

# Объектно-реляционное преобразование в технологии EJB

# Объектно-реляционное преобразование в технологии EJB

- **Entity Beans** (Компоненты-сущности)
  - > Особый тип EJB-компонентов, предназначенных для организации компонентного интерфейса доступа к данным из того или иного источника
- **Java Persistence API**
  - > Набор прикладных интерфейсов языка Java для объектно-реляционного преобразования на платформах **Java SE** и **Java EE**
  - > **Entity** (Сущность) – *легкий* устойчивый объект предметной области

# Источник данных

- Entity Beans
  - > Container-Managed Persistence
    - > РБД
  - > Хранилище данных любой природы с помощью JCA-адаптера
    - > ООБД
    - > файловая система
    - > XML-хранилище
    - > Корпоративная информационная система (КИС)
- Java Persistence API
  - > Только РБД

# Работа с данными

- Entity Beans и JPA
  - > Работа с данными как с **системой объектов**
  - > **Состояние** объектов (значения полей) отражает состояние данных в соответствующем источнике
  - > Клиент изменяет состояние объекта, вызывая его **методы**
  - > **Изменения** состояния объектов **синхронизируются** с источником данных **автоматически**
    - > под управлением контейнера или
    - > по требованию приложения.

# Отношения между данными

- Entity Beans
  - > Поддерживается моделирование отношений
  - > Контейнер **обеспечивает** соблюдение правил целостности
- Java Persistence API
  - > Поддерживается моделирование отношений
  - > Контейнер **не обеспечивает** соблюдение правил целостности

# Запросы к данным

- Entity Beans
  - > Источники данных различных типов => язык запросов должен работать на компонентном уровне и не зависеть от типа источника данных
  - > Язык запросов **EJB QL** оперирует компонентами-сущностями
    - > Аналог SELECT в SQL
- Java Persistence API
  - > Язык запросов **JP QL** оперирует сущностями
    - > Поддерживает операторы SELECT, UPDATE, DELETE

# Принципы разработки и использования

- Entity Beans
  - > Клиент работает с компонентом через **представление**
  - > Логика компонента реализуется классом компонента
  - > ЖЦ, назначение и детали реализации сущности существенно отличаются от SB
- Java Persistence API
  - > Клиент работает с экземпляром сущности **напрямую**
  - > Логика реализуется классом сущности
  - > Для управления ЖЦ используется не домашний интерфейс, а **менеджер сущностей**

# Сущности



# Пример

```
@Entity public class Employee {  
    @Id private int id;  
    private String firstName;  
    private String lastName;  
    @ManyToOne(fetch=LAZY)  
    private Department dept;  
    ...  
}  
  
@Entity public class Department {  
    @Id private int id;  
    private String name;  
    @OneToMany(mappedBy = "dept", fetch=LAZY)  
    private Collection<Employee> emps = new ...;  
    ...  
}
```

# Требования к классу сущности

- 1) Должен быть отмечен аннотацией **@Entity** или указан как сущность в XML-дескрипторе
- 2) Открытый или защищенный конструктор без параметров (могут быть и другие конструкторы)
- 3) Должен быть классом верхнего уровня (top-level class), не **перечислимым типом** или **интерфейсом**
- 4) Класс сущности, его методы и постоянные поля не должны быть **final**
- 5) Если экземпляр сущности должен передаваться как отсоединенный объект (например, через удаленный интерфейс), класс сущности должен реализовывать интерфейс **Serializable**

# Постоянные поля и свойства

- Постоянное состояние экземпляра сущности – значения полей экземпляра класса сущности
  - > Могут соответствовать свойствам JavaBeans
- Доступ к полям сущности напрямую должен вестись исключительно из методов этой сущности
- **Клиент** получает доступ к состоянию экземпляра сущности только через **get-/set-методы** или другие бизнес-методы
- Поля класса сущности не должны быть открытыми

# Постоянные поля и свойства

- **Провайдер персистентности** осуществляет доступ к постоянному состоянию экземпляра сущности:
    - > через свойства (с помощью get-/set-методов в стиле JavaBeans)
      - > ОРП-аннотации для **get**-методов
      - > Дополнительная бизнес-логика в get-/set-методах, но
      - > Порядок вызова методов не определен
- ```
@Entity public class Employee {  
    private int id;  
    @Id public int getId() { return id; }  
    ...  
}
```
- > через поля – ОРП-аннотации для полей
  - > сочетать способы **нельзя**

# Постоянные поля и свойства

- Все постоянное, кроме временного – `@Transient`
- Допустимы следующие типы постоянных полей и свойств:
  - > примитивные типы и `java.lang.String`;
  - > сериализуемые типы (в том числе оболочки примитивных типов, `java.math.BigInteger`, `java.math.BigDecimal`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`);
  - > массивы `byte[]`, `Byte[]`, `char[]` и `Character[]`;
  - > перечислимые типы;
  - > типы сущностей и коллекции типов сущностей;
  - > вложимые классы

# Постоянные поля и свойства

- Допустимые типы коллекций: `java.util.Collection`, `java.util.Set`, `java.util.List`, `java.util.Map`, а также их параметризованные варианты
  - > Конкретный тип коллекции можно использовать только при инициализации поля или свойства перед сохранением сущности.

```
@Entity public class Department {  
    @Id private int id;  
    private String name;  
    @OneToMany(mappedBy = "dept", fetch=LAZY)  
    private Collection<Employee> emps =  
        new ArrayList<Employee>();  
    ...  
}
```

# Создание экземпляра сущности

```
Customer cust = new Customer();  
// экземпляр сущности cust НЕ является устойчивым  
  
EntityManager em = ...;  
em.persist(cust);  
// теперь экземпляр сущности cust является устойчивым
```

# Первичные ключи и идентичность сущности

- У каждой сущности должен быть **первичный ключ**
  - > Определяется **1** раз в иерархии классов сущностей
- **Простой первичный ключ**
  - > Единственное постоянное поле (свойство) **@Id**
- **Составной первичный ключ**
  - > Класс первичного ключа – **@Embeddable** либо **@IdClass**
  - > Одно постоянное поле (свойство) **@EmbeddedId** либо
  - > Несколько постоянных полей (свойств) **@Id**



# Первичные ключи и идентичность сущности

- Допустимые типы полей (свойств) первичного ключа:
  - > 1) примитивный тип
  - > 2) класс-оболочка примитивного типа
  - > 3) `java.lang.String`
  - > 4) `@Temporal(DATE)` `java.util.Date`, `java.sql.Date`
- Способ доступа к классу первичного ключа должен совпадать со способом доступа к классу сущности
- Приложение **НЕ** должно **изменять** значение первичного ключа после сохранения сущности

# Первичные ключи и идентичность сущности

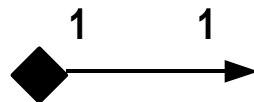
- Требования к классу первичного ключа:
  - > Класс первичного ключа должен быть открытым и должен иметь открытый конструктор без параметров
  - > Класс первичного ключа должен быть сериализуемым.
  - > В классе первичного ключа должны быть определены методы `equals()` и `hashCode()`
    - > Реализация должна обеспечивать те же результаты, что операция `=` в SQL

# Автоматическая генерация значения первичного ключа

- Только для целочисленных типов
- `@Id`  
`@GeneratedValue(strategy=GenerationType.AUTO)`  
`private long id;`
  - > Способы генерации:
    - > `AUTO` (по умолчанию) – выбор наиболее подходящего способа для конкретной СУБД
    - > `SEQUENCE` – уникальные значения берутся из последовательности значений
    - > `IDENTITY` – уникальное значение формируется СУБД в столбце особого типа при вставке строки в таблицу (например, автоинкрементное поле в MySQL)
    - > `TABLE` – дополнительная таблица, содержащая пары «имя сущности – последнее значение первичного ключа»

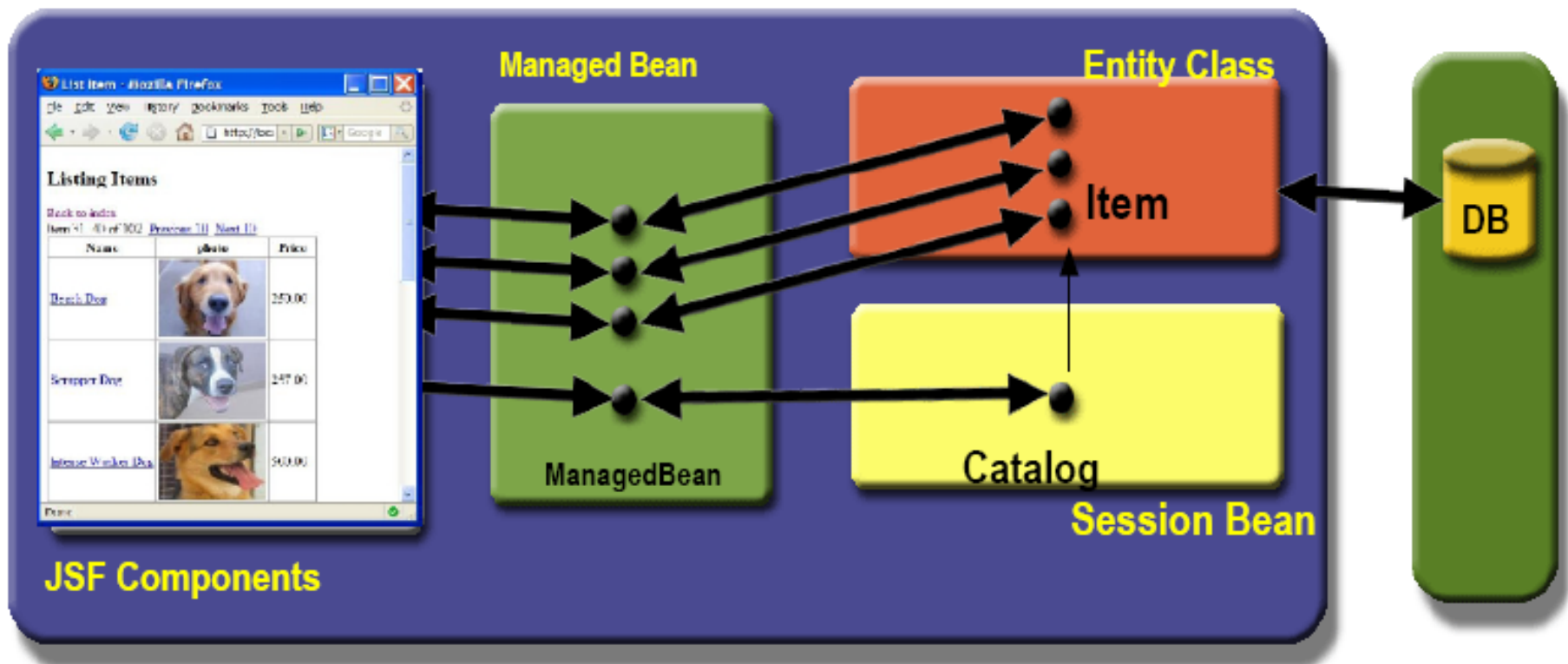
# Вложимые классы

- Для представления состояния сущности можно использовать другие классы, называемые *вложимыми*
- Экземпляры вложимых классов **не имеют** собственной **идентичности** и существуют только как вложенные объекты той сущности, к которой они принадлежат
  - > Как правило, отображаются на таблицы вместе с владеющей ими сущностью
- Вложимый класс должен отвечать требованиям к классу сущности, но **@Entity**, а **@Embeddable**
- Поддерживается только **один** уровень вложенности



# Управление сущностями

# Структура Java EE-приложения, использующего JPA



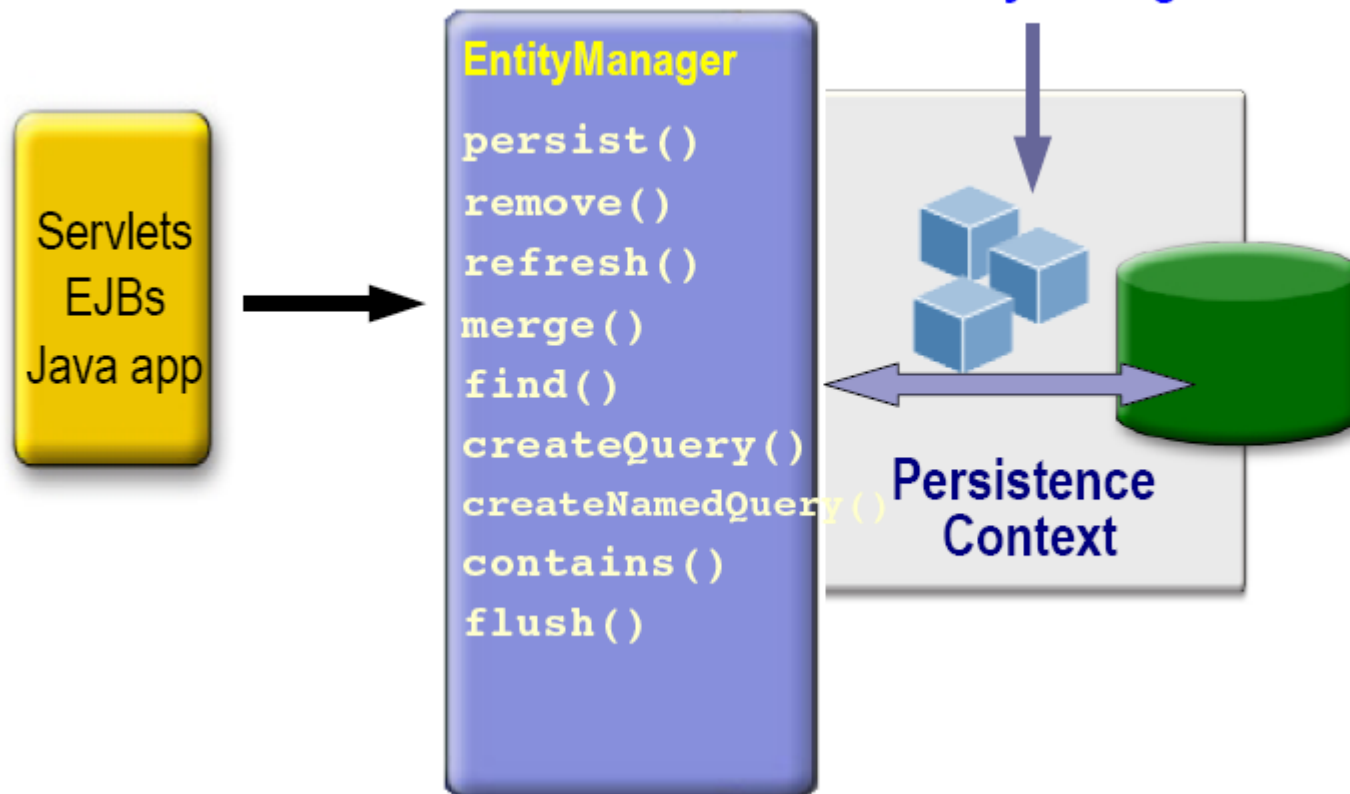
# Модуль персистентности (persistence unit)

- Средство упаковки и установки классов сущностей
  - > Конфигурационная информация (в т.ч. источник данных, тип провайдера персистентности, способ управления транзакциями) – **persistence.xml**
  - > Множество классов сущностей
  - > Отображение классов сущностей на РБД (аннотации или XML-дескриптор)
- Модуль персистентности **(N)** – **(1)** РБД
  - > Область действия запросов и связей между сущностями
  - > РБД сложной структуры → композиция PU

# Контекст персистентности (Persistence Context)

- Множество экземпляров сущностей **одного** модуля *персистентности*, которые были прочитаны из БД и/или которые нужно сохранить в БД

Множество сущностей,  
управляемых  
Entity Manager

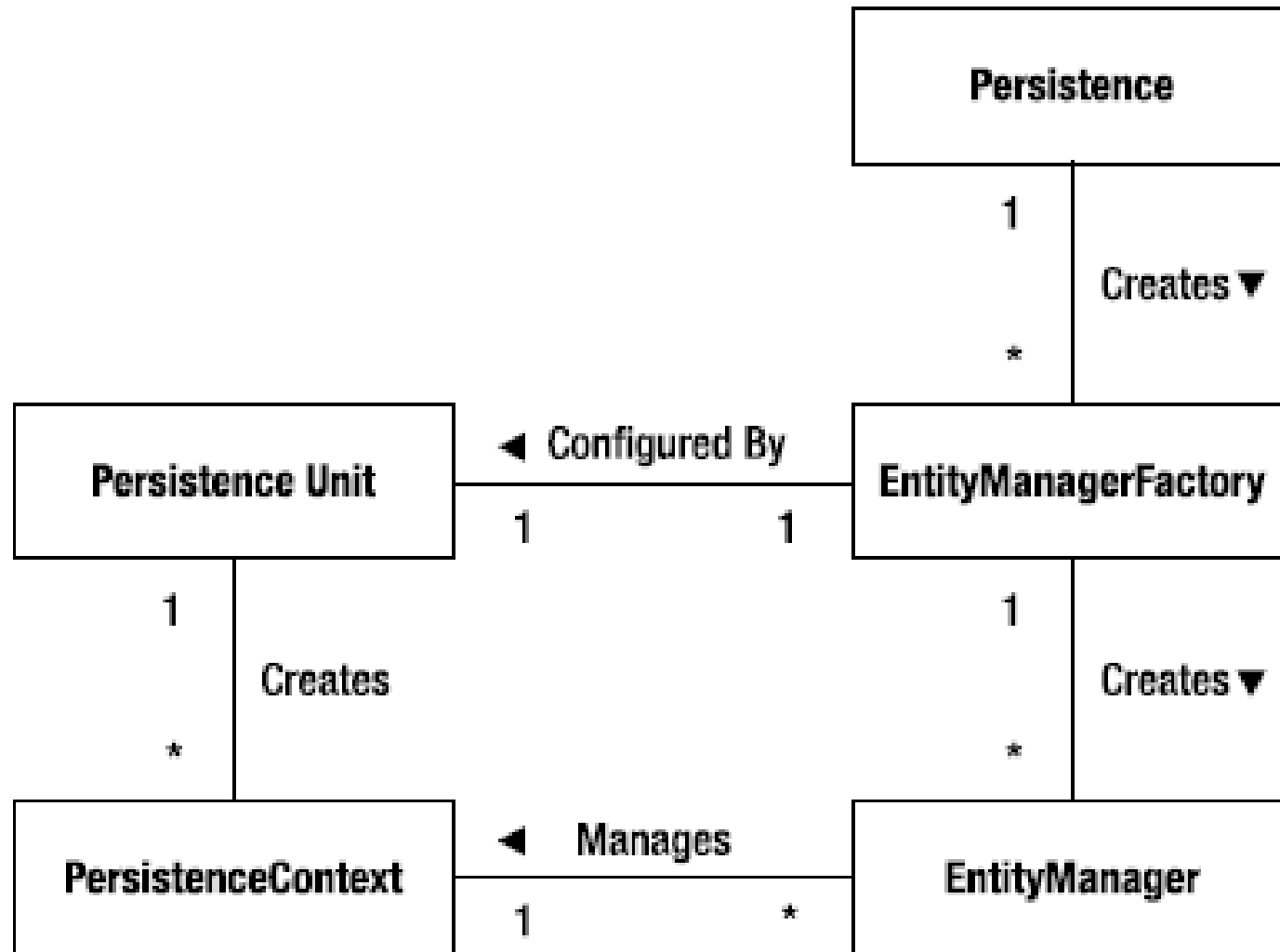




# Менеджер сущностей (Entity Manager)

- Объект типа `javax.persistence.EntityManager`
- Управляет жизненным циклом экземпляров сущностей, находящихся в конкретном контексте персистентности
- Менеджер сущностей **(N)** → **(1)** Контекст персистентности
- Группы методов:
  - > управление жизненным циклом экземпляров сущностей
  - > поиск экземпляра сущности по первичному ключу;
  - > создание объектов типа Query, предназначенных для выполнения запросов;
  - > управление контекстом персистентности (`flush()`, `clear()`, `close()`, `getTransaction()`, ...)

# Концепции JPA



# Пример работы с менеджером сущностей 1

`@Stateless`

```
public class Catalog implements CatalogService {
```

```
    @PersistenceContext(unitName="PetCatalogPu")  
    EntityManager em;
```

```
    @TransactionAttribute(NOT_SUPPORTED)
```

```
    public List<Item> getItems(int firstItem,  
                               int batchSize) {  
        Query q = em.createQuery("select i from Item i");  
        q.setMaxResults(batchSize);  
        q.setFirstResult(firstItem);  
        List<Item> items= q.getResultList();  
        return items;  
    }  
}
```

# Пример работы с менеджером сущностей 2

- > Любому постоянному идентификатору сущности соответствует единственный экземпляр сущности

```
@Stateless public ShoppingCartBean implements  
ShoppingCart {  
    @PersistenceContext EntityManager entityManager;  
  
    public OrderLine createOrderLine(Product product,  
        Order order) {  
        OrderLine orderLine =  
            new OrderLine(order, product);  
        entityManager.persist(orderLine);  
        OrderLine orderLine2 =entityManager.find(OrderLine,  
            orderLine.getId());  
        (orderLine == orderLine2); // TRUE  
        return (orderLine);  
    }  
}
```

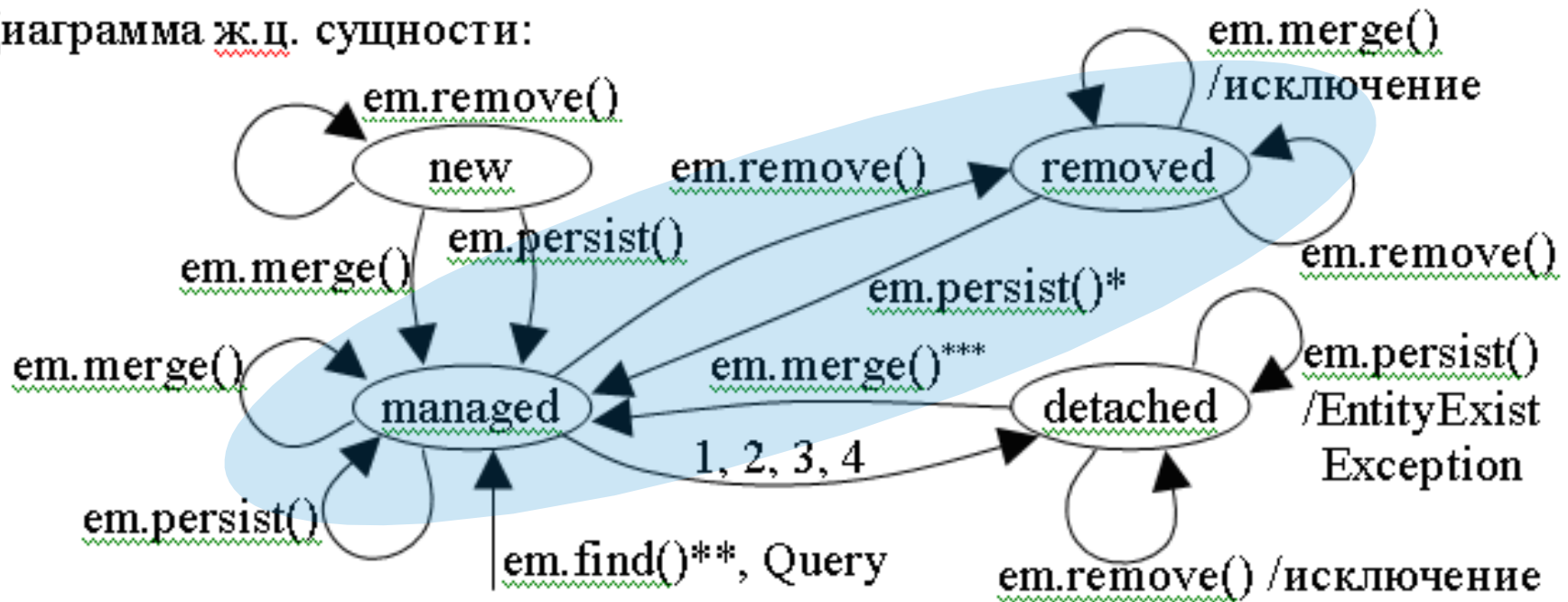
# Контексты персистентности по типу управления

- Управляемый контейнером
  - > предназначен для Java EE
  - > передается вместе с JTA-транзакцией
  - > получается через инъекцию зависимости (`@PersistenceContext`) или поиск в окружении;
  - > может иметь как расширенное время жизни (в SFSB), так и время жизни в контексте транзакции
- Управляемый приложением
  - > используется в средах Java SE и Java EE
  - > только локальные транзакции
  - > создается с помощью фабрики
    - > `Persistence.createEntityManagerFactory()`
  - > имеет расширенное время жизни

# Жизненный цикл экземпляра сущности

- Новый, управляемый, отсоединенный, удаленный

Диаграмма ж.ц. сущности:



# Жизненный цикл экземпляра сущности

```
@Stateless public ShoppingCartBean
implements ShoppingCart {
```

**Новый**

```
@PersistenceContext EntityManager entityManager;
```

**Контекст  
персистентности**

```
public OrderLine createOrderLine(Product product,
    Order order) {
```

```
    OrderLine orderLine =
        new OrderLine(order, product);
```

```
    entityManager.persist(orderLine);
```

```
    return (orderLine);
```

**Управляемый**

**Отсоединенный**

# Обработчики событий ЖЦ сущности

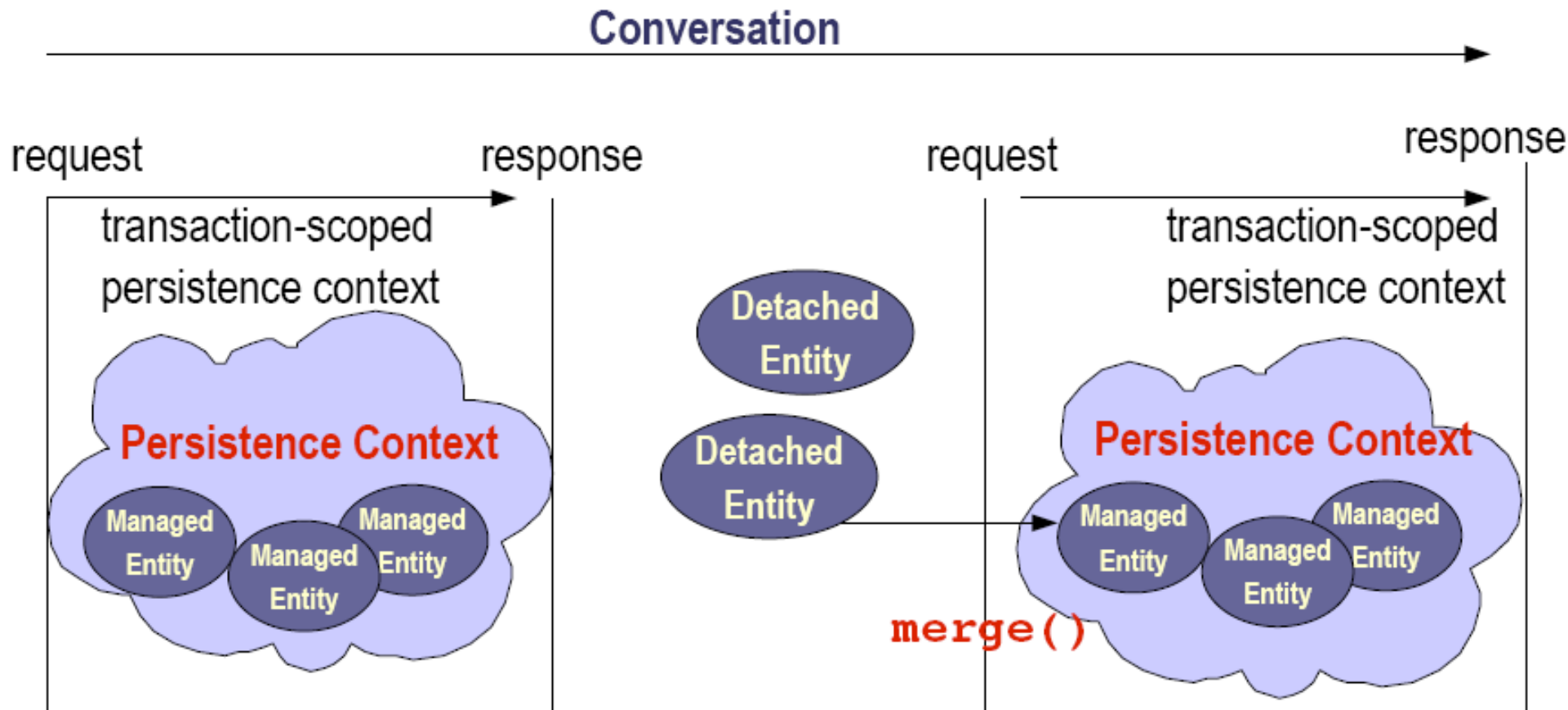
- Методы в классе сущности либо класс обработки событий (назначается `@EntityListeners`)
- События ЖЦ:
  - > `@PrePersist` / `@PostPersist` ← доступен auto ID
  - > `@PreRemove` / `@PostRemove`
    - > Post-обработчики – сразу после вызова метода либо при синхронизации контекста персистентности с БД
  - > `@PreUpdate` / `@PostUpdate`
  - > `@PostLoad` – после загрузки экземпляра сущности из БД либо после выполнения метода *refresh()*
- Проверка правильности состояния сущности перед или после синхронизации с БД
- Выброс исключения приводит к откату транзакции<sub>32</sub>



# Время жизни контекста персистентности

- Контекст существует только во время выполнения приложения (пока существует связанный с ним менеджер сущностей)
- Имеет ограниченное время жизни:
  - > в рамках транзакции или
  - > расширенное, охватывающее несколько транзакций
- По окончании времени жизни контекст очищается

# Контекст персистентности со временем жизни в рамках транзакции



# Контекст персистентности со временем жизни в рамках транзакции

```

@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);
    }

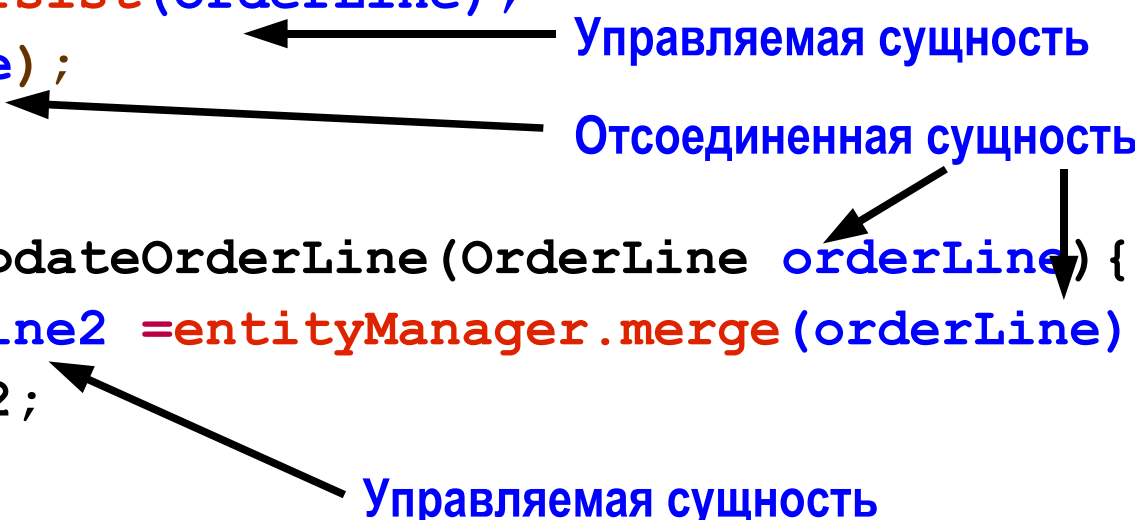
    public OrderLine updateOrderLine(OrderLine orderLine) {
        OrderLine orderLine2 = entityManager.merge(orderLine);
        return orderLine2;
    }
}

```

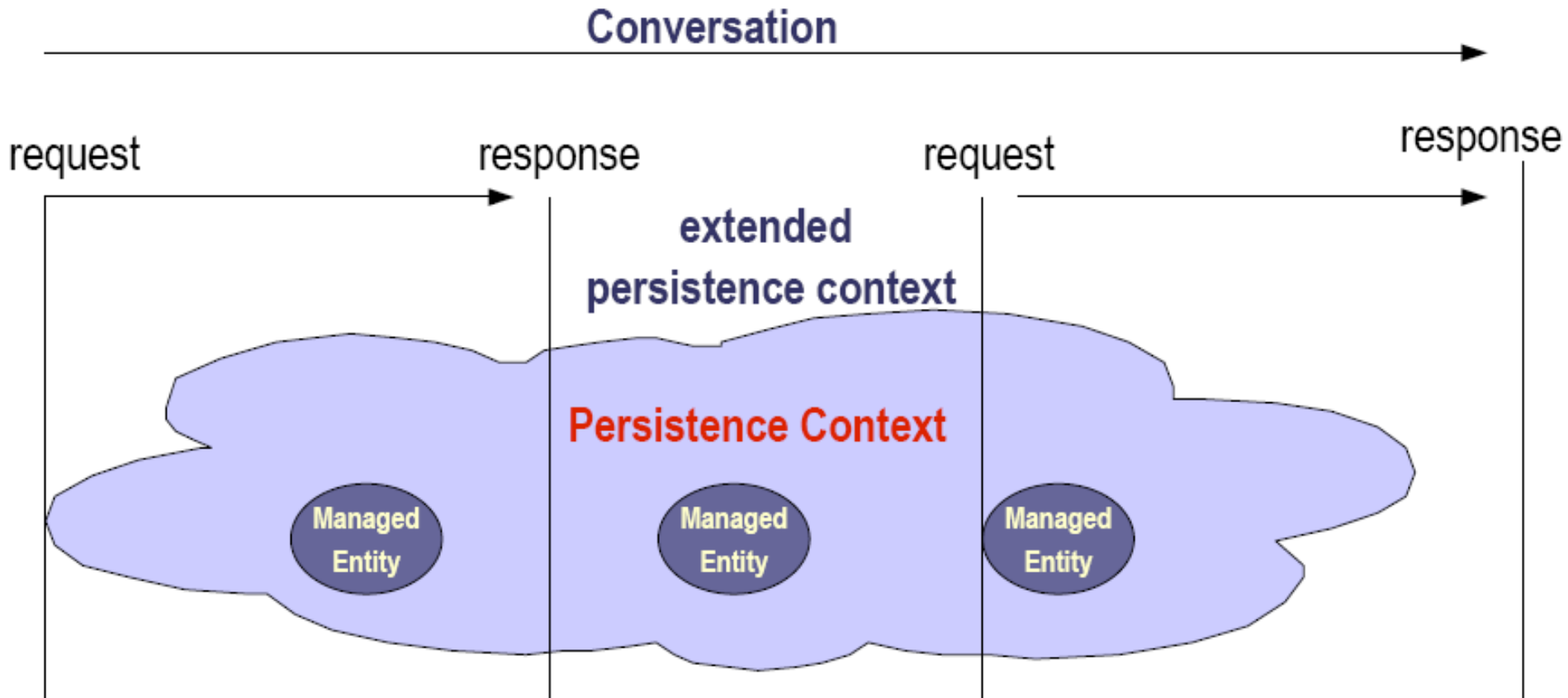
Управляемая сущность

Отсоединенная сущность

Управляемая сущность



# Контекст персистентности с расширенным временем жизни



# Контекст персистентности с расширенным временем жизни

```
@Stateful public class OrderMgr {  
    //Нужен расширенный контекст персистентности  
    @PersistenceContext(type=PersistenceContextType.EXTENDED)  
    EntityManager em;  
    //Кэшированный заказ  
    private Order order;  
    //Создать и кэшировать заказ  
    public void createOrder(String itemId) {  
        //Заказ остается управляемым, пока существует компонент  
        Order order = new Order(cust);  
        em.persist(order);  
    }  
    public void addLineItem(OrderLineItem li) {  
        order.lineItems.add(li);  
    }  
}
```