Java Persistence API

Patrick Linskey
EJB Team Lead
~~BEA Systems~~ Oracle
plinskey@bea.com

bea
Think liquid.™


Patrick Linskey
EJB Team Lead at BEA
JPA 1, 2 EG Member

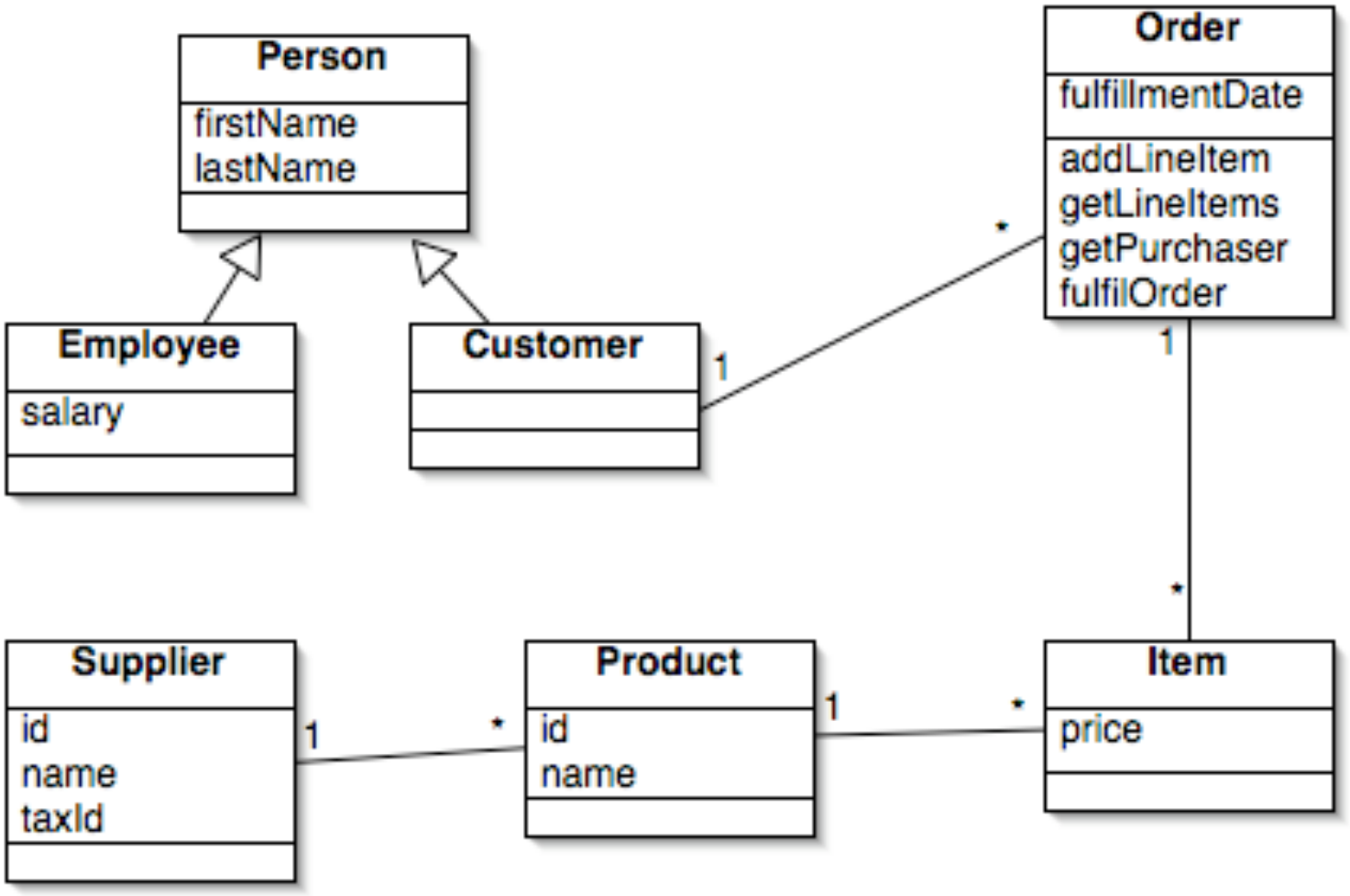
Agenda

- JPA Basics
- What's New
  - ORM
  - Configuration
  - APIs
  - Queries

The Java Persistence API

- Object / Relational Mapping for Java
- Direct support for POJOs
- Powerful query language
  - OLTP-centric
- Standardized ORM metadata
- Direct support for Data Transfer Objects
- Designed for testability
- Java EE and Java SE

Example Object Model



JPA Usage Example – a Session Bean

```
@Stateless
@Remote
public OrderManagerImpl
  implements OrderManager {

  @PersistenceContext private EntityManager em;

  public Order newOrderForProduct(int custId, int prodId)
  {
    Customer c = em.find(Customer.class, custId);
    Product p = em.find(Product.class, prodId);

    Order o = new Order(c);
    em.persist(o);
    o.addLineItem(new Item(p));

    return o;
  }
}
```

JPQL Syntax Examples

```
// customers 20-30 named 'Joe', ordered by last name
Query q = em.createQuery("select c from Customer c where
    c.firstName = :fname order by c.lastName");

q.setParameter("fname", "Joe");
q.setFirstResult(20);
q.setMaxResults(10);
List<Customer> customers = (List<Customer>)
    q.getResultList();


// all orders, as a named query
@Entity
@NamedQuery(name="Order.findAllOrders",
    query="select o from Order o")
public class Order { ... }

Query q = em.createNamedQuery("Order.findAllOrders");
```

Optimistic Locking

```
@Entity
public class Order {
     @Id private long pk;
     @Version private int oplock;
     ...
}
```

- **Version field can be:**
  - **numeric**
  - **temporal (java.util.Date, java.sql.Date, etc)**
- **At commit time, JPA checks that locked records have not changed in the database**

Optimistic Lock Types

- The JPA specification defines two lock types:
  - LockType.READ
    - Asserts that the record has not changed since the data was first read
  - LockType.WRITE
    - Performs a read lock
    - Increments the version field

Optimistic Lock Control

- **Optimistic write locks are automatically obtained for all changed or deleted records**
- **Optimistic read locks must be obtained explicitly:**

```
em.lock(
    em.getReference(Order.class, 17),
    LockType.READ)
```

- **Explicit optimistic write locks to 'touch' records:**

```
em.lock(
    em.getReference(Order.class, 17),
    LockType.WRITE)
```

getReference() and record deletion

Often, you may want to delete the record that corresponds to a given primary key

```
em.delete(em.find(Order.class, 17))
```

1. ```
   SELECT o.id, o.version, o.description
       FROM Order o WHERE o.id = 17
   ```
2. ```
   DELETE FROM Order o WHERE o.id = 17
       AND o.version = ?
   ```

getReference() and record deletion

`getReference()` may bypass the lookup that you would see when using `find()`

```
em.delete(em.getReference(Order.class, 17))
```

1. **`DELETE FROM Order o WHERE o.id = 17`**

- **DELETE JPQL statements can do this also, but are more opaque to the compiler, since they rely on a string definition language**

Disclaimers

- This is all subject to change
- I'm certainly forgetting some stuff
- This talk is not expert-group-approved

New Object Model Features

Collections and Maps

- **`@Entity`, `@Embeddable` and basic types can be used in Collections and Maps**

```
@Entity class PurchaseOrder {

    @OneToMany(mappedBy="order")
    Map<Product,LineItem> lineItems;
}
```

Collections and Maps

- **`@Entity`, `@Embeddable` and basic types can be used in Collections and Maps**

```
@Entity class MarketingEvent {
    @ElementCollection
    Collection<EventType> typeCodes;
}

enum EventType { CUSTOMER, SALES, MARKETING }
```

Mixed-mode access

```
@Entity @Access(FIELD) class Person {
    @Id int pk;
    @Transient String name;

    @Access(PROPERTY) String getLastName() {
        return extractLastName(name);
    }
    void setLastName(String lastName) { ... }
}
```

Related entities as primary keys

```
@IdClass(LineItemPK.class)

@Entity class LineItem {

    @Id int itemNumber;

    @Id @ManyToOne PurchaseOrder owner;

}
```

`@Embeddable` classes

- Can contain most JPA field types
  - `@Embeddable`
  - `@Entity`
  - Collections and Maps of basic or `@Embeddable` types

@OrderColumn

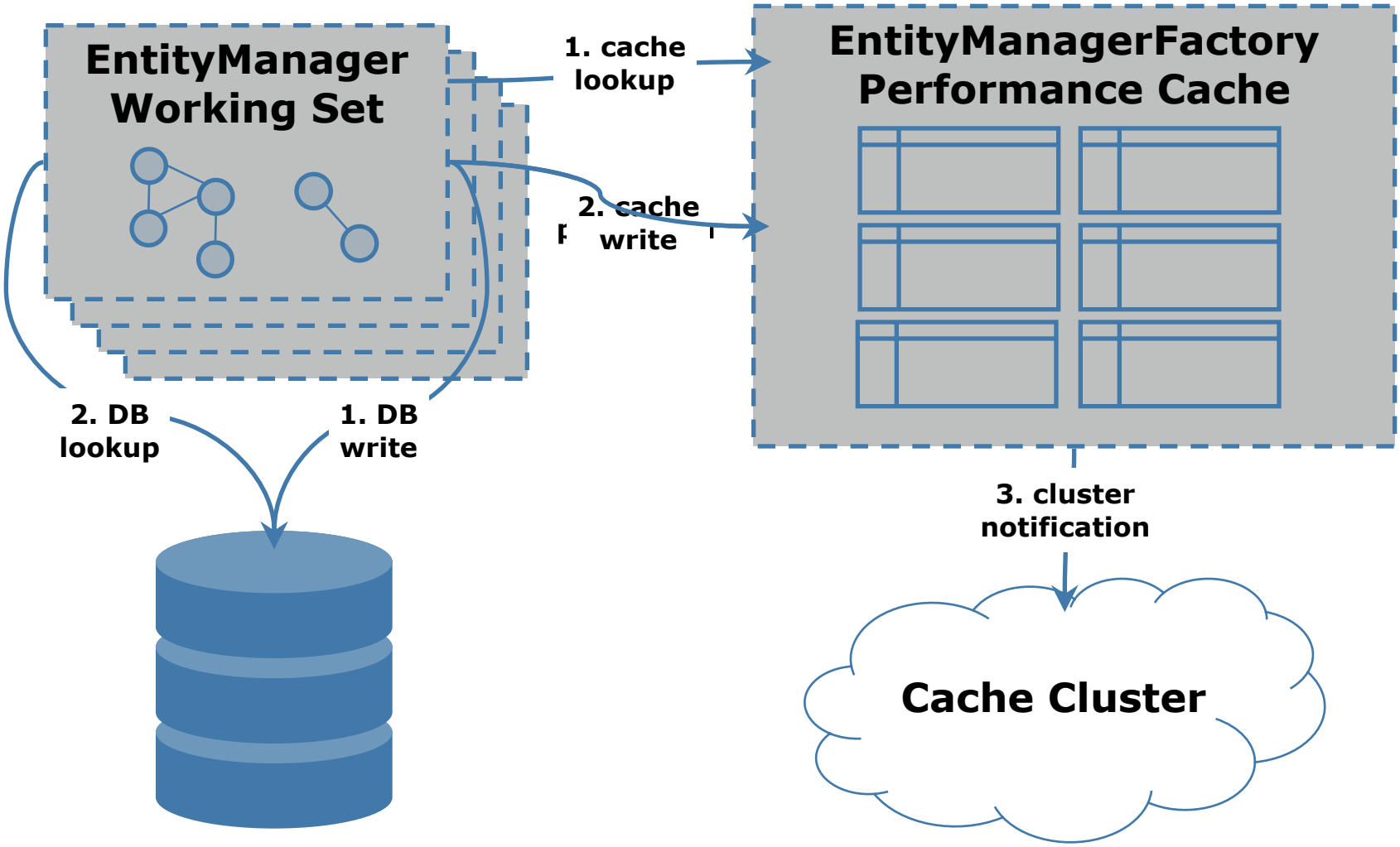
- Preserves order as defined by the List
- Different than `@OrderBy`

- Position info can be used in JPQL

```
select o from Order o
    where o.lineItems[3] = :item
```

New API Methods

Performance Caching



Performance Cache Maintenance

```
// returned object is never null
Cache cache = em.getEntityManagerFactory().getCache();


// cache maintenance after out-of-band updates
cache.evict(PurchaseOrder.class);
cache.evict(LineItem.class);


// more-targeted maintenance, plus interrogation
if (cache.contains(Person.class, 17)
    cache.evict(Person.class, 17);


// less-targeted maintenance
cache.evictAll();
```

Explicit Detachment

- `void EntityManager.clear(T root);`
- **Still working on semantic details**
  - **Detach-in-place vs. detach-a-copy**
  - **Detach graph size and definition**

Data Access Pattern Definition

- `EntityManager.isLoaded(`
  `Object entity, String field)`

- `EntityManager.load(`
  `Object entity, String field)?`

- `EntityManager.load(`
  `Object entity, AccessPattern ap)?`

- **Something via Query?**

Programmatic Query Definition

- **Prior Art**
  - **Hibernate Criteria**
  - **TopLink ExpressionFactory**
  - **Quaere (http://quaere.codehaus.org)**
- **JPA 2 Expert Group Status**
  - **Forming a subgroup**
  - **Hoping to reuse load graph definition**

Locking

```
enum LockModeType {
    NONE,
    OPTIMISTIC,
    OPTIMISTIC_FORCE_INCREMENT,
    PESSIMISTIC,
    PESSIMISTIC_FORCE_INCREMENT
}
```

```
interface EntityManager {

    <T> T find(Class<T> entityType,
                Object pk, LockModeType lockMode);

    void refresh(Object e, LockModeType mode);
    void lock(Object e, LockModeType lockMode);

    LockModeType getLockMode();
}

interface Query {
    Query setLockMode(LockModeType type);
    LockMode getLockMode();
}
```

Query API Introspection

```
interface Query {
    ...
    int getFirstResult();
    int getMaxResults();

    String getName();
    QueryType getQueryType();
    String getQueryString();

    Set<String> getParameterNames();
    Object getNamedParameter(String name);
    Set<String> getHints();
}
```

EntityManager API Additions

```
interface EntityManager {
    ...
    EntityManagerFactory getEntityManagerFactory();
    Object getId(Object entity);

    // the properties this EM was configured with
    Map<String, Object> getProperties();

    // the properties that could have been used
    // when creating this EM
    Set<String> getSupportedProperties();
}
```

New Configuration Settings

Section Format

```
<persistence>
    <persistence-unit name="foo">
        <properties>
            <property name="new-property"
                value="value"/>
        </properties>
    </persistence-unit>
</persistence>

<persistence>
    <persistence-unit name="foo">
        <new-setting>value</new-setting>
    </persistence-unit>
</persistence>
```

Java SE JDBC Configuration

`jdbc-driver`	**Fully-qualified name of the driver class**
`jdbc-url`	**Driver-specific URL for connecting to the database**
`jdbc-user`	**Username for creating connections**
`jdbc-password`	**Password for creating connections**

Performance Cache Configuration

`enable-secondary-cache`	**Controls whether the EntityManagerFactory cache is enabled for this persistence unit**
`logging.show-sql`	**Whether or not to output SQL**
`schema.generate`	**values might include** `create | force-create | update`

Things On My List

Named Query Namespace Improvements

```
@NamedQuery(name="findAll",
    query="select p from Person p")
@Entity
public class Person { ... }
```

```
@NamedQuery(name="findAll",
    query="select po from PurchaseOrder po")
@Entity
public class PurchaseOrder { ... }
```

```
Query q = em.createNamedQuery("findAll");
```

```
Query q = em.createNamedQuery(Person.class, "findAll");
```

Detachment and Lazy Loading

```
Person p = em.find(Person.class, 17);
Person detached = em.detach(p);


// unloaded fields set to Java defaults
assert detached.getFriends() == null;


// unloaded field access throws exception
try {
    detached.getFriends();
} catch (FieldNotLoadedException e) {
    // handle exception
}
```

`@GeneratedValue` on non-PK fields

```
@Entity class Person {

    @Id @GeneratedValue int pk;

    @GeneratedValue UUID uuid;

}
```

orm.xml:persistence-unit-defaults

- **Parts of `orm.xml` belong in `persistence.xml`**
  - **`cascade-persist`**
  - **`entity-listeners`**

`cascade-persist` Enabled By Default

```
@Entity class Person {
    @OneToOne Person spouse;
}

Person p = new Person("Fred");
p.setSpouse(new Person("Wilma");
em.persist(p);

=> throws an IKnowWhatYouWantedToDoException
```

Externalization

```
@Entity public class Person {
    ...
    @Externalizer("toExternalForm")
    private URL homePage;
}
```

- This example will automatically use '`new URL(String)`' to reverse the process
- Other options for generating the domain type are available via `@Factory` annotatio

External Values

- **Lookup tables: a special case of externalization**
- **Often useful for supporting enumerated type (or even Java enum types)**

```
@Entity
public class Magazine {

    @ExternalValues({"true=1", "false=2"})
    @Type(int.class)
    private boolean isWeekly;
}
```

Questions