WHERE c.name = 'Fred' AND o.quantity > 100
>
>
> QueryBuilder qb = ...;
> CriteriaQuery cq = qb.create();
> Root<Customer> c = cq.from(Customer.class);
> Join<Customer, Order> o =
    c.join(Customer_.orders);
> cq.where(qb.equal(c.get(Customer_.name), "Fred"),
>          qb.gt(o.get(Order_.quantity), 100))
>   .select(c);
>
```

Example
Navigation chaining

```
>
> SELECT c.name
> FROM Customer c JOIN c.orders o
> WHERE o.product.productType = 'printer'
>
> QueryBuilder qb = ...;
> CriteriaQuery cq = qb.create();
> Root<Customer> c = cq.from(Customer.class);
> Join<Customer, Order> o = c.join(Customer_.orders);
> cq.where(
>   qb.equal(o.get(Order_.product)
>              .get(Product_.productType),
>            'printer')
>      )
>   .select(c.get(Customer_.name));
```

Example
Paths

```
>
> SELECT c.name
> FROM Customer c JOIN c.orders o
> WHERE o.product.productType = 'printer'
>
>
> CriteriaQuery cq = qb.create();
> Root<Customer> c = cq.from(Customer.class);
> Join<Customer, Order> o =
    c.join(Customer_.orders);
> Path<Order, Product> product =
    o.get(Order_.product);
> Path<Product, String> productType =
>     product.get(Product_.productType);
> cq.where(qb.equal(productType, 'printer'))
>    .select(c.get(Customer_.name));
```

Example
Subqueries

```
SELECT e FROM Employee e
WHERE e.salary > ALL (
    SELECT m.salary FROM Manager m
    WHERE e.dept = m.dept)

QueryBuilder qb = ...;
CriteriaQuery cq = qb.create();
Root<Employee> e = cq.from(Employee.class);
Subquery<Float> subq = cq.subquery(Float.class);
Root<Manager> m = subq.from(Manager.class);
subq.select(m.get(Manager_.salary))
    .where(qb.equal(m.get(Employee_.dept),
                    e.get(Manager_.dept)));
cq.select(e)
  .where(qb.gt(e.get(Employee_.salary),
               qb.all(subq)));
```

Example

Maps

```
// photos is map from name to image

SELECT item.name, photo
FROM Item item JOIN item.photos photo
WHERE KEY(photo) LIKE '%egret%'


CriteriaQuery cq = qb.create();
Root<Item> item = cq.from(Item.class);
MapJoin<Item, String, Object> photo =
    item.join(Item_.photos);
cq.select(item.get(Item_.name), photo)
  .where(qb.like(photo.key(), "%egret%"));
```

Example
Using Strings for Navigation

```
>
> SELECT item.name, photo
> FROM Item item JOIN item.photos photo
> WHERE KEY(photo) LIKE '%egret%'
>
> CriteriaQuery cq = qb.create();
> Root<Item> item = cq.from(Item.class);
> MapJoin<Item, String, Object> photo =
      • item.join("photos");
> cq.select(item.get("name"), photo)
>   .where(qb.like(photo.key(), "%egret%"));
```

Optimistic Concurrency
Assumptions

- Database at read-committed isolation
  - Short-term read locks
  - Long-term write locks
- Typically deferred writes to database
-
- Optimistic locking (@Version)
  - Verify version attributes for updated entities before transaction commit
-

Optimistic Locking (Lock Modes)

- Layered on top of @Version use
-
- Lock Modes
  - OPTIMISTIC (READ)
  - OPTIMISTIC_FORCE_INCREMENT (WRITE)
  -
- “READ” locks => verify versions for clean data
- “WRITE” locks => update versions for clean data
-

Pessimistic Locking

- Grab database locks upfront
  - for data being updated
  - for data being read
- Spec defines semantics, not mechanisms
-
- Lock Modes:
  - PESSIMISTIC_READ
  - PESSIMISTIC_WRITE
  - PESSIMISTIC_FORCE_INCREMENT

Pessimistic Locking
What gets locked

- By default:
  - Persistent state of entity (except element collections)
  - Relationships where entity holds foreign key
- Only if specified (with javax.persistence.lock.scope):
  - Element collections
  - Relationships owned by the entity
    - in join tables
    - with foreign key in other table
- Phantoms are possible

Locking APIs

- EntityManager methods: lock, find, refresh
- Query methods: setLockMode
- NamedQuery annotation: lockMode element
-
- javax.persistence.lock.scope property
- javax.persistence.lock.timeout hint
-
- Effect of locking failures:
  - PessimisticLockException
  - LockTimeoutException
-

Second Level Cache Control

- APIs and control options added for portability
  - Cache interface
    - evict, evictAll, contains
  - @Cacheable + shared-cache-mode element (persistence.xml)
    - ALL, NONE, ENABLE_SELECTIVE, DISABLE_SELECTIVE
  - Properties for find, refresh, setProperty methods
    - CacheRetrieveMode property
      - USE, BYPASS
    - CacheStoreMode property
      - USE, BYPASS, REFRESH
  -
  -

Validation

- Leverages work of Bean Validation JSR (JSR 303)
- Automatic validation integrated with lifecycle events
  - PrePersist
  - PreUpdate
  - PreRemove
- persistence.xml validation-mode element
  - AUTO
  - CALLBACK
  - NONE

Example
Validation

```
@Entity public class Employee {
  @Id Integer empId;
  String name;
  Float salary;
  @Max(15) Integer vacationDays;
  @Valid Address worksite;
  ...
}

@Embeddable public class Address {
    • @Size(max=30) String street;
    • @Size(max=20) String city;
    • @Zipcode String zipcode;
}
```

Summary
A lot accomplished

- More flexible modeling capabilities
- Expanded O/R mapping functionality
- Additions to JPQL
- Metamodel API
- Criteria API
- Pessimistic locking
- Standardization of many configuration options
- Support for validation
- Improved portability

Status

- Proposed Final Draft available
  - http://jcp.org/en/jsr/detail?id=317
- Final release planned for late summer with Java EE 6
- Reference implementation: EclipseLink
  - http://www.eclipse.org/eclipselink/
- Available with GlassFish
  - http://glassfish.org
  - 

eclipselink

What's Next?
Come tell us!

- Java Persistence BOF: Meet the Expert Group
  - This room; Thursday @ 6:30
- Mailing list:
  - jsr-317-pfd-feedback@sun.com

Related Sessions at JavaOne

- TS-5184 Bean Validation, Thursday @ 1:30
- TS-3977 Keeping a Relational Perspective for Optimizing JPA, Thursday @ 4:10
- BOF-5215 Java Persistence BOF, Meet the Experts, Thursday @ 6:30
- TS-5265 JPA Mapping Magical Mystery Tour, Friday @ 10:50

Thank You

Linda DeMichiel
Sun Microsystems