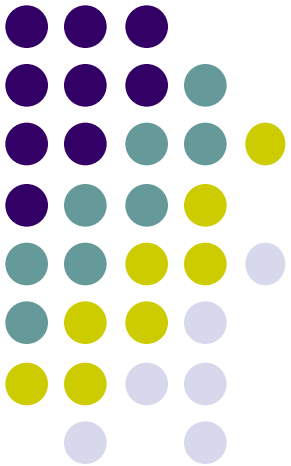
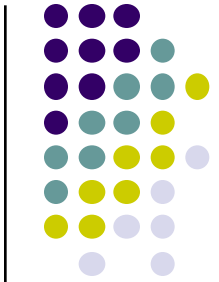
JPA

Java Persistence API

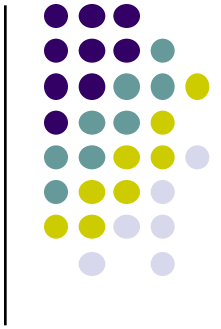


Тема лекції



- JPA Entities
- JPA Framework
- JPA Entity Manager
- PersistanceContext
- Запити в JPA
- Приклади

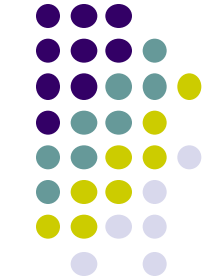
JPA



- Java Persistence API (JPA) – специфікація, яка стандартизує ORM для Java
  - Є частиною JSR-220
  - Пакет javax.persistance
  - JPA надає механізм збереження Java-об’єктів у реляційній БД

- Реалізації:
  - Hibernate, EclipseLink, TopLink та інші
  - Spring JPA йде на базі Hibernate

JPA. Вимоги до об’єктів, що зберігаються

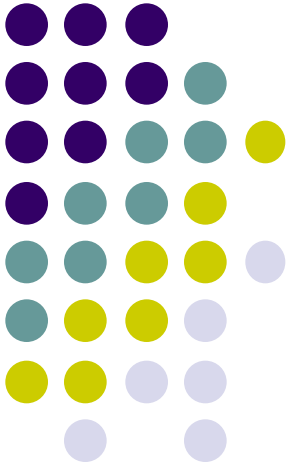


- Об’єкти, що зберігаються = persistence entity = “сутності”
- Вимоги до об’єктів, що зберігаються:
  - POJO
  - Класи верхнього рівня
  - Анотація класу **@Entity**
  - Повинно бути поле із анотацією **@Id**
  - Конструктор без аргументів (public, protected)
  - Не повинно бути модифікатора final
    - Для класу та полів, що зберігаються
  - Клас повинен реалізувати Serializable (якщо необхідна передача даних у інший адресний простір)

```
@Entity
public class AppUser {

  @Id
  private Long id;
  private String username;
  private String password;

}
```


JPA

Анотації для сутностей

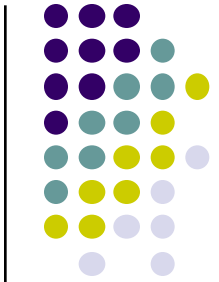


JPA-анотації для сутностей

- Анотації визначені в пакеті javax.persistence
- Анотації можуть застосовуватись:
  - До полів
  - До властивостей (для методів get)

  - Рекомендується застосовувати анотації до полів

Ідентифікатори



- У сутності повинно бути визначене поле (поля), яке відповідає первинному ключу (PK) в БД
- Типи ідентифікаторів для первинних ключів:
  - Простий (із одного поля)
  - Складений (із декількох полів)
- Типи анотацій:
  - **@Id** – простий ідентифікатор, який складається із одного поля. Найбільш розповсюджений варіант.
  - **@IdClass** – складений ідентифікатор.
  - **@EmbeddedId** – складений ідентифікатор.
- Для складених ідентифікаторів Java-класи, що реалізують PK, повинні:
  - Реалізувати Serializable
  - Перевизначати equals(), hashCode()

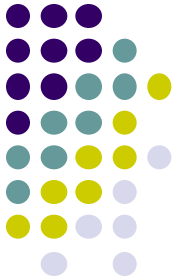
@IdClass

- Поля складеного ідентифікатора є різними полями сутності

```
@Entity
@IdClass(ArtistPK.class)
public class Artist {

   @Id
   private Long idOne;
   @Id
   private Long idTwo;
}
```

```
public class ArtistPK
    implements Serializable {

   private Long idOne;
   private Long idTwo;

   public boolean equals(Object obj);
   public int hashCode();
}
```

@EmbeddedId

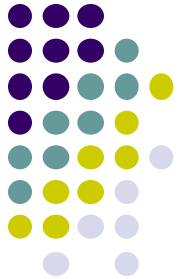


- Складений ідентифікатор подається як окреме поле сутності

```
@Entity
public class Artist {

   @EmbeddedId
   private ArtistPK key;
}
```

```
@Embedded
public class ArtistPK
    implements Serializable {

   private Long id1;
   private Long id2;

   public boolean equals(Object obj);
   public int hashCode();
}
```

@GeneratedValue

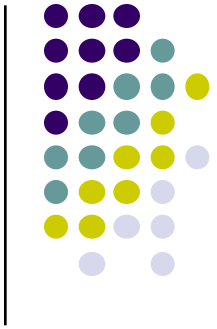


- **@GeneratedValue** підтримує генерацію значень для поля

- Типи генерації:
  - **GenerationType.AUTO**
    - автоматичний вибір однієї з трьох нижченаведених стратегій, в залеженості в БД
  - **GenerationType.IDENTITY**
    - отримання значення із колонок IDENTITY
    - MySQL, DB2, SQL Server, Sybase, Postgres
  - **GenerationType.SEQUENCE**
    - застосування «послідовностей»
    - Oracle, DB2, Postgres
  - **GenerationType.TABLE**
    - отримання значення із таблиці

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

@GeneratedValue. Приклади



- **GenerationType.IDENTITY**

```
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private long id;
    ...
}
```

- **GenerationType.SEQUENCE**

```
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="EMP_SEQ")
    @SequenceGenerator(name="EMP_SEQ", sequenceName="EMP_SEQ",
    allocationSize=100)
    private long id;
    ...
}
```

- **GenerationType.TABLE**

```
@Entity
public class Employee {
    @Id
    @TableGenerator(name="TABLE_GEN", table="SEQUENCE_TABLE",
    pkColumnName="SEQ_NAME",
        valueColumnName="SEQ_COUNT", pkColumnValue="EMP_ID")
    @GeneratedValue(strategy=GenerationType.TABLE, generator="TABLE_GEN")
    private long id;
    ...
}
```

@Table, @Column

- **@Table** – визначає відповідність між класом, об'єкти якого зберігаються, та таблицею
- **@Column** – визначає відповідність між полем, що зберігається, та колонкою

```
@Entity
@Table(name = "TBL_ARTIST")
public class Artist {

   @Id
   @Column(name = "ARTIST_ID")
   private Long id;


   @Column(name = "ARTIST_NAME")
   private String name;
}
```

@Temporal

- **@Temporal** визначає збереження дати/часу в БД
- Використовується із значеннями типу java.util.Date, java.util.Calendar
- Типи збереження:
  - TemporalType.DATE (java.sql.Date)
  - TemporalType.TIME (java.sql.Time)
  - TemporalType.TIMESTAMP (java.sql.Timestamp)

```
...
@Temporal(value=TemporalType.DATE)
@Column(name="BIO_DATE")
private Date bioDate;
...
```

TBL_ARTIST	
ARTIST_ID	NUMERIC
BIO_DATE	DATE

@Enumerated

- Визначає збереження в БД значень **enum**
- Типи збереження:
  - EnumType.ORDINAL (default)
  - EnumType.STRING

```
@Entity
public class Album {
   ...
   @Enumerated(EnumType.STRING)
   private Rating rating;
   ...
}
```

ALBUM	
ALBUM_ID	NUMERIC
RATING	VARCHAR(10)

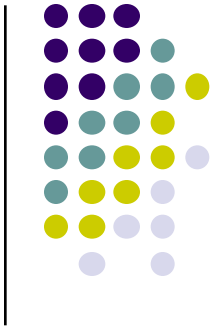
@Lob

- **@Lob** визначає збереження даних у колонки типу BLOB/CLOB
- Часто застосовується із @Basic для встановлення lazy-loading

```
@Entity
public class Album {
   ...
   @Lob
   @Basic (fetch = FetchType.LAZY)
   @Column(name = "ALBUM_ART")
   private byte[] artwork;
   ...
}
```

ALBUM	
ALBUM_ID	NUMERIC
ALBUM_ART	BLOB

@Version



- JPA підтримує оптимістичне блокування (на основі версій)
- Поле, яке анотоване @Version, не повинно змінюватись застосуванням
- Можливі типи даних: примітивні типи short, int, long, відповідні класи-оболонки (Short, Integer, Long), java.sql.Timestamp

```
@Version
private Integer version;
```

@Transient

- По замовчанню JPA вважає, що усі поля сутності зберігаються
- Якщо поле не повинно зберігатися, то воно повинно мати модифікатор `transient` або анотацію **@Transient**

```
@Entity
public class Genre {

   @Id
   private Long id;                    ← persistent

   private transient String value1;    ← not persistent

   @Transient
   private String value2;              ← not persistent
}
```

@Embedded, @Embeddable

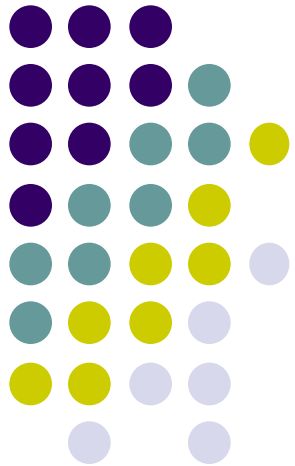
```
@Entity
public class Artist {
   ...
   @Embedded
   private Bio bio;
}
```

ARTIST	
ALBUM_ID	NUMERIC
BIO_DATE	DATE
BIO_TEXT	CLOB

```
@Embeddable
public class Bio {

   @Temporal(value=TemporalType.DATE)
   @Column(name="BIO_DATE")
   private Date bioDate;

   @Lob
   @Column(name="BIO_TEXT")
   private String text;
}
```

JPA

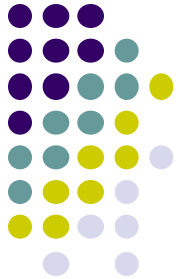
Анотації для відношень



Типи відношень

- JPA підтримує наступні типи відношень:
  - Один-до-одного
  - Один-до-багатьох
  - Багато-до-одного
  - Багато-до-багатьох
- JPA підтримує наступні типи відношені:
  - Одновнаправлені
  - Двонаправлені
- JPA підтримує наступні типи відношень:
  - Композиція (каскадні операції)
  - Агрегація

Двонаправлене відношення “один до одного”



- Двонаправлене відношення “один-до-одного” задається за допомогою анотації **@OneToOne** на обох сторонах відношення


```java
@Entity
public class Artist {

   @Id
   private Long id;

   @OneToOne
   @JoinColumn(name = "MANAGER_ID")
   private Manager manager;
}
```

```java
@Entity
public class Manager {

   @Id
   private Long id;

   @OneToOne(mappedBy="manager")
   private Artist artist;
}
```

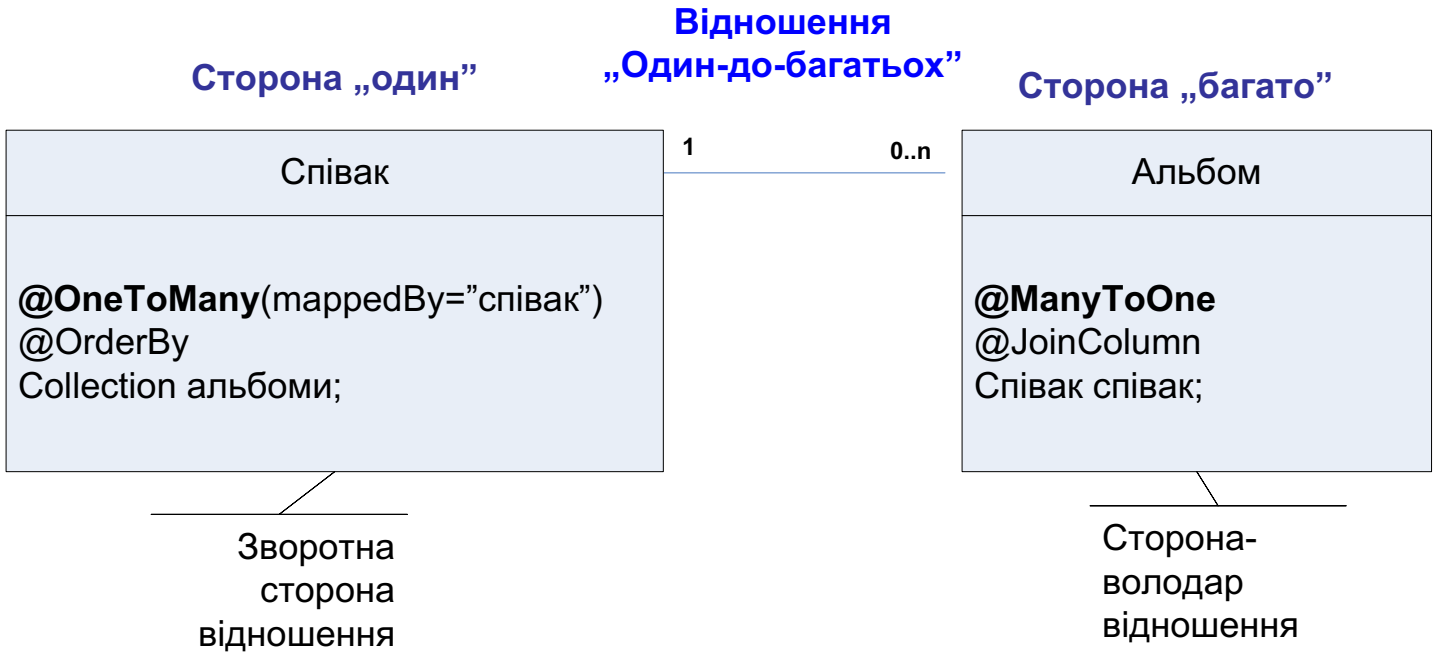
Визначає відношення на основі вторинного ключа для колонки MANAGER_ID

Двонаправлене відношення “один-до-багатьох”

- Двонаправлене відношення “один-до-багатьох” задається за допомогою анотацій **@OneToMany, @ManyToOne, @JoinColumn**

- Анотація **@OneToMany** визначається на стороні “один”
  - **@OneToMany** визначає колекцію дочірніх елементів, яка подається як реалізація інтерфейсу Collection
  - **@OrderBy** вказує, що при завантаженні колекції дочірніх елементів необхідне впорядкування

- Анотація **@ManyToOne** визначається на стороні “багато”
  - **@JoinColumn** визначає колонку, на яку вказує вторинний ключ



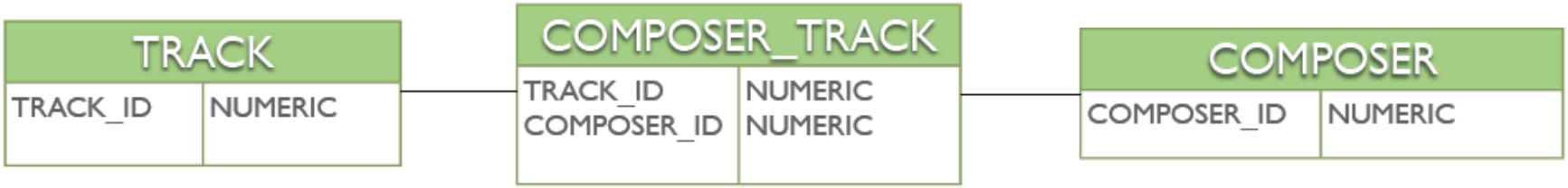
Приклад


```java
@Entity
public class Artist {

   @Id
   @Column(name = "ARTIST_ID")
   private Long id;

   @OneToMany(mappedBy = "artist")
   private Set<Album> albums =
                new HashSet<Album>();
}
```

```java
@Entity
public class Album {

   @Id
   @Column(name = "ALBUM_ID")
   private Long id;

   @ManyToOne
   @JoinColumn(name = "ARTIST_ID")
   private Artist artist;
}
```

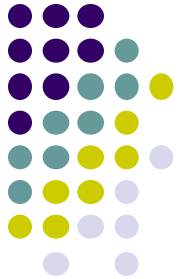
Двонаправлене відношення “багато-до-багатьох”

- Двонаправлене відношення “багато-до-багатьох” задається за допомогою анотації **@ManyToMany**, яка вказується на обох сторонах відношення
- **@JoinTable** вказує на стороні-володарі відношення
  - Сторона-володар вибирається довільно
- **@JoinColumn** вказує на колонки в зв’язувальній таблиці



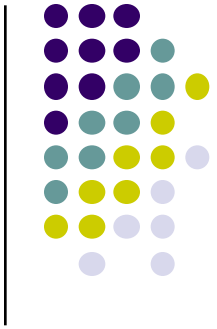

```
@Entity
public class Track {

    @Id
    @Column(name = "TRACK_ID")
    private Long id;

    @ManyToMany(mappedBy="compositions")
    private Set<Composer> composers
                = new HashSet<Composer>();
}
```

```
@Entity
public class Composer {
    @Id
    @Column(name = "COMPOSER_ID")
    private Long id;

    @ManyToMany
    @JoinTable(name="COMPOSER_TRACK",
    joinColumns = { @JoinColumn(name = "COMPOSER_ID") },
    inverseJoinColumns = { @JoinColumn(name = "TRACK_ID")
    private Set<Track> compositions;
}
```



- Як задаються однонаправлені відношення “один-до-одного”, “один-до-багатьох”, “багато-до-одного” та “багато-до-багатьох” – **для самостійного розгляду**

Каскадні операції



- JPA підтримує наступні каскадні операції для відношень:
  - **CascadeType.PERSIST** - створення
  - **CascadeType.MERGE** – збереження змін
  - **CascadeType.REMOVE** - видалення
  - **CascadeType.REFRESH** – поновлення стану
  - **CascadeType.DETACH** – detach-операція
  - **CascadeType.ALL** – усі операції

```
@Entity
public class Artist {
   @Id
   @Column(name = "ARTIST_ID")
   private Long id;
   @OneToMany(mappedBy = "artist“,
     cascade=CascadeType.ALL)
   private Set<Album> albums = new
     HashSet<Album>();
   …
}
```

```
@Entity
public class Album {
   @Id
   @Column(name = "ALBUM_ID")
   private Long id;
   @ManyToOne
   @JoinColumn(name = "ARTIST_ID")
    private Artist artist;
    …
}
```

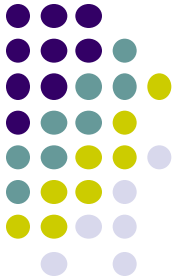
Fetch Type

FatchType:

- FetchType.***EAGER***
- FetchType.***LAZY***

```
@OneToMany(fetch=…)
@ManyToOne(fetch=…)
@ManyToMany(fetch=…)
```

Orphan Removal

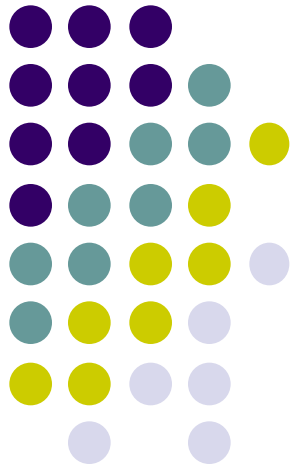


```
@OneToMany(orphanRemoval = true)
```

```
@Entity
public class Artist {
    @Id
    @Column(name = "ARTIST_ID")
    private Long id;
    @OneToMany(mappedBy="artist",
        cascade=CascadeType.ALL, orphanRemoval=true)
    private Set<Album> albums = new HashSet<Album>();
    …
}
```

JPA

Анотації для відношення успадкування

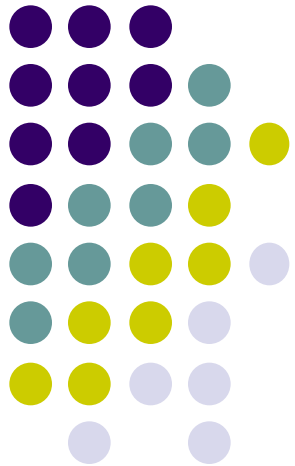


Відношення успадкування

- Стратегії збереження відношення успадкування задаються анотацією @Inheritance:
  - **InheritanceType.SINGLETABLE** – збереження усіх нащадків у одній таблиці
  - **InheritanceType.JOINED** – збереження полів нащадків у приєднаних таблицях
  - **InheritanceType.TABLE_PER_CLASS** – збереження кожної сутності у окремій таблиці

```
@Entity
@Table(name="PROJECT")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TYPE", discriminatorType=DiscriminatorType.STRING,length=20)
@DiscriminatorValue("P")
public class Project {

   @Id
   protected BigInteger id;
   protected String description;
   …
}

@Entity
@DiscriminatorValue("L")
public class LargeProject extends Project {

   protected BigInteger budget;
   …
}
```

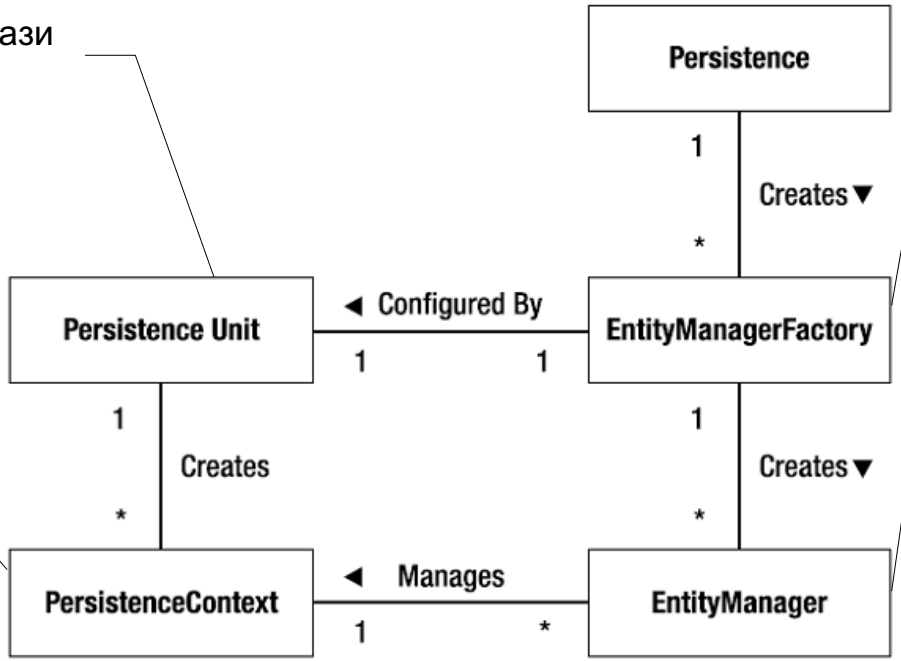
JPA Entity Manager



Основні поняття JPA

- Співвідношення між поняттями JPA:
  - Persistence (JPA), EntityManagerFactory, EntityManager, Persistence Unit, PersistenceContext

Persistence Unit задає наступні конфігураційні параметри:
- атрибути підключення до бази даних
- параметри EntityManager (параметри кешу тощо)

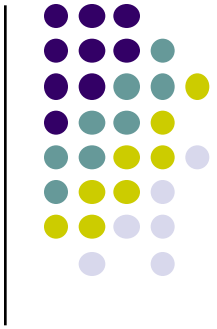


EntityManagerFactory – створює EntityManager на основі конфігураційних параметрів, заданих у Persistence Unit

EntityManager надає операції для роботи над сутностями, що зберігаються в БД:
- створення,
- збереження,
- видалення,
- пошук на основі запитів.

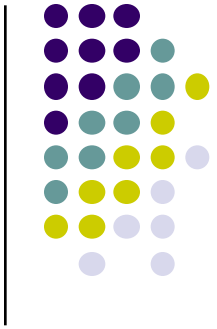
Persistence Context – це програмний елемент, який під час виконання пов'язує між собою:
- сутності, які зберігаються у БД, що вказана у Persistence Unit
- Entity Manager, який управляє вказаними сутностями

Persistence Unit



- Persistence Unit – це набір конфігураційних параметрів, який вказується в застосуванні у конфігураційному файлі **`persistence.xml`**
- В одному **`persistence.xml`** може бути задано декілька persistence unit
- Persistence Unit містить:
  - Назву
  - Загальні JPA-параметри (настройки кеша тощо)
  - Специфічні для JPA-провайдера параметри
  - Вказує на джерело даних (обмежений тільки одним джерелом даних)
  - Вказує ті класи сутностей, які будуть управлятися EntityManager
  - Вказані класи сутностей мають бути визначені в одному EJB, WAR, EAR модулі разом із файлом persistence.xml

persistence.xml

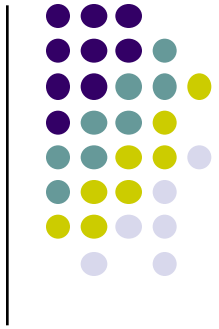


```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
   xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
   >

  <persistence-unit name="Students" transaction-type="JTA">
    <jta-data-source>AppJDBC</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-
   classes>
    <properties/>
  </persistence-unit>

</persistence>
```

Визначене джерело даних

В одному persistence.xml може бути задано декілька persistence unit

No persistence.xml



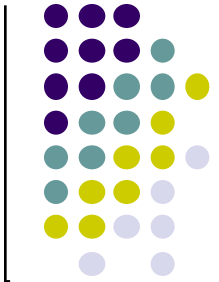
- Конфігурація може бути задана програмно, без використання persistance.xml
- В Spring будемо робити саме так

Persistence Context

- Persistence Context = контекст збереження

- Це “робоча копія” Persistence Unit (об’єкти, до яких йде доступ/що модифікуються під час транзакції)
- Після завершення транзакції –> flush to DB
- Для управляння Persistence Context надається API – Entity Manager

EntityManager



- **EntityManager** – це інтерфейс, який надає JPA, для управління сутностями. Забезпечує усі операції по збереженню та пошуку даних

- **Створення Entity Manager:**
  - Вручну
    - Назва: **EntityManager, що управляється застосуванням**
  - Автоматично контейнером
    - Назва: **EntityManager, що управляється контейнером**

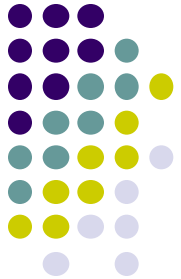
- **EntityManager** is not thread safe

EntityManager, що управляється застосуванням (2/2)

- Увага - транзакції локальні

```
public class CourseJpaDao {

    public CourseJpaDao() {
        emf = Persistence.createEntityManagerFactory("Students");
    }
    private EntityManagerFactory emf = null;

    public EntityManager getEntityManager() {
        return emf.createEntityManager();
    }

    public void create(Course course) {
        EntityManager em = null;
        try {
            em = getEntityManager();
            em.getTransaction().begin();
            em.persist(course);
            em.getTransaction().commit();
        } finally {
            if (em != null) { em.close(); }
        }
    }
…
}
```

EntityManager, що управляється контейнером (2/2)



- **Працює тільки для тих компонентів, які створюються контейнером**

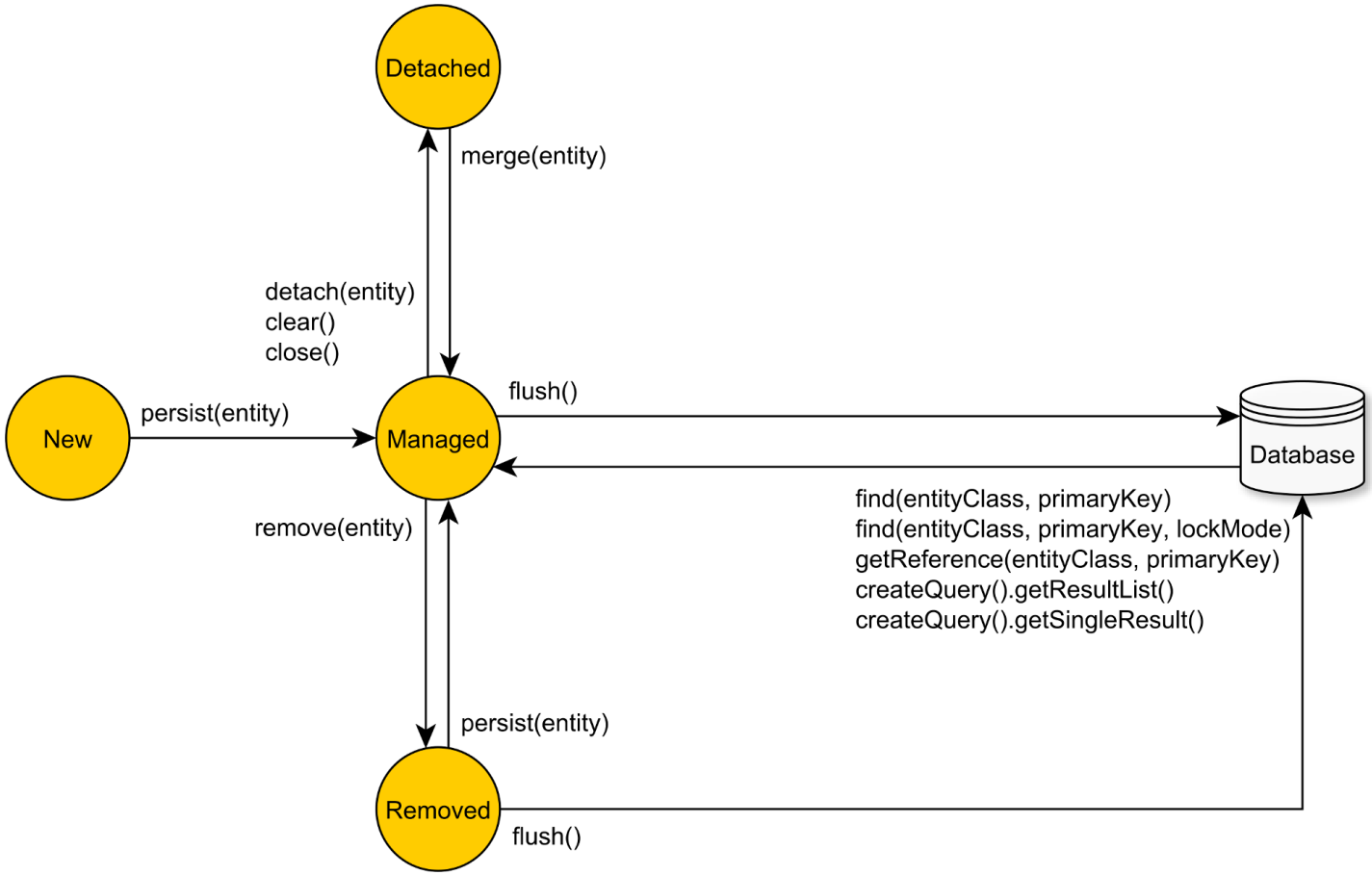
```
public class CourseFacade {
    @PersistenceContext(unitName = "Students")
    private EntityManager em;

    public void create(Course course) {   em.persist(course);   }

    public void edit(Course course) {  em.merge(course);  }

    public void remove(Course course) {  em.remove(em.merge(course));  }

    public Course find(Object id) {  return em.find(Course.class, id);  }

    public List<Course> findAll() {
        CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
        cq.select(cq.from(Course.class));
        return em.createQuery(cq).getResultList();
    }

    public int count() {
        CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
        Root<Course> rt = cq.from(Course.class);
        cq.select(em.getCriteriaBuilder().count(rt));
        Query q = em.createQuery(cq);
        return ((Long) q.getSingleResult()).intValue();
    }
}
```

Призначення основних методів EntityManager

Метод	Призначення
`flush`	Викликає синхронізацію тих об'єктів, які знаходяться в контексті збереження (persistence context), в БД. Викликається автоматично при завершенні транзакції.
`refresh`	Оновлює стан об'єкта в контексті збереження із БД
`find`	Пошук об'єкта. Виконується запит для пошуку об'єкта в БД по первинному ключу
`contains`	Перевіряє, чи є вказаний об'єкт є в контексті збереження. Якщо так, то це означає, що об'єкт знаходится у стані "управляється"
`merge`	Занурює стан вказаного об'єкту в контекст збереження (=операція модифікації)
`remove`	Видалення вказаного об'єкту
`persist`	Робить об'єкт таким, що зберігається (=операція створення в БД)
`detach`	Від'єднати об'єкт від PersistanceContext

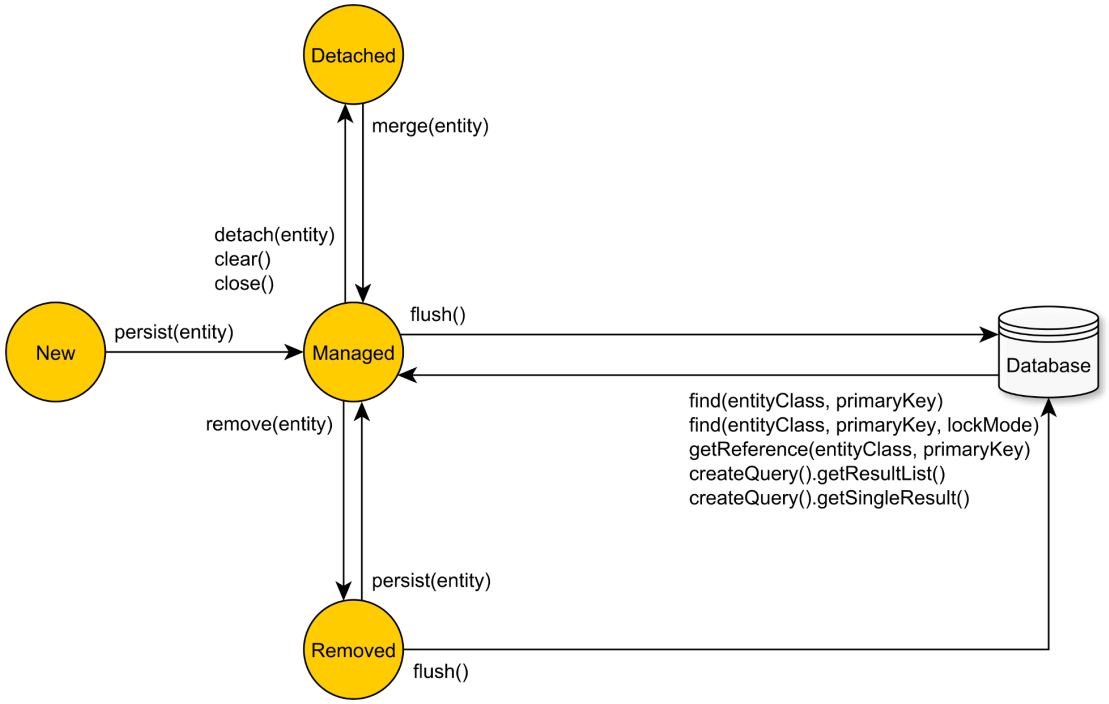
Стани об’єктів (1/2)

- На рисунку подано стани об’єктів в PersistanceContext та відповідні операції EntityManager, які переводять між станами



Стани об’єктів (2/2)

- Сутність, що зберігається, може бути в одному із 4 станів:
  - **“новий” (new)** – об’єкт створений та ще не пов’язаний із контекстом збереження
  - **“управляється” (managed)** – об’єкт пов’язаний із контекстом збереження та має унікальний ідентифікатор. Тільки один об’єкт із вказаним ідентифікатором може існувати в контексті збереження
  - **“відокремлений” (detached)** – об’єкт не пов’язаний із контекстом збереження
  - **“видалений” (removed)** - об’єкт має бути видалений



Transactions and Persistence Context

- Commit (flush)
- Rollback (clear)

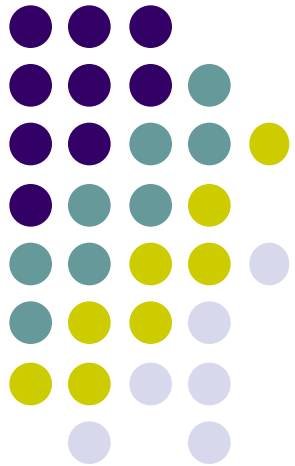
Сповіщення про виконання операцій над об’єктом

- Екземпляр сутності, що зберігається, може бути сповіщений про виконання операцій. Це настроюється за допомогою наступних анотацій:
  - `@PrePersist / @PostPersist`
  - `@PreRemove / @PostRemove`
  - `@PreUpdate / @PostUpdate`
  - `@PostLoad`
  - `@EntityListeners`

- Приклад:

```
@Entity
public class Employee {
    @Id
    private int id;
    private String name;

    @PostLoad
    private void foo () {
        System.out.println(
            "Employee.foo called on employee id: " + getId());
    }
...
}
```

JPA. Запити



Методи розробки запитів

- JPA надає наступні методи для розробки запитів:
  - Мова запитів Java Persistence Query Language (JPQL). Схожа на SQL.
  - Criteria API. Створення запитів, використовуючи спеціалізоване API.
  - Мова запитів SQL (Native Query)

JPQL (1/2)

Приклади

- Іменовані параметри

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

- Позиційні параметри

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

- Запит сторінки даних

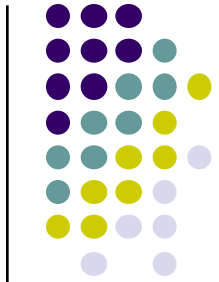
```
public List findAll(int first, int size) {
    return em.createQuery(
        "SELECT c FROM Customer c")
        .setMaxResults(size)
        .setFirstResult(first)
        .getResultList();
}
```

JPQL (2/2)

Синтаксис мови запитів дуже багатий. Див. приклади у Java EE 6 Tutorial. С. 404:

- `SELECT p FROM Player p`
- `SELECT DISTINCT p FROM Player p, IN(p.teams) t`
  - Отримання усіх тих гравців, які грають в якихось командах
  - Той самий запит: `SELECT DISTINCT p FROM Player p WHERE p.team IS NOT EMPTY`
- `SELECT DISTINCT p FROM Player p, IN (p.teams) t WHERE t.league = :league`
  - Отримання тих гравців, які грають в командах вказаної ліги
- І так далі

SQL



- Приклад:

```
Query query = em.createNativeQuery("SELECT * FROM Customer",
   Customer.class);
List customers = query.getResultList();
```

Criteria API

- Приклад

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Metamodel m = em.getMetamodel();
EntityType<Pet> Pet_ = m.entity(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.where(pet.get(Pet_.color).isNull());
cq.select(pet);
TypedQuery<Pet> q = em.createQuery(cq);
List<Pet> allPets = q.getResultList();
```