Persitence layer

ORM and Java Persistant API

Some slides from:

Enterprise JavaBean 3.0 & Java Persistence APIs: Simplifying Persistence

Carol McDonald, Java Architect

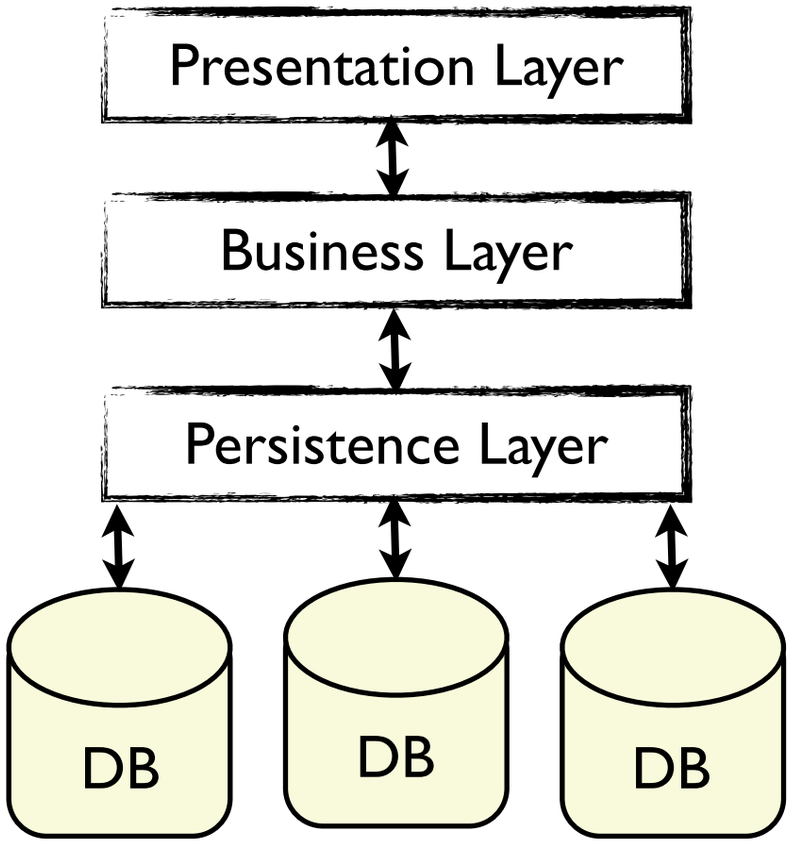
Sun Microsystems, Sun Tech Days 2006-2007

What's (data) persistence?

Data outlives programs

- one of the fundamental concepts in application development
- Issues
  - ★ storage, organization, and retrieval of structured data
  - ★ concurrency
  - ★ data integrity
  - ★ data sharing
  - ★ *model mismatches*
- in most situations, it amounts to store data in a relational database using SQL

Persistence layer



Persistence in OO world

allows an object to outlive the process that created it

- **Transient** object vs. **persistent** object
- Ability to
  - ★ store the state of an object in a disk,
  - ★ re-create an object with the same state

  ... not only for an isolated object but also for an entire **networks** of **interconnected** objects
- Typical applications contain a mix of persistent and transient objects

Implementation of persistence layers

- hand-coding with SQL/JDBC
  - ★ SQL/JDBC, DAO, ...
- Serialization
  - ★ Java built-in persistent mechanism : application states -> byte stream -> file/DB
  - (–) serialized network can only be accessed as a whole
- OODBMS
- ORM

Object/relational paradigm mismatch

Object Model

Object-oriented

- Object class
- Object (Instance)
- Identity
- Complex types
- Inheritance
- Polymorphism

Relational Model

Value-oriented

- Relation schema
- Tuple (row)
- Key
- Foreign key
- 1 NF

...but modern DBMS are non-first NF

*«up to **30 percent** of the Java application code written is to handle the tedious SQL/JDBC and manual bridging of the object/relational paradigm mismatch»*

Object/relational mismatch

- Granularity
- Subtypes
- Object identity
- Association
- Data navigation

Granularity

- Relational mapping of Address

★ Set of attributes in the table User

★ A separate Table

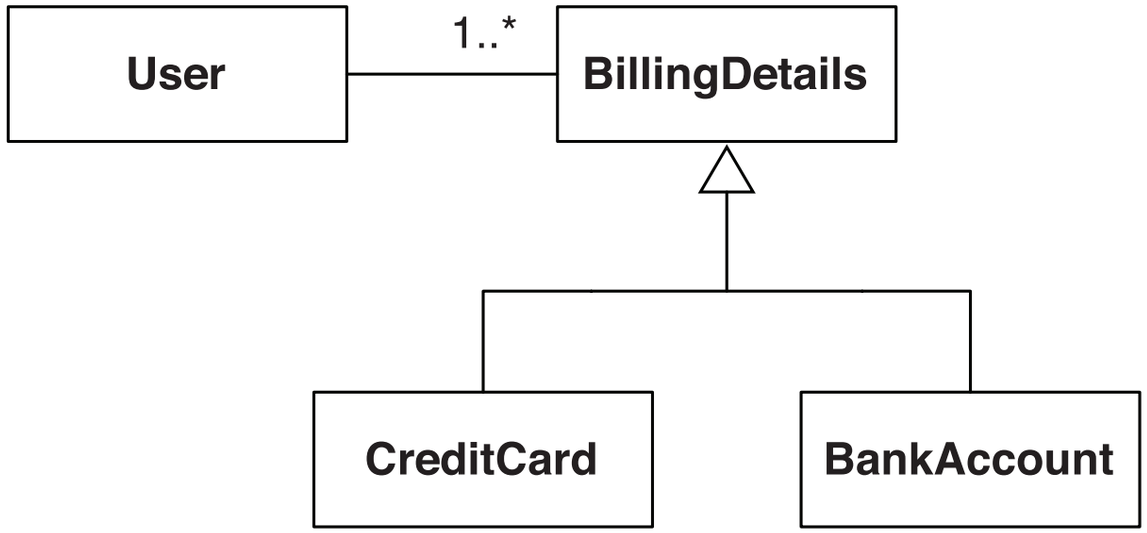
★ UDT (user-defined datatype)

- Model mistmatch

Domain model: classes with different levels of granularity (e.g., User, Address, Zipcode, emailaddress, ..)

Relational model: 2 levels of granularity (table, column)

Subtypes



- Problems
  - ★ Representation of a class hierarchy
  - ★ Polymorphism

Object identity

- Object identity
  - OO world: identity vs. equality
  - RDB: equality of PK

Association

- OO World: object references
  - object references are directional (need of an explicit inverse)
  - many-to-many
- RDB: FK
  - FK are not directional
  - one-to-one or one-to-many

Data navigation

- Different access mechanisms

✳ OO World: navigation in a nerwork of objects

✳ RDB: join

➡ Translation of navigation to SQL queries is inefficient

n+1 selects problem

ORM: pros

- Productivity
- Maintainability
- Vendor independence
- Performance

JPA

Java Persistent API

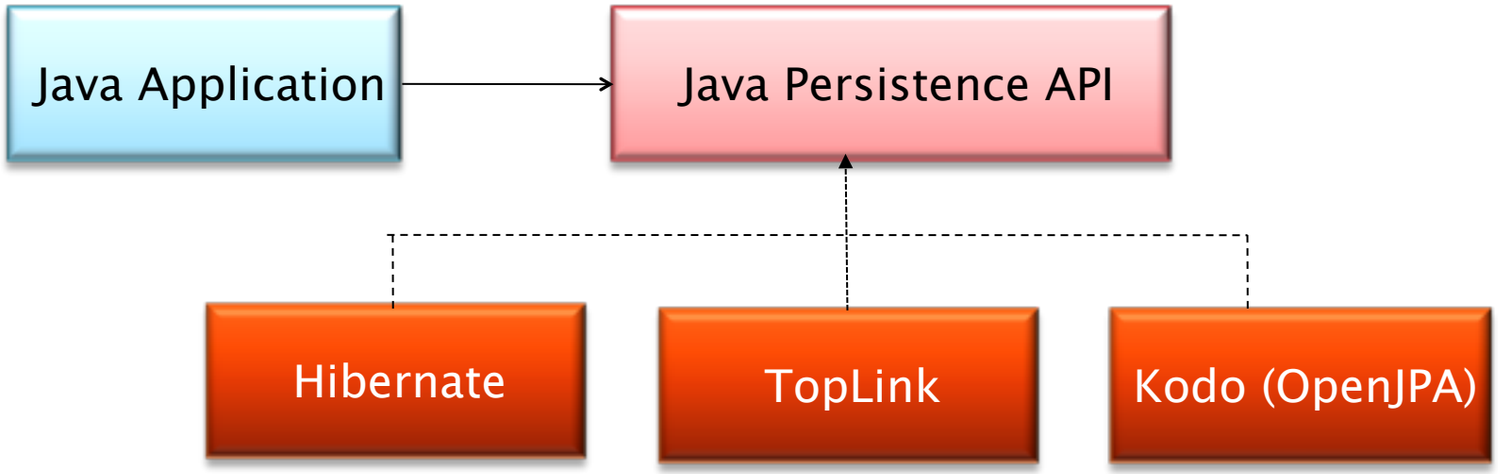
JPA

- Specification released under JEE 5 specification
- An implementation of the persistence part of Enterprise Java Beans 3.0
- JPA defines an interface to perform CRUD operations between POJO's and a data store

JPA (cont.)

- Standard API for object/relational persistence
- Automatic mapping
  - ★ Annotations vs. XML configuration file
  - ★ Useful default: configure by exception only (configure only to override default)
- Support for pluggable 3rd party Persistence providers
- Usable in Java EE, Enterprise JavaBeans 3.0 and Java SE environments
- JPQL (Java Peristence Query Language)
  - ★ SQL-Like query language
  - ★ Static and dynamic queries

JPA (cont.)



Main components

- A set of interfaces and classes to isolate client applications from persistence providers
- A set of annotations to specify mappings between java classes and relational tables
- A persistence provider
- An XML file, *persistence.xml*, that configures the persistence (provider, datasource, ..)

JPA: essential ORM

- Entities
- Basic types
- Embeddable classes
- Relationships
- Inheritance

What's an entity ?

- A lightweight persistence domain object

Entity class ⟶ Relational table

Entity ⟶ a row in a table

- Persistent states: persistent fields or persistent properties
- Object/Relational mapping annotations

  to map entities and entity relationships to relational structures in a relational database

➡ Support fine-grained domain model: *more classes than tables*

Entities

- Plain Old Java Objects (not an EJB)
- Created by means of new
- No required interfaces (home, ejb, callback..)
- Have persistent identity
- May have both persistent and non-persistent state
- Simple types (e.g., primitives, wrappers, enums, serializables)
- Composite dependent object types (e.g., Address)
- Non-persistent state (transient or @Transient)
- Can extend other Entity and non-Entity classes
- Serializable; usable as detached objects in other tiers
  - ★ No need for data transfer objects

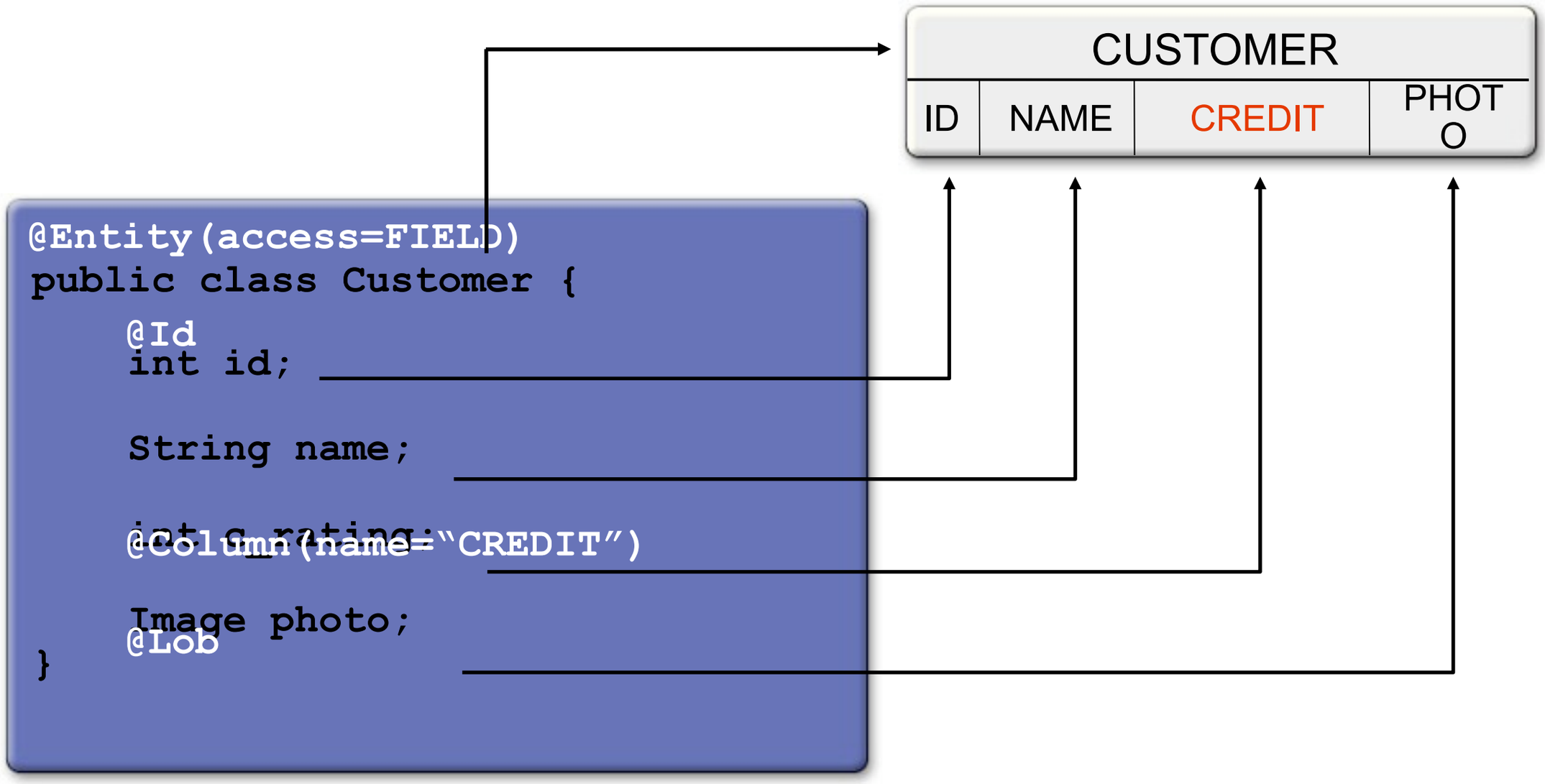
Example

Annotated as “Entity”

@Id denotes primary key

```
@Entity
public class Customer implements Serializable {
   @Id protected Long id;
   protected String name;

   public Customer() {}

   public Long getId() {return id;}


   public String getName() {return name;}
   public void setName(String name) {this.name = name;}

   …
}
```

Simple Mapping



Mapping defaults to matching column name. Only configure if entity field and table column names are different

Entity

Annotated as “Entity”

@Id denotes primary key

```
@Entity
public class Customer implements Serializable {
   @Id protected Long id;
   protected String name;
   @Transient protected int reservationCount;

   public Customer() {}

   public Long getId() {return id;}


   public String getName() {return name;}
   public void setName(String name) {this.name = name;}

   …
}
```

Entity Class for Customer

Annotated as “Entity”

Data are accessed as fields

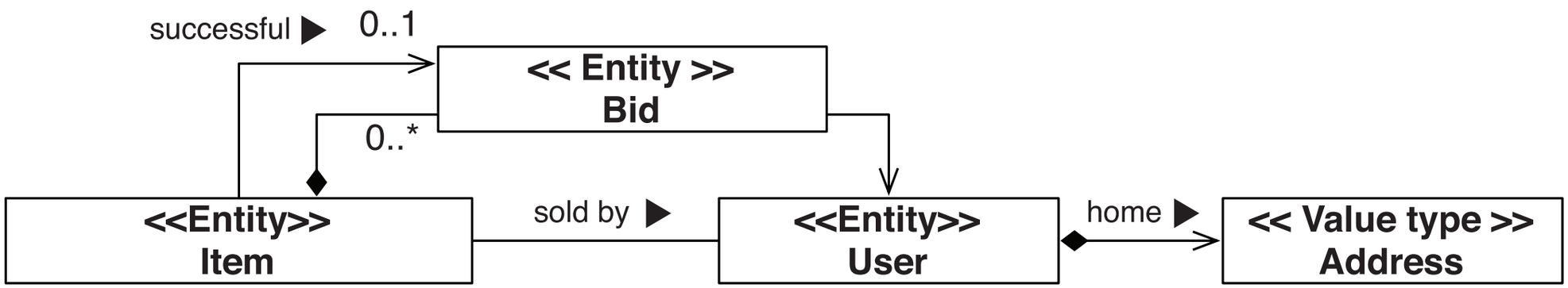
```
@Entity(access=FIELD)
@Table(name = “customer”)
public class Customer {
   @Id public int id;
   ...
   public String name;
   @Column(name=”CREDIT”) public int c_rating;

   @LOB public Image photo;
   ...
}
```

Maps to “customer” table

@Id denotes primary key

Specify the table column to map to

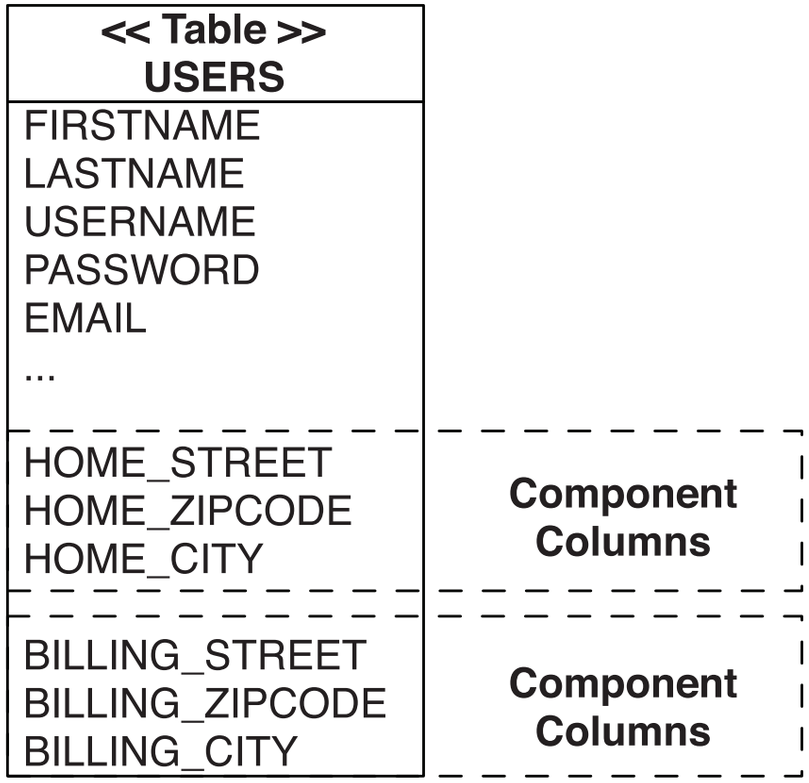
Entities vs. values types



- Entity type
  - ★ Own DB id
  - ★ references to an entity instance: FK
  - ★ independent lifecycle
- Value type
  - ★ No DB id
  - ★ belongs to an entity instance
  - ★ persitent state embedded in the table row of the owning entity instance
  - ★ lifespan bounded to lifespan of the owning entity
  - ★ do not support shared references

Mapping components

```
@Entity
@Table(name = "USERS")
public class User {
    ...
    @Embedded
    private Address homeAddress;
    ...
}
```




```
@Embeddable
public class Address {
    @Column(name = "ADDRESS_STREET", nullable = false)
    private String street;
    @Column(name = "ADDRESS_ZIPCODE", nullable = false)
    private String zipcode;
    @Column(name = "ADDRESS_CITY", nullable = false)
    private String city;
    ...
}
```

Identity vs. equality

- OO world
  - ★ Identity (==): same references (i.e., pointer to the same memory location)
  - ★ Equality: two non identical objects have the same «value» (implemented by the equals() method)
- With persitence
  - ★ Add DB identity: same PK value

Example

```
@Entity
@Table(name="CATEGORY")
public class Category {
    private Long id;
    ...
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "CATEGORY_ID")
    public Long getId() {
        return this.id;
    }
    private void setId(Long id) {
        this.id = id;
    }
    ...
}
```

- Testing identity vs. equality
  - ★ a==b
  - ★ (a.equals(b))
  - ★ a.getId().equals(b.getId())

Identity

- Every entity has a persistence identity
  - Uniquely identifies that entity
  - Maps to primary key in the database
- Identity can be application or database generated
- Must be defined on the ROOT of entity hierarchy or mapped superclass

Types of Identity

- Simple
  - > @Id – single field/property in entity class
  - > @GeneratedValue

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

- User defined:
  - > @EmbeddedId – single field/property in entity class

```
@EmbeddedId
private EmployeePK pk;
```

Class must be @Embeddable

  - > @IdClass – corresponds to multiple id field in entity class

```
@Entity
@IdClass(EmployeePK.class)
public class Employee {
    @Id private String empName;
    @Id private int dept;
```

Key generation

Generator name	JPA GenerationType	Options	Description
native	AUTO	–	The native identity generator picks other identity generators like identity, sequence, or hilo, depending on the capabilities of the underlying database. Use this generator to keep your mapping metadata portable to different database management systems.
identity	IDENTITY	–	This generator supports identity columns in DB2, MySQL, MS SQL Server, Sybase, and HypersonicSQL. The returned identifier is of type long, short, or int.
sequence	SEQUENCE	sequence, parameters	This generator creates a sequence in DB2, PostgreSQL, Oracle, SAP DB, or Mckoi; or a generator in InterBase is used. The returned identifier is of type long, short, or int. Use the sequence option to define a catalog name for the sequence (hibernate_sequence is the default) and parameters if you need additional settings creating a sequence to be added to the DDL.
(JPA only)	TABLE	table, catalog, schema, pkColumnName, valueColumnNam e, pkColumnValue, allocationSize	Much like Hibernate's hilo strategy, TABLE relies on a database table that holds the last-generated integer primary key value, and each generator is mapped to one row in this table. Each row has two columns: pkColumnName and valueColumnName. The pkColumn-Value assigns each row to a particular generator, and the value column holds the last retrieved primary key. The persistence provider allocates up to allocationSize integers in each turn.

Entity Inheritance: 3 possibilities:

Entity can extend

- **Concrete/Abstract entity class**
  - >Annotated with @Entity
  - >Behaves like an entity
- Mapped superclass
- Non entity class

Entity Inheritance – (Abstract) Entity Class

```
@Entity public abstract class Person {
   @Id protected Long id;
   protected String name;
   @Embedded protected Address address;
}

@Entity public class Customer extends Person {
   @Transient protected int orderCount;

   @OneToMany
   protected Set<Order> orders = new HashSet();
}

@Entity public class Employee extends Person {
   @ManyToOne
   protected Department dept;
}
```

Entity Inheritance

Entity can extend:

- Concrete/Abstract entity class
- **Mapped superclass**
  - Contains common fields for all entities
    - Provides common entity state
  - Mapped superclass is **NOT** an entity
  - Annotated with @MappedSuperclass
- Non entity class

Entity Inheritance – Mapped Superclass

```
@MappedSuperclass public class Person {
   @Id protected Long id;
   protected String name;
   @Embedded protected Address address;
}

@Entity public class Customer extends Person {
   @Transient protected int orderCount;

   @OneToMany
   protected Set<Order> orders = new HashSet();
}

@Entity public class Employee extends Person {
   @ManyToOne
   protected Department dept;
}
```

Entity Inheritance

Entity can extend

- Concrete/Abstract entity class
- Mapped superclass
- **Non entity class**
  - > State of non entity super class is not persisted
  - > Can **not** contain O/R mapping annotations

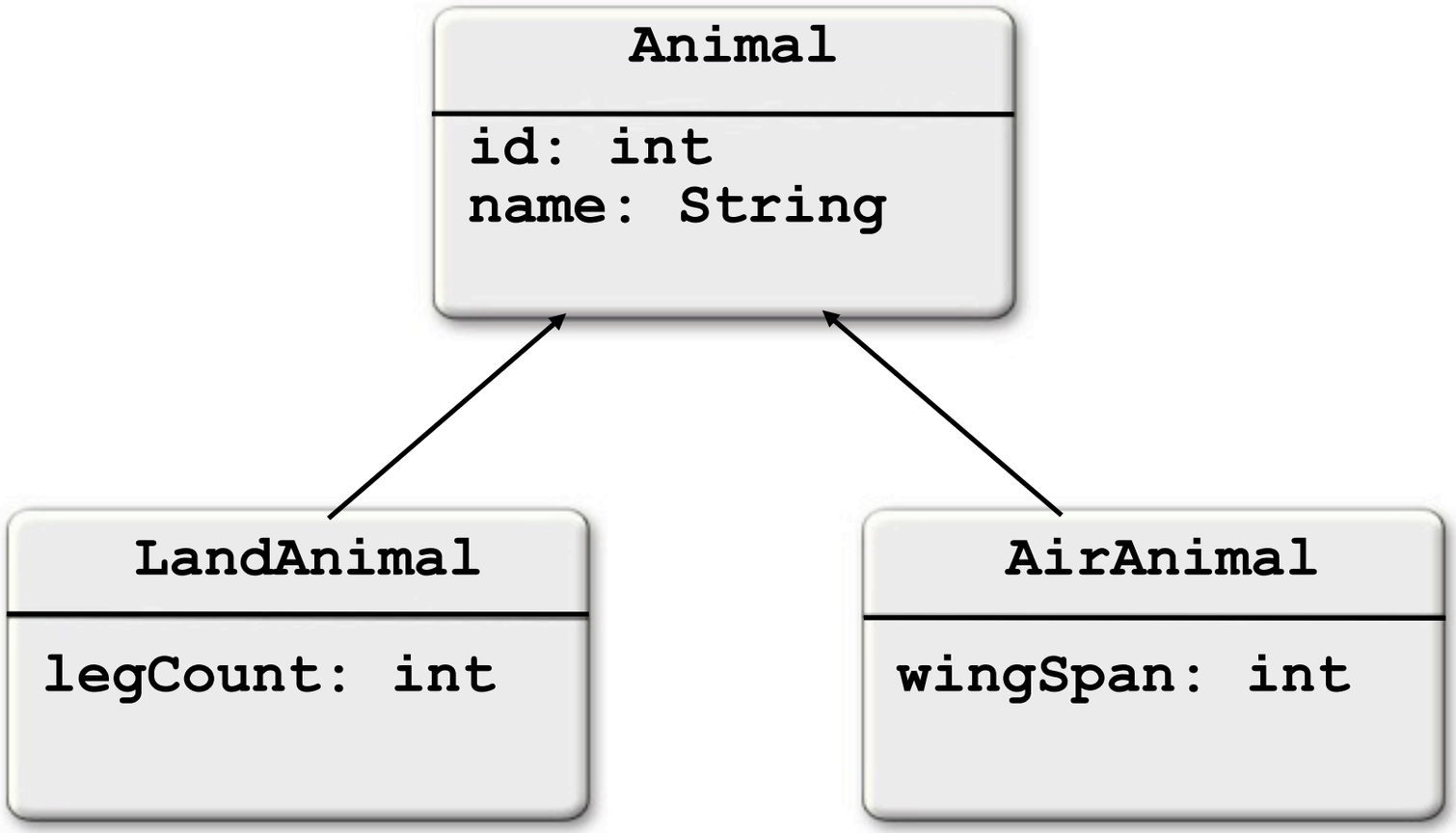
Entity Inheritance – Non Entity Class

```
public class Person {
   protected String name;

}

@Entity public class Customer extends Person {
   //Inherits name but not persisted
   @Id public int id;
   @Transient protected int orderCount;

   @OneToMany
   protected Set<Order> orders = new HashSet();
}

@Entity public class Employee extends Person {
   //Inherits name but not persisted
   @Id public int id;
   @ManyToOne
   protected Department dept;
}
```

Mapping Entity Inheritance to DB Table

- Inheritance mapping strategies:
  - Single table
  - Joined subclass
  - Table per class

Inheritance mapping strategies
Object Model



Inheritance mapping strategies
Data Models

Single table:

all classes stored in the **same table**

ANIMAL				
ID	DISC	NAME	LEG_CNT	WING_SPAN

Joined Subclass:

each class is stored in a **separate table**

ANIMAL	
ID	NAME

LAND_ANML	
ID	LEG_COUNT

AIR_ANML	
ID	WING_SPAN

Table per Class:

each **concrete** class is stored in a **separate table**

LAND_ANML		
ID	NAME	LEG_COUNT

AIR_ANML		
ID	NAME	WING_SPAN

Discriminator Column and Value for Single Table

```
@Entity @Table(name="ANIMAL")
@Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorColumnName(name="DISC")
@DiscriminatorValue("ANIMAL")
public class Animal {
   @Id protected int id;
   protected String name;
...
}
```

Discriminator required for single table

Specifies value stored for type

ANIMAL				
ID	DISC	NAME	LEG_CNT	WING_SPAN

```
@Entity @DiscriminatorValue("LANDANIMAL")
public class LandAnimal extends Animal {

   @Column(name="LEG_CNT") protected int legCount;
...
```

Discriminator Value for SingleTable

ID	DISC	NAME	LEG_CNT	WING_SPAN
1	LANDANIMAL	Cat	4	
2	LANDANIMAL	Dog	4	
3	AIRANIMAL	Eagle		7
4	AIRANIMAL	Dragonfly		0.6
...	...	...	...	...

Entity Relationships

- One-to-one, one-to-many, many-to-many, many-to-one, relationships among entities
  - bi-directional or uni-directional
  - Support for Collection, Set, List and Map
- Need to specify **owning** side in relationships
  - Owning side table has the foreign key
  - OneToOne relationship – the side with the **foreign key**
  - OneToMany, ManyToOne – **many** side

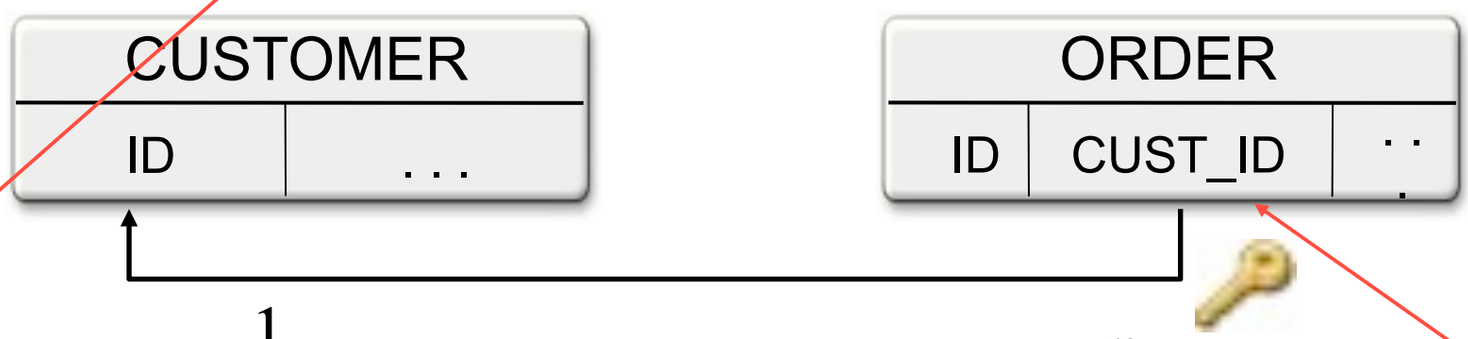
Relationship Mappings – OneToMany

Bidirectional

```
@Entity(access=FIELD)
 public class Customer {

   @Id
   int id;
   ...
   @OneToMany(mappedBy="cust")

  Set<Order> orders;
 }
```

```
@Entity(access=FIELD)
public class Order {

   @Id
   int id;
   ...
  @ManyToOne
   Customer cust;
}
```

Inverse of Relationship



has to say where the foreign key is using mappedBy

Relationship **Owner**

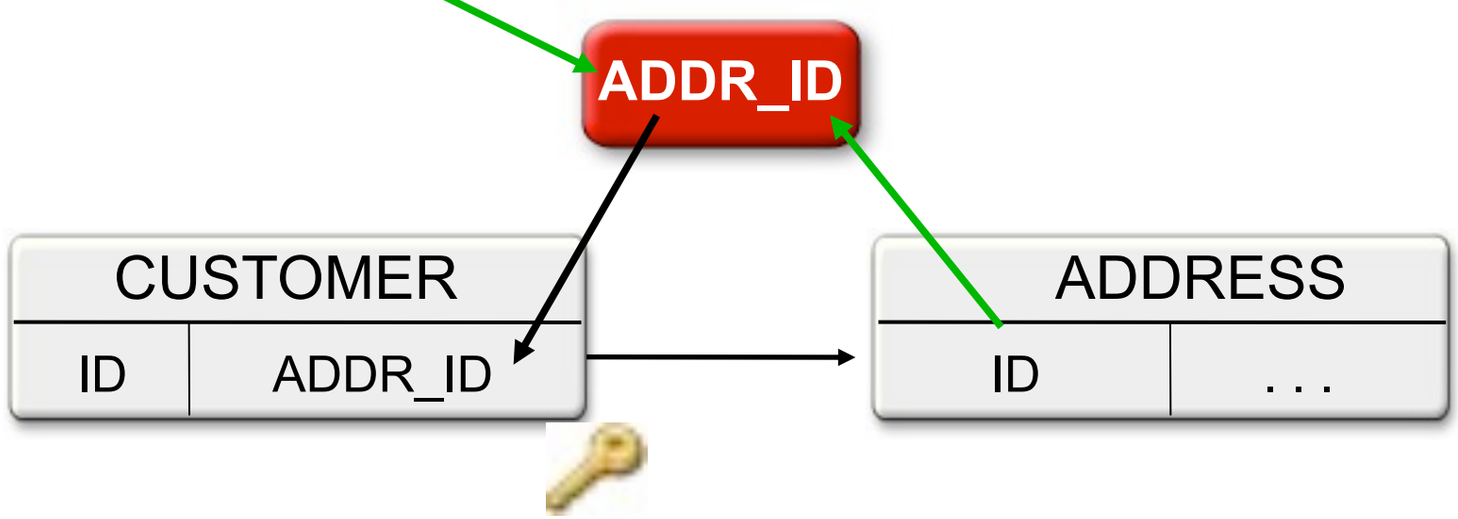
Example ManyToOne bi-directional

```
@Entity public class Order {
    @Id protected Long id;
    …
    @ManyToOne protected Customer cust;
    …
   public Customer getCustomer() {return cust;}
   public void setCustomer(Customer cust) {
      this.cust = cust;
   }

}

@Entity public class Custom
    @Id protected Long id;
    …
    @OneToMany(mappedBy="cust")
    protected Set<Order> orders = new HashSet();
    …
public void addOrder(Order order) {
    this.orders.add(order);
    order.setCustomer(this);}

}
```

Order is the owner
must call order.setCustomer(cust) whenever order added to customer.

Relationship Mappings – ManyToOne

```
@Entity(access=FIELD)
public class Customer {

    @Id
    int id;

    @ManyToOne
    Address addr;

}
```



Automatically creates a ADDR_ID field for mapping. Can be overridden via @ManyToOne annotation

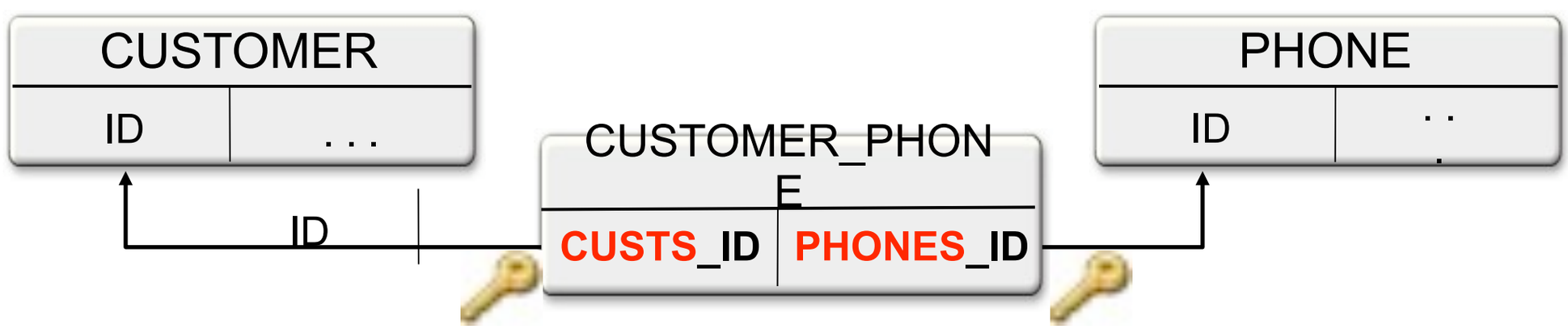
Relationship Mappings – ManyToMany

owner
of Relationship

```
@Entity(access=FIELD)
 public class Customer {

   @Id
   int id;
   ...
   @ManyToMany
   Collection<Phone> phones;
 }
```

Inverse
of Relationship

```
@Entity(access=FIELD)
 public class Phone {
   @Id
   int id;
   ...
@ManyToMany(mappedBy="phones")
  Collection<Customer> custs;
 }
```



Join table name is made up of the 2 entities. Field name is the name of the entity plus the name of the PK field

Relationship Mappings – ManyToMany

owner
of Relationship

```
@Entity(access=FIELD)
public class Customer {
   ...
   @ManyToMany
   @JoinTable(table=@Table(name="CUST_PHONE"),
      joinColumns=@JoinColumn(name="CUST_ID"),
      inverseJoinColumns=@JoinColumn(name="PHONES_ID"))
   Collection<Phone> phones;
}
```

Can override the default column names

Example ManyToMany bi-directional

```
@Entity public class Phone {
    @Id protected Long id;
    …
    @ManyToMany(mappedBy="phones")
   protected Set<Customer> customers = new HashSet();
    …


}
@Entity public class Customer {
    @Id protected Long id;
    …
    @ManyToMany
    protected Set<Phone> phones = new HashSet();
    …
    private Set<Phone> getPhones() {return phones;}

public void addPhone(Phone phone) {
    this.getPhones().add(phone);
}
public void removePhone(Phone phone) {
    this.getPhones().remove(phone);
}

}
```

customer is the **owning** side: must add/ remove the phone from the customer's phones property.

Using orm.xml for Mapping

- Can put some or all the mapping metadata in XML mapping files
  - orm.xml located in META-INF directory of JAR
  - Can use orm.xml to specify entity mappings
  - Annotations not required
- At runtime, orm.xml **override** annotations

Example of orm.xml File

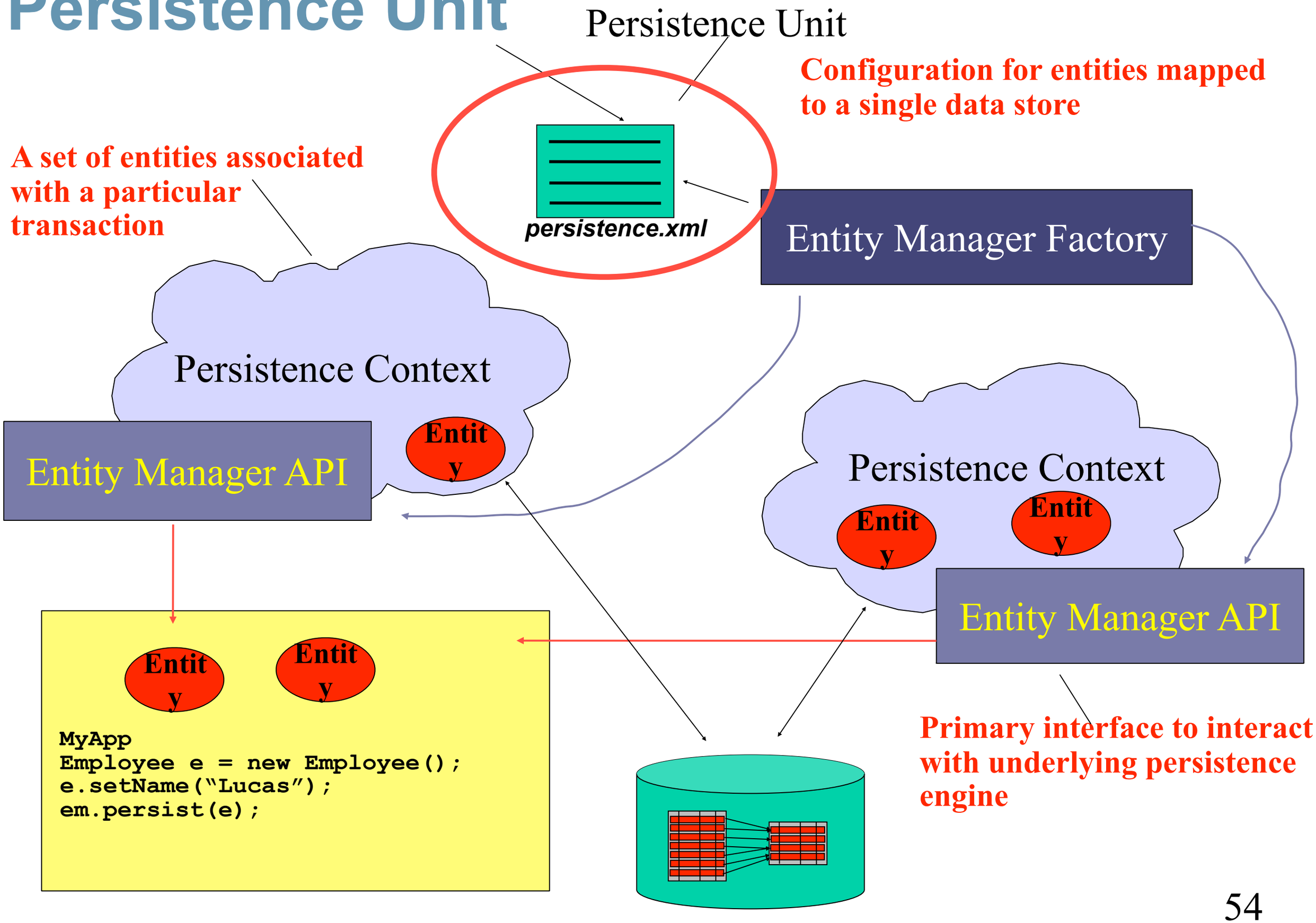
Overriding the default annotations in source

```
<entity-mappings>
   <entity class="Customer">
      <id name="id">
         <generated-value/>
      </id>
      <basic name="c_rating">
         <column name="ratings"/>
      </basic>
      ...
      <one-to-many name="orders" mapped-by="cust"/>
   </entity>
   ...
</entity-mappings>
```

Persistence – Key Concepts

- **Persistence unit**
- Entity manager
- Persistence context
- Transactions

Persistence Unit



Persistence Unit

- Persistence Unit
  - > **Configuration** to **map Entity** classes in an application to a relational **database**
- persistence.xml defines one or more persistence units
  - > the **JAR** file that contains persistence.xml will be **scanned** for any classes annotated with @Entity

```
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="OrderPU" transaction-type="JTA">
    <provider>org.hibernate.HibernatePersistence</provider>
    <jta-data-source>jdbc/order</jta-data-source>
    <jar-file>../lib/order.jar</jar-file>
    <class>com.acme.Order</class>
  </persistence-unit>
</persistence>
```

Optional to define entity classes

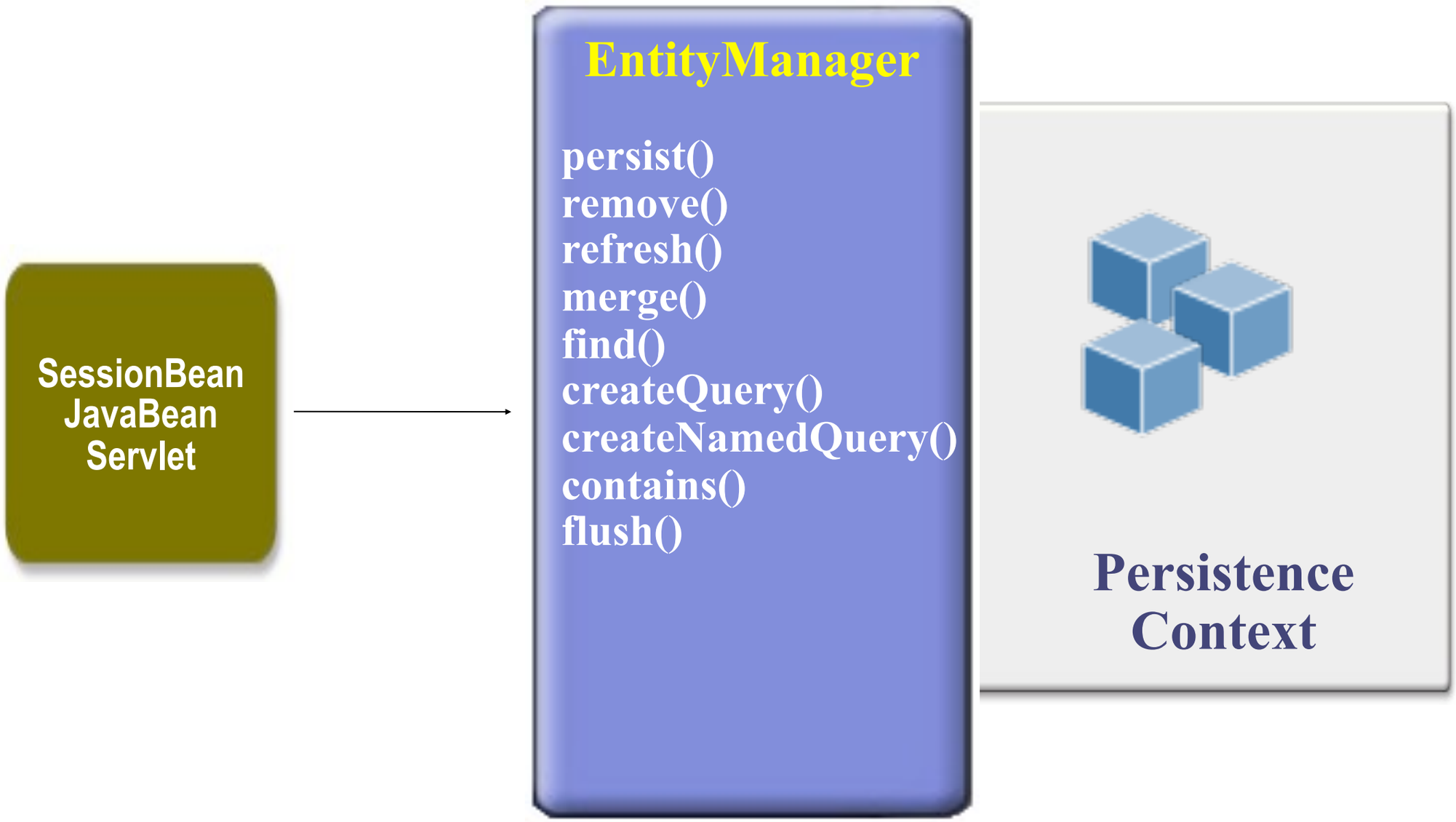
Persistence – Key Concepts

- Persistence unit
- **Entity manager**
  - Primary interface to interact with underlying persistence engine
- **Persistence context**
  - A set of entities managed by Entity Manager
- Transactions

Persistence Context and EntityManager

- EntityManager
  - > API to manage the entity instance lifecycle
  - > Allows your program to interact with underlying persistence engine
  - > Provides the following functionalities
    - >Lifecycle operations – **persist(), remove(), refresh(), merge()**
    - >Finder – find(), getReference()
    - >Factory for query objects – createNamedQuery(), createQuery(), createNativeQuery()
    - >Managing persistence context – flush(), clear(), close(), getTransaction(), ...
- Persistence Context
  - > Set of managed entities, belonging to a single persistence unit

Persistence Context and EntityManager



EntityManager Example

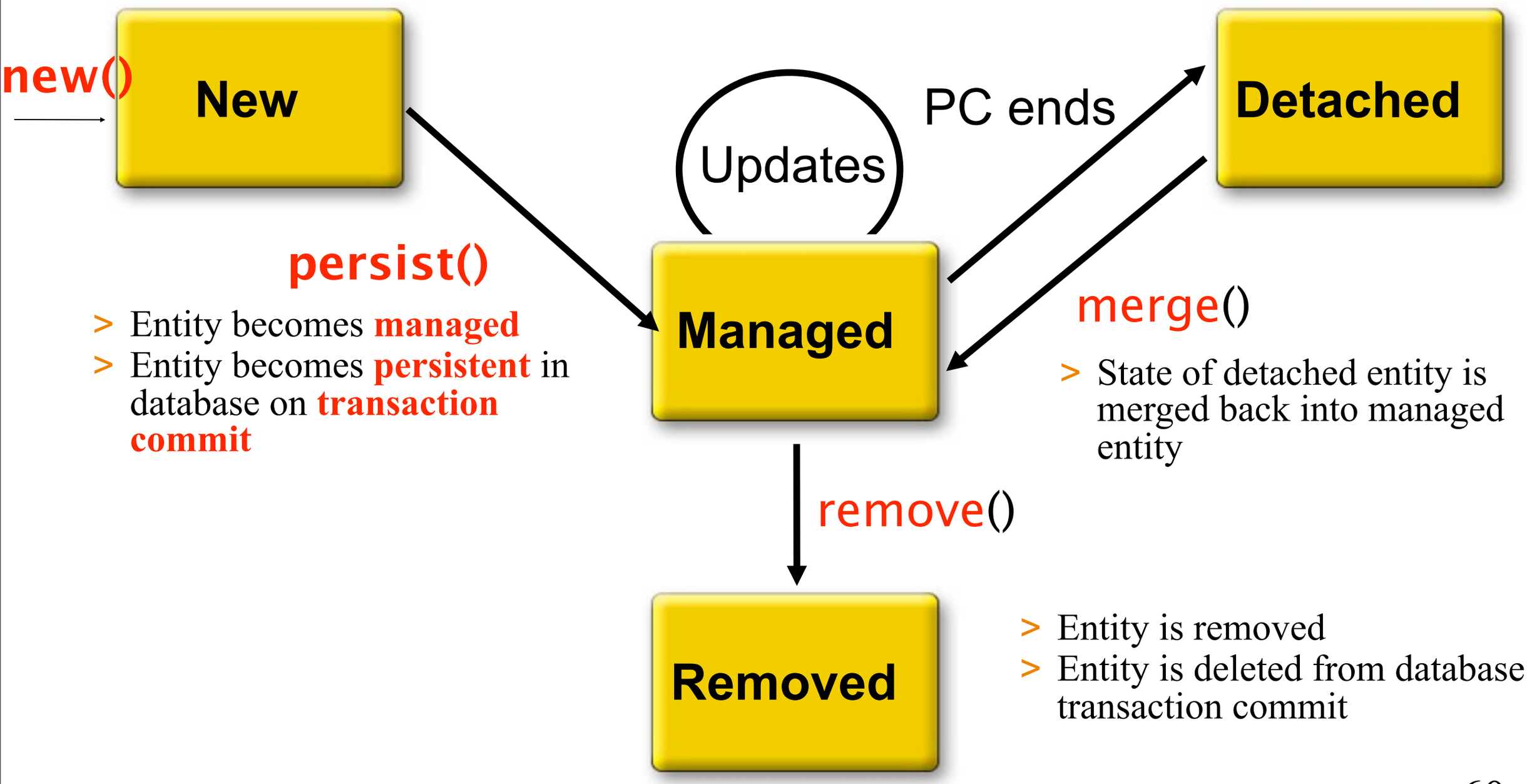
Dependency injection

```
@Stateless public ShoppingCartBean
   implements ShoppingCart {

   @PersistenceContext EntityManager entityManager;

   public OrderLine createOrderLine(Product product
         , Order order) {

      OrderLine orderLine = new OrderLine(order, product);
      entityManager.persist(orderLine);
      return (orderLine);
   }

   public OrderLine updateOrderLine(OrderLine orderLine) {
      return (entityManager.merge(orderLine));
   }
}
```

Entity Lifecycle State Diagram

> New entity instance is created
> Entity is **not** yet **managed** or **persistent**

no longer associated with persistence context



> Entity becomes **managed**
> Entity becomes **persistent** in database on **transaction commit**

> State of detached entity is merged back into managed entity

> Entity is removed
> Entity is deleted from database on transaction commit

Entity Lifecycle Illustrated – The Code

```
@Stateless public ShoppingCartBean
   implements ShoppingCart {

   @PersistenceContext EntityManager entityManager;

   public OrderLine createOrderLine(Product product
         , Order order) {
      OrderLine orderLine = new OrderLine(order, product);
      entityManager.persist(orderLine);
      return (orderLine);
   }

   public OrderLine updateOrderLine(OrderLine orderLine) {
       OrderLine newOL=entityManager.merge(orderLine)
      return (newOL));
   }
}
```

Entity Lifecycle Illustrated – The Code

```
@Stateless public ShoppingCartBean
   implements ShoppingCart {

   @PersistenceContext EntityManager entityManager;

   public OrderLine createOrderLine(Product product
         , Order order) {
      OrderLine orderLine = new OrderLine(order, product);
      entityManager.persist(orderLine);
      return (orderLine);
   }

   public OrderLine updateOrderLine(OrderLine orderLine) {
      OrderLine newOL=entityManager.merge(orderLine)
      return (newOL));
   }
}
```

New entity

Managed entity

Detached entity

Entity Manager Persist:

- Insert a new instance of the entity into the database (when transaction commits)
- The entity instance becomes “managed” in the persistance context
- Persist operation optionally **cascades to related** objects

```
public Customer createCustomer(intid, String name) {
Customer cust= new Customer(id, name);
entityManager.persist(cust);
return cust;
}
```

Cascading Persist example: ManyToOne

```
@Entity(access=FIELD)
 public class Customer {

   @Id
   int id;
   ...

  @OneToMany(mappedBy="cust")

  Set<Order> orders;
 }
```

```
@Entity(access=FIELD)
public class Order {

   @Id
   int id;
   ...

  @ManyToOne
   Customer cust;
}
```

Persist OneToMany bi-directional
Not Cascading

```
@Stateless public class OrderManagementBean
   implements OrderManagement {
   …
   @PersistenceContext EntityManager em;
   …
   public Order addNewOrder(Long id,Product product){

      Customer cust = em.find(Customer.class,
         id);
      Order order = new Order(product);
      customer.getOrders().add(order);
       order.setCustomer(cust);
      em.persist(order);
      return order;
   }
}
```

em.persist to persist the order

Cascading Persist

```
@Entity
public class Customer {
   @Id protected Long id;
   …
   @OneToMany(cascade=PERSIST)
   protected Set<Order> orders = new HashSet();
}
```

```
…
public Order addNewOrder(Customer customer,
Product product) {
   Order order = new Order(product);
   customer.getOrders().add(order);
   order.setCustomer(cust);
   return order;
}
```

cascade=Persist Order persisted automatically, when added to Customer

Entity Manager Find, Remove:

- Find
  - Get a managed entity instance with a given persistent **identity**
  - Return null if not found
- Remove
  - Delete entity with the given persistent identity from the database (deleted when transaction commits)
  - Optionally cascades to related objects

```
public void removeCustomer(Long custId) {
Customer cust=
entityManager.find(Customer.class, custId);
entityManager.remove(cust);
}
```

Remove

```
@Entity
public class Order {
   @Id protected Long orderId;
   …
   @OneToMany(cascade={PERSIST,REMOVE})
   protected Set<LineItem> lineItems = new
HashSet();
}

…
@PersistenceContext EntityManager em;
…
public void deleteOrder(Long orderId) {
   Order order = em.find(Order.class, orderId);
   em.remove(order);
}
```

When order is removed, the lineItems are removed with it

Merge:

- State of detached entity gets merged into a managed copy of entity
- Managed entity is returned
- Merge operation optionally cascades to related objects

```
public Customer storeUpdatedCustomer(Customer cust) {
Customer customer =entityManager.merge(cust);
return customer;
}
```

Managed

Not managed

Merge

```
@Entity
public class Order {
   @Id protected Long orderId;
   …
   @OneToMany(cascade={PERSIST, REMOVE, MERGE})
   protected Set<LineItem> lineItems = new
HashSet();
}


…
@PersistenceContext EntityManager em;
…
public Order updateOrder(Order changedOrder) {
   return em.merge(changedOrder);
}
```

When order is merged, the lineItems are merged with it

Detached Objects as DTOs

Detached after return

Returned to caller as Detached

Detached

```
public OrderLine updateOrderLine(
    OrderLine orderLine) {
    OrderLine newOL=em.merge(orderLine)
    return(newOL);
}
```

Managed

- Entity beans may be passed by value as detached objects
- Must implement Serializable interface if detached object is to be sent across the wire