



# Java Persistence API

# Topics

- **Introduction to java persistence**
- The Java Persistence API
  - Entities
  - EntityManager & the Persistent Context
  - Persistence Units
  - Exceptions
  - JPA Query Language

# Introduction

- Previously we learnt about
    - JDBC
1. In JDBC, we "hard coded" SQL into our application
  2. Then used Data Source/Connection Pooling
  3. Then used DAO
  4. But this just "hides" implementation from our business logic, you still implement DAO with JDBC

# Issues not solved

- However,
  - We still have to understand a lot of implementation details (eg: connections, statements, resultsets etc)
  - What about relationships? Joins? Inheritance?
  - Object  $\leftrightarrow$  database impedance mismatch
- J2EE tried to solve this with "Entity Enterprise JavaBeans (EJB)"
- Simpler alternatives included Object Relational Mapping (ORM) tools:
  - e.g. Java Data Objects (JDO), Hibernate, iBatis, TopLink

# Java EE 5/6 to the rescue

- Java SE 5/6 added new constructs to Java language
  - Generics
  - Annotations
  - Enumerations
- Java EE 5/6 used these features to provide
  - New Java Persistence API (JPA) replaced Entity EJB

# Java EE 5/6 persistence

- Java EE 5/6 still keeps JDBC
- EJB 3 spec (JSR 220) split into 2:
  1. Session Beans, Message Beans
  2. Java Persistence API (JPA)
- JPA jointly developed by TopLink, Hibernate, JDO, EJB vendors and individuals
- JPA can also be used in Java SE 5 without a container!!!!

# Java Persistence

- Java Persistence consists of three areas:
  - The Java Persistence API
  - The query language
  - Object/relational mapping metadata
- JPA implementation
  - Reference implementation: TopLink (GlassFish project)
  - Most ORM vendors now have JPA interface
    - eg: Hibernate-JPA, EclipseLink (based on TopLink), OpenJPA (based on BEA Kodo)

# Topics

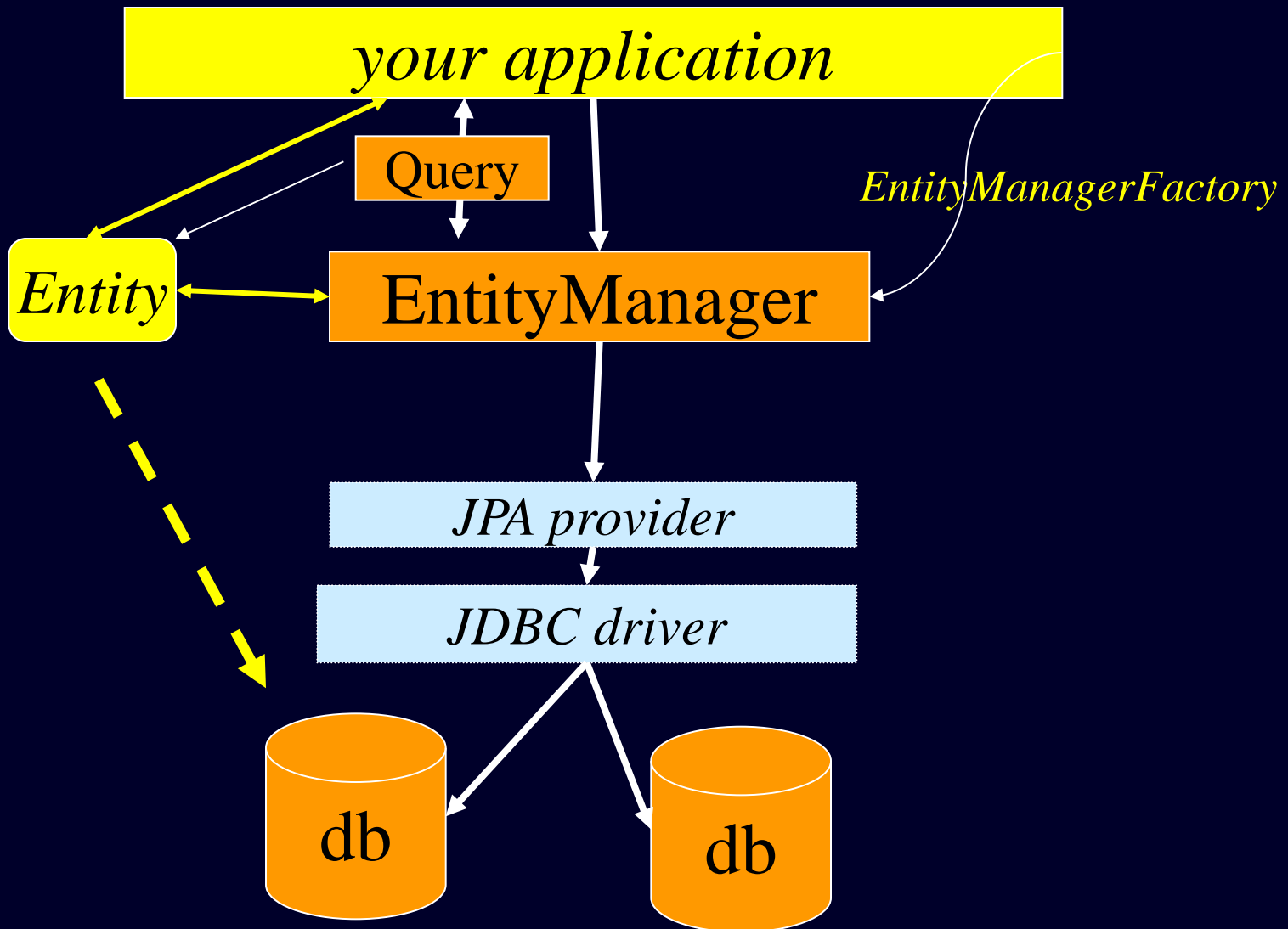
- Introduction to java persistence
- **The Java Persistence API**
  - **Entities**
  - EntityManager & the Persistent Context
  - Persistence Units
  - Exceptions
  - JPA Query Language



# Java Persistence API

- javax.persistence.\*
  - EntityManager
  - EntityManagerFactory
  - EntityTransaction
  - Query
  - "*Entity*" – we use Plain Old Java Objects (POJO) instead.

# JPA classes



# Entities

- An **entity** is a plain old java object (POJO)
- The **Class** represents a **table** in a relational database.
- **Instances** correspond to **rows**
- Requirements:
  - annotated with the **`javax.persistence.Entity`** annotation
  - **`public`** or **`protected`**, no-argument constructor
  - the class must not be declared **`final`**
  - no methods or persistent instance variables must be declared **`final`**

# Requirements for Entities (cont.)

- May be `Serializable`, but not required
  - Only needed if passed by value (in a remote call)
- Entities may extend both entity and non-entity classes
- Non-entity classes may extend entity classes
- Persistent instance variables must be declared private, protected, or package-private
- No required business/callback interfaces

- Example:

```
@Entity
class Person{
    . . .
}
```

# Persistent Fields and Properties

- The persistent state of an entity can be accessed:
  - through the entity's **instance variables**
  - through **JavaBeans-style properties** (getters/setters)
- Supported types:
  - primitive types, String, other serializable types, enumerated types
  - other entities and/or collections of entities
  - embeddable classes
- All fields not annotated with **@Transient** or not marked as Java transient will be persisted to the data store!

# Primary Keys in Entities

- Each entity must have a unique object identifier (persistent identifier)

**@Entity**

```
public class Employee {
```

```
    @Id private int id;
```

```
    private String name;
```

```
    private Date age;
```

```
    public int getId() { return id; }
```

```
    public void setId(int id) { this.id = id; }
```

```
    . . .
```

```
}
```

Primary key



# Persistent Identity

- Identifier (id) in entity = primary key in database
- Uniquely identifies entity in memory and in DB
- Persistent identity types:
  - Simple id – single field/property  
`@Id int id;`
  - Compound id – multiple fields/properties  
`@Id int id;`  
`@Id String name;`
  - Embedded id – single field of PK class type  
`@EmbeddedId EmployeePK id;`

# Identifier Generation

- Identifiers can be generated in the database by specifying **@GeneratedValue** on the identifier
- Four pre-defined generation strategies:
  - AUTO, IDENTITY, SEQUENCE, TABLE
- Generators may pre-exist or be generated
- Specifying strategy of AUTO indicates that the provider will choose a strategy

```
@Id @GeneratedValue(strategy=AUTO)  
private int id;
```



# Customizing the Entity Object

- In most of the cases, the defaults are sufficient
- By default the table name corresponds to the unqualified name of the class
- Customization:

```
@Entity
@Table(name = "FULLTIME_EMPLOYEE")
public class Employee{ ..... }
```

- The defaults of columns can be customized using the @Column annotation

```
@Id @Column(name = "EMPLOYEE_ID", nullable = false)
private String id;

@Column(name = "FULL_NAME" nullable = true, length = 100)
private String name;
```

# Entity Relationships

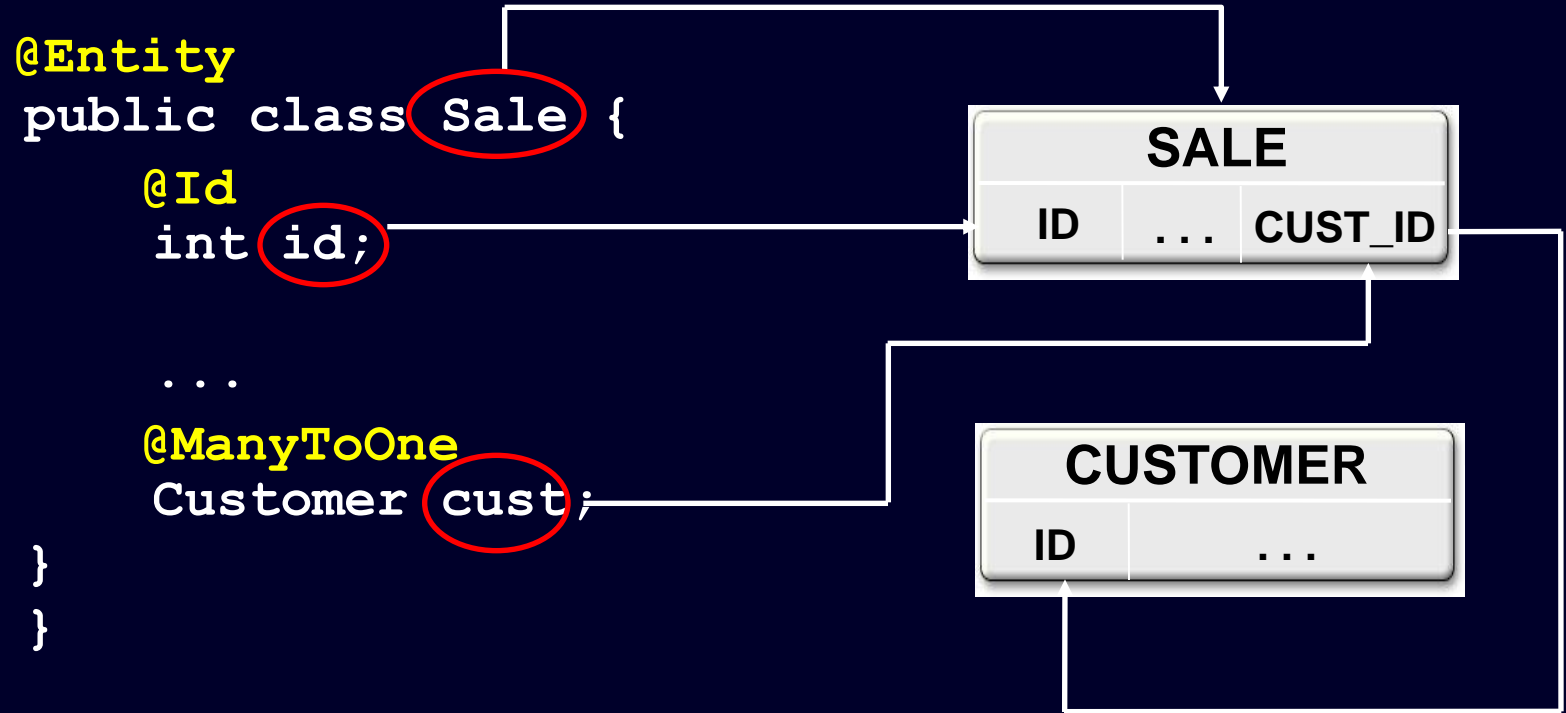
- There are four types of relationship multiplicities:
  - @OneToOne
  - @OneToMany
  - @ManyToOne
  - @ManyToMany
- The direction of a relationship can be:
  - **bidirectional** – owning side and inverse side
  - **unidirectional** – owning side only
- Owning side specifies the physical mapping

# Entity Relation Attributes

- JPA supports cascading updates/deletes
  - CascadeType
    - ALL, PERSIST, MERGE, REMOVE, REFRESH
- You can declare performance strategy to use with fetching related rows
  - FetchType
    - LAZY, EAGER
  - (Lazy means don't load row until the property is retrieved)

```
@ManyToMany(  
    cascade = {CascadeType.PERSIST, CascadeType.MERGE},  
    fetch = FetchType.EAGER)
```

# ManyToOne Mapping



# OneToMany Mapping

```
@Entity  
public class Customer {
```

```
    @Id  
    int id;
```

```
    ...
```

```
    @OneToMany(mappedBy="cust"  
    Set<Sale> sales;
```

```
}
```

```
@Entity  
public class Sale {
```

```
    @Id  
    int id;
```

```
    ...
```

```
    @ManyToOne  
    Customer cust;
```

```
}
```



# ManyToMany Mapping

```
@Entity
public class Customer {
    ...
    @JoinTable(
        name="CUSTOMER_SALE",
        joinColumns=@JoinColumn(
name="CUSTOMER_ID",referencedColumnName="customer_id"),
        inverseJoinColumns=@JoinColumn(
name="SALE_ID", referencesColumnName="sale_id")
        Collection<Sale> sales;
    }
```

```
@Entity
public class Sale {
    ...
    @ManyToMany(mappedBy="sales")
    Collection<Customer> customers;
}
```

# Entity Inheritance

- An important capability of the JPA is its support for inheritance and polymorphism
- Entities can inherit from other entities and from non-entities
- The **@Inheritance** annotation identifies a mapping strategy:
  - SINGLE\_TABLE
  - JOINED
  - TABLE\_PER\_CLASS

# Inheritance Example

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DISC",
                    discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue(name="CUSTOMER")
public class Customer { . . . }

@Entity
@DiscriminatorValue(name="VCUSTOMER")
public class ValuedCustomer extends Customer { . . . }
```

- SINGLE\_TABLE strategy - all classes in the hierarchy are mapped to a single table in the database
- **Discriminator column** - contains a value that identifies the subclass
- **Discriminator type** - {STRING, CHAR, INTEGER}
- **Discriminator value** - value entered into the discriminator column for each entity in a class hierarchy



# Topics

- Introduction to java persistence
- The Java Persistence API
  - Entities
  - **EntityManager & the Persistent Context**
  - Persistence Units
  - Exceptions
  - JPA Query Language

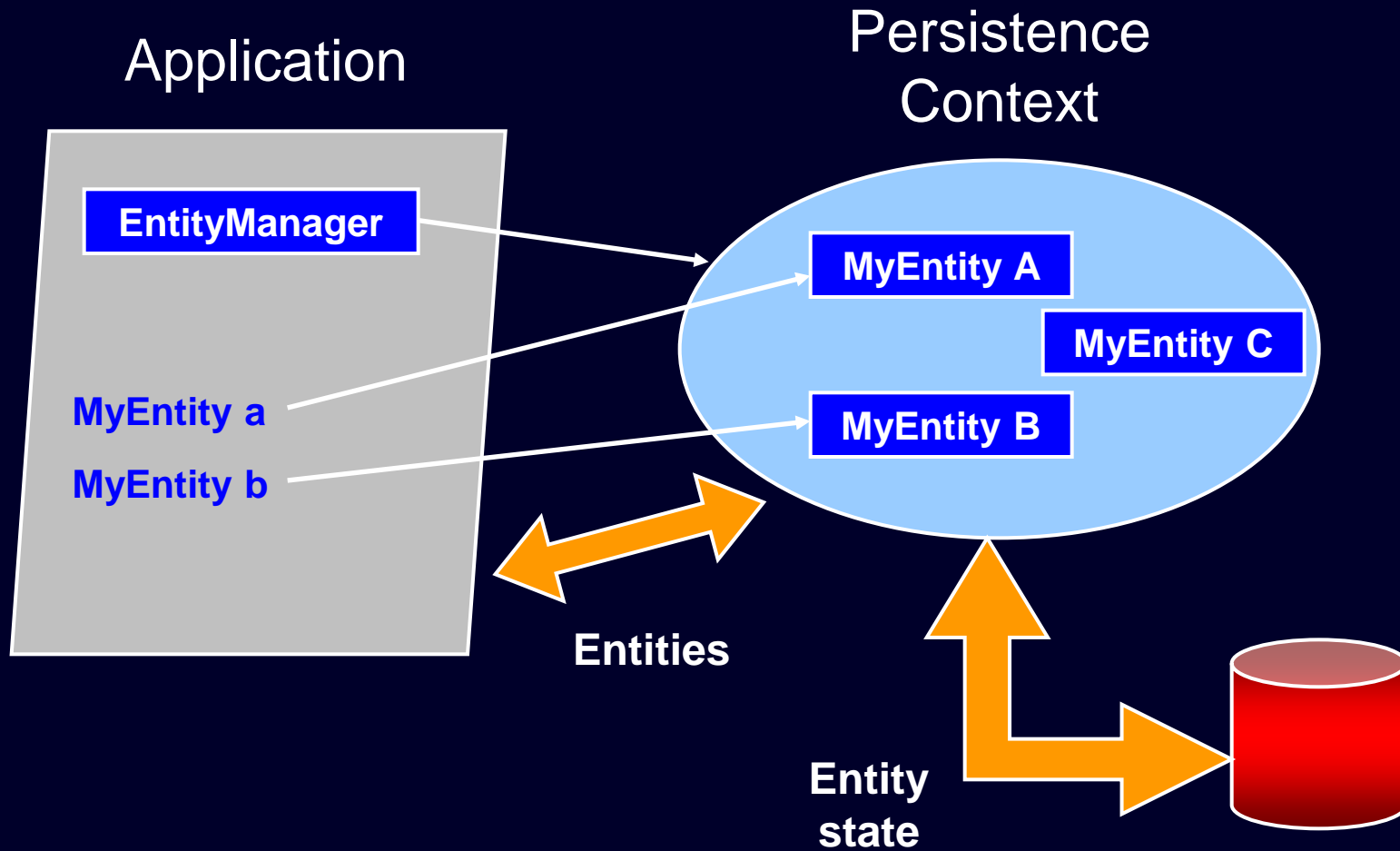
# Managing Entities

- Entities are managed by the **entity manager**
- The entity manager is represented by `javax.persistence.EntityManager` instances
- Each EntityManager instance is associated with a **persistence context**
- A persistence context defines the scope under which particular entity instances are created, persisted, and removed

# Persistence Context

- A **persistence context** is a set of managed entity instances that exist in a particular data store
  - Entities **keyed** by their persistent **identity**
  - Only **one** entity with a given persistent identity may exist in the persistence context
  - Entities are added to the persistence context, but are not individually removable (“detached”)
- Controlled and managed by **EntityManager**
  - Contents of persistence context change as a result of operations on EntityManager API

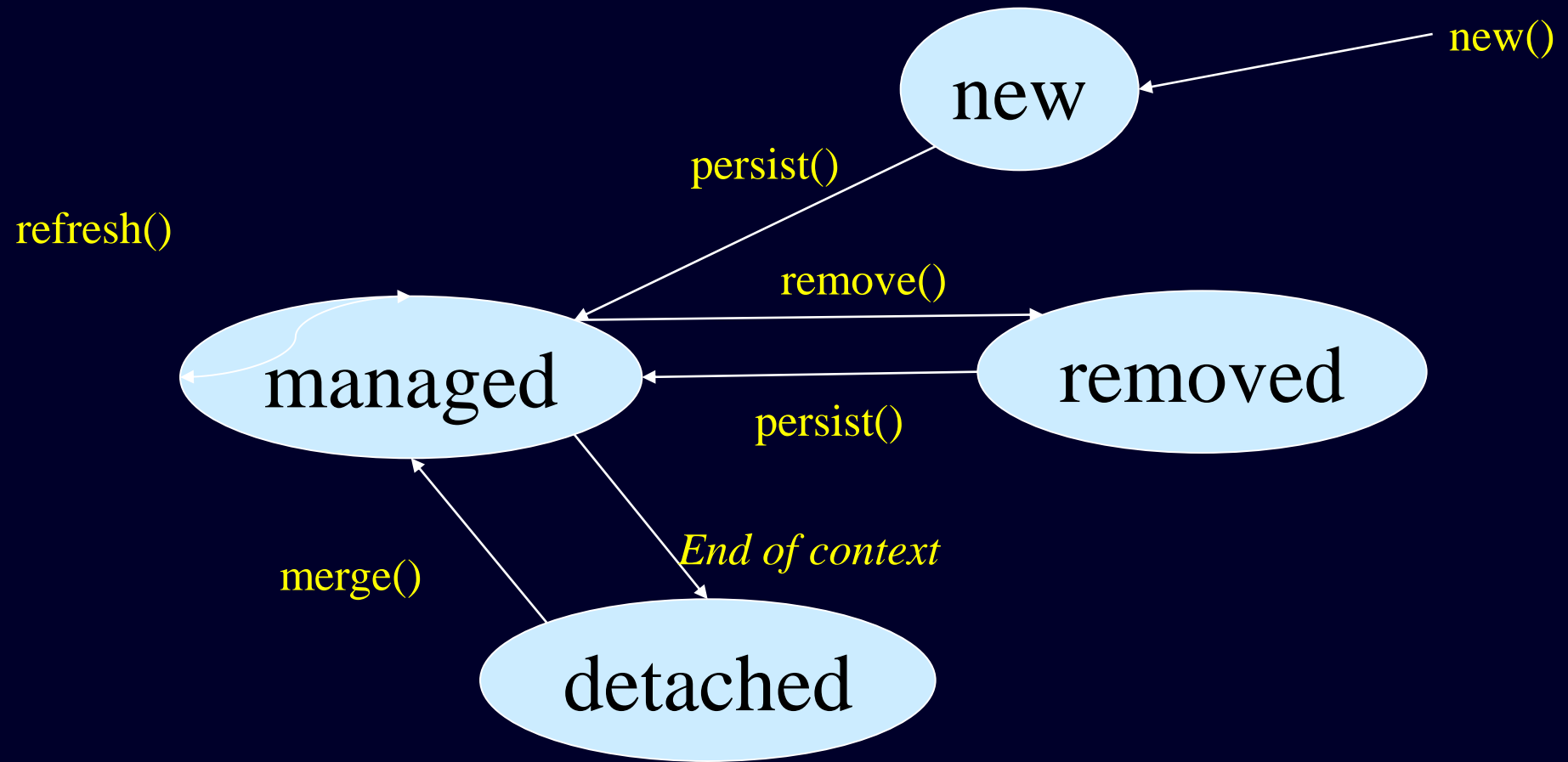
# Persistence Context



# Entity Manager

- An **EntityManager** instance is used to manage the state and life cycle of entities within a persistence context
- Entities can be in one of the following states:
  1. New
  2. Managed
  3. Detached
  4. Removed

# Entity Lifecycle



# Entity Lifecycle

- **New** – entity is instantiated but not associated with persistence context. Not linked to database.
- **Managed** – associated with persistence context. Changes get synchronised with database
- **Detached** – has an id, but not connected to database
- **Removed** – associated with persistence context, but underlying row will be deleted.

The **state** of persistent entities is synchronized to the database when the transaction **commits**

# Entity Manager

- The EntityManager API:
  - **creates** and **removes** persistent entity instances
  - **finds** entities by the entity's primary key
  - allows **queries** to be run on entities
- There are two types of EntityManagers:
  - **Application-Managed** EntityManagers
    - ie: run via Java SE
  - **Container-Managed** EntityManagers
    - ie: run via Java EE Container eg: Tomcat



# Application-Managed EntityManager

Java SE applications create EntityManager instances by using **directly** `Persistence` and `EntityManagerFactory`:

- **`javax.persistence.Persistence`**
  - Root class for obtaining an EntityManager
  - Locates provider service for a named persistence unit
  - Invokes on the provider to obtain an EntityManagerFactory
- **`javax.persistence.EntityManagerFactory`**
  - Creates EntityManagers for a named persistence unit or configuration

# Application-Managed EntityManager

- Applications must manage own transactions too..

```
public class PersistenceProgram {
    public static void main(String[] args)
    {
        EntityManagerFactory emf =
Persistence.createEntityManagerFactory("SomePUnit");
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        // Perform finds, execute queries,
...
        // update entities, etc.
        em.getTransaction().commit();
        em.close();
        emf.close();
    }
}
```

# Container-Managed EntityManager

- Containers provide naming and transaction services for JPA
  - (eg: Web Container like Tomcat, EJB Container like WebLogic)
- JPA relies on the container to insert the actual reference to the EntityManager for the current context via ***dependency injection***
- Use the Java5/6 annotations to do this

# Container-Managed EntityManager

- An EntityManager with a transactional persistence context can be **injected** by using the **@PersistenceContext** annotation

```
@PersistenceContext (unitName="SomePUUnit")
    EntityManager em;
// Perform finds, execute queries,
...
// update entities, etc.
em.close();
```

container inserts  
reference to the  
container's  
EntityManager

- You could also use the **@Resource(name="jndi:name")** annotation to insert a named entity manager.

# Transactions

- JPA transactions can be managed by:
  - the users application
  - a framework (such as Spring)
  - a Java EE container
- Transactions can be controller in two ways:
  - **Java Transaction API (JTA)**
    - container-managed entity manager
  - **EntityTransaction API** (`tx.begin()`, `tx.commit()`, etc)
    - application-managed entity manager

# Operations on Entity Objects

- EntityManager API operations:
  - **persist()** - Save the entity into the db
  - **remove()** - Delete the entity from the db
  - **refresh()** - Reload the entity state from the db
  - **merge()** - Synchronize a detached entity with the p/c
  - **find()** - Find by primary key
  - **createQuery()** - Create query using dynamic JP QL
  - **createNamedQuery()** - Create a predefined query
  - **createNativeQuery()** - Create a native "pure" SQL query.  
Can also call stored procedures.
  - **contains()** - Is entity is managed by p/c
  - **flush()** - Force synchronization of p/c to database

Note: p/c == the current persistence context

# Topics

- Introduction to java persistence
- The Java Persistence API
  - Entities
  - EntityManager & the Persistent Context
  - **Persistence Units**
  - Exceptions
  - JPA Query Language

# Persistence Units

- A **persistence unit** defines a set of all entity classes that are managed by `EntityManager` instances in an application
- Each persistence unit can have different providers and database drivers
- Persistence units are defined by the **persistence.xml** configuration file



# persistence.xml

A **persistence.xml** file defines one or more persistence units

```
<persistence>
  <persistence-unit name="SomePUnit">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>myapp.MyEntity</class>
    <properties>
      <property name="hibernate.connection.url"
        value="jdbc:oracle:thin:@smaug.it.uts.edu.au:1522:ell"/>
      <property name="hibernate.connection.driver_class"
        value="oracle.jdbc.driver.OracleDriver"/>
      <property name="hibernate.connection.username"
        value="user"/>
      <property name="hibernate.connection.password"
        value="password"/>
    </properties>
  </persistence-unit>
</persistence>
```

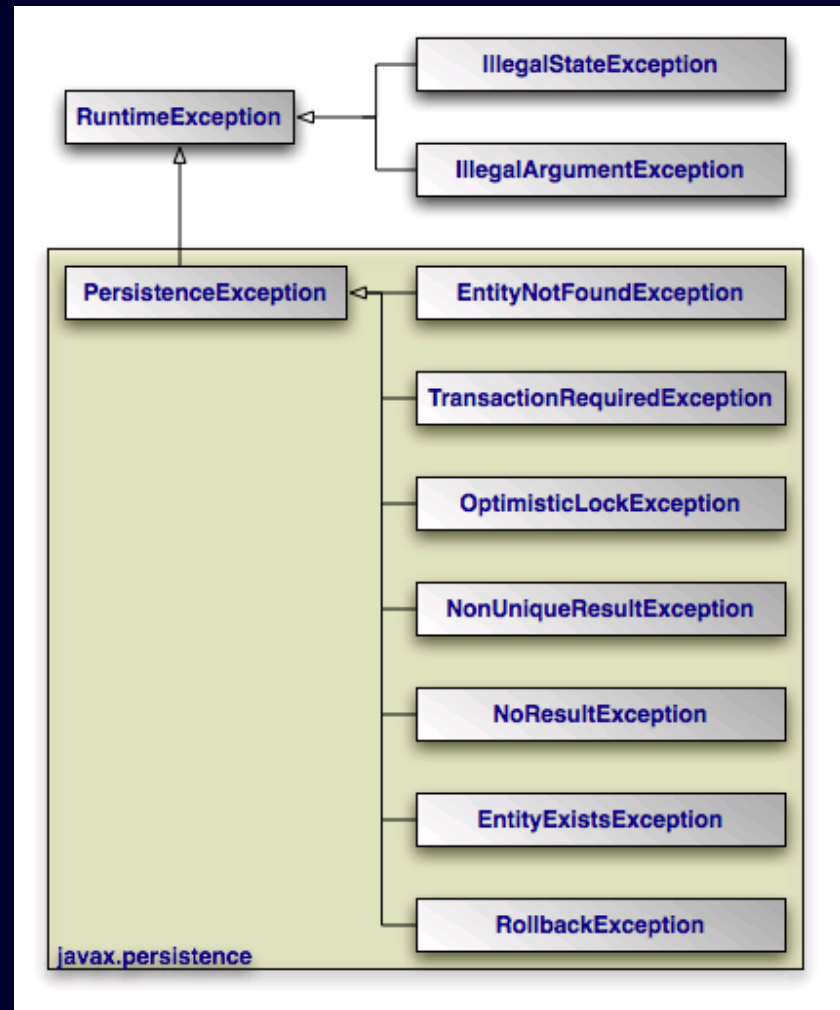
# persistence.xml

You can also use JNDI + datasource in `persistence.xml` instead of hard coding driver details. Requires container to manage this.

```
<persistence>
  <persistence-unit name=" SomePUnit">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/thinOracleDataSource</jta-data-source>
    <class>myapp.MyEntity</class>
  </persistence-unit>
</persistence>
```

# JPA exceptions

- All exceptions are unchecked
- Exceptions in `javax.persistence` package are self-explanatory



# Topics

- Introduction to java persistence
- The Java Persistence API
  - Entities
  - EntityManager & the Persistent Context
  - Persistence Units
  - Exceptions
  - **JPA Query Language**

# JPQL Introduction

- JPA has a query language based on SQL
- JPQL is an extension of EJB QL
- More robust flexible and object-oriented than SQL
- The persistence engine parses the query string, transform the JPQL to the native SQL before executing it

# Creating Queries

- Query instances are obtained using:
  - EntityManager.**createNamedQuery** (static query)
  - EntityManager.**createQuery** (dynamic query)
  - EntityManager.**createNativeQuery** (native query)

## Query API:

- **getResultList()** – execute query returning multiple results
- **getSingleResult()** – execute query returning **single** result
- **executeUpdate()** – execute bulk update or delete
- **setFirstResult()** – set the first result to retrieve
- **setMaxResults()** – set the maximum number of results to retrieve
- **setParameter()** – bind a value to a named or positional parameter
- **setHint()** – apply a vendor-specific hint to the query
- **setFlushMode()** – apply a flush mode to the query when it gets run

# Static (Named) Queries

- Defined statically with the help of **@NamedQuery** annotation together with the entity class
- @NamedQuery elements:
  - **name** - the name of the query that will be used with the createNamedQuery method
  - **query** – query string

```
@NamedQuery(name="findAllCustomers",  
            query="SELECT c FROM Customer")
```

```
Query findAllQuery =  
entityManager.createNamedQuery("findAllCustomers");  
List customers = findAllQuery.getResultList();
```

# Multiple Named Queries

Multiple named queries can be logically defined with the help of **@NamedQueries** annotation

```
@NamedQueries( {  
    @NamedQuery(name = "Mobile.selectAllQuery"  
        query = "SELECT M FROM MOBILEENTITY"),  
    @NamedQuery(name = "Mobile.deleteAllQuery"  
        query = "DELETE M FROM MOBILEENTITY")  
} )
```



# Dynamic Queries

- Dynamic queries are queries that are defined directly within an application's business logic
- ! Not efficient & slower. Persistence engine has to parse, validate & map the JPQL to SQL at run-time

```
public List findAll(String entityName) {  
    return entityManager.createQuery(  
        "select e from " + entityName + " e")  
        .getResultList();  
}
```

# Named Parameters

- Named parameters are parameters in a query that are prefixed with a **colon** (:)
- To bound parameter to an argument use method:
  - **Query.setParameter**(String name, Object value)

```
public List findWithName(String name) {  
    return em.createQuery(  
        "SELECT c FROM Customer c WHERE c.name LIKE :custName")  
        .setParameter("custName", name)  
        .getResultList();  
}
```

# Positional Parameters

- Positional parameters are prefixed with a **question mark (?)** & number of the parameter in the query
- To set parameter values use method:
  - `Query.setParameter(integer position, Object value)`

```
public List findWithName(String name) {  
    return em.createQuery(  
        "SELECT c FROM Customer c WHERE c.name LIKE ?1")  
        .setParameter(1, name)  
        .getResultList();  
}
```

# Native Queries

- Queries may be expressed in native SQL
- Use when you need to use native SQL of the target database
- Can call stored procedures using "call procname" syntax

```
Query q = em.createNativeQuery(  
    "SELECT o.id, o.quantity, o.item " +  
    "FROM Order o, Item i " +  
    "WHERE (o.item = i.id) AND (i.name = 'widget')",  
    com.acme.Order.class);
```

- Use **@SqlResultSetMapping** annotation for more advanced cases

# Query Operations – Multiple Results

- `Query.getResultList()` will execute a query and may return a List object containing multiple entity instances

```
Query query = entityManager.createQuery("SELECT C FROM CUSTOMER");  
List<MobileEntity> mobiles = (List<MobileEntity>)query.getResultList();
```

- Can only execute on select statements as opposed to UPDATE or DELETE statements
  - For a statement other than SELECT run-time `IllegalStateException` will be thrown

# Query Operations – Single Result

- A query that returns a single entity object

```
Query singleSelectQuery = entityManager.createQuery(  
    "SELECT C FROM CUSTOMER WHERE C.ID = 'ABC-123'");  
Customer custObj = singleSelectQuery.getSingleResult();
```

- If the match wasn't successful, then **EntityNotFoundException** is returned
- If more than one matches occur during query execution a run-time exception **NonUniqueResultException** will be thrown

# Paging Query Results

```
int maxRecords = 10; int startPosition = 0;
String queryString = "SELECT M FROM MOBILEENTITY";
while(true){
    Query selectQuery = entityManager.createQuery(queryString);
    selectQuery.setMaxResults(maxRecords);
    selectQuery.setFirstResult(startPosition);
    List<MobileEntity> mobiles =
        entityManager.getResultList(queryString);
    if (mobiles.isEmpty()){ break; }
    process(mobiles);        // process the mobile entities
    entityManager.clear();  // detach the mobile objects
    startPosition = startPosition + mobiles.size();
}
```

# Flushing Query Objects

- Two modes of flushing query objects
  - **AUTO** (default) and **COMMIT**
- **AUTO** - any changes made to entity objects will be reflected the very next time when a SELECT query is made
- **COMMIT** - the persistence engine may only update all the state of the entities during the database COMMIT
- set via `Query.setFlushMode()`



# JPQL Statement Language

- JPQL statement types:
  - SELECT, UPDATE, DELETE
- Supported clauses:
  - FROM
  - WHERE
  - GROUP\_BY
  - HAVING
  - ORDER BY
  - ...
- Conditional expressions, aggregate functions,...

# OO-style vs. SQL-style queries

- The main difference:  
\*\*\* query the application model, i.e. the entities,  
rather than any database tables
- Better productivity by using OO-style queries, e.g.

`employee.getManager().getAddress()`

which becomes:

```
SELECT t3.* FROM EMP t1, EMP t2, ADDR t3
WHERE t1.EMP_ID = "XYZ" AND t1.MGR_ID = t2.EMP_ID
AND t2.ADDR_ID = t3.ADDR_ID
```

- Notice that the two-step object traversal was packed into a single DB query

**Q/A**

# Resources

- TopLink Essentials (reference implementation)

<https://glassfish.dev.java.net/javaee5/persistence/>

- JPA Annotation Reference

<http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html>

# Resources

- JPQL Language Reference

[http://openjpa.apache.org/builds/1.0.2/apache-openjpa-1.0.2/docs/manual/jpa\\_langref.html](http://openjpa.apache.org/builds/1.0.2/apache-openjpa-1.0.2/docs/manual/jpa_langref.html)

- JPA Query API

[http://www.javabeat.net/javabeat/ejb3/articles/2007/04/introduction\\_to\\_java\\_persistence\\_api\\_jpa\\_ejb\\_3\\_0\\_6](http://www.javabeat.net/javabeat/ejb3/articles/2007/04/introduction_to_java_persistence_api_jpa_ejb_3_0_6)

- Standardizing Java Persistence with the EJB3 Java Persistence API – Query API

<http://www.onjava.com/pub/a/onjava/2006/05/17/standardizing-with-ejb3-java-persistence-api.html?page=last&x-showcontent=text>