JPA (Java Persistence API)

By
Srikanth Pragada

What is Persistence?

- Storing an object beyond the process (program) that created it is called persistence.
- Objects are created in Java program. They are in RAM.
- Objects are to be persisted to tables in relational database, which is on hard disk.

Paradigm Mismatch or Impedance Mismatch

- Object contains data in the form of attributes.
- Objects may be associated with other objects. They hold references to other objects.
- Object may be derived from other objects – inheritance.
- Data in the database is in the form of rows and columns.
- A table references other tables using foreign key.
- There is no support for inheritance in relational model.

How persistence is handled in Java?

- We use JDBC API to convert objects to rows in table.
- To retrieve data from database and to project it as a collection, we need to retrieve rows, convert rows to objects and place them in a collection.
- JDBC needs considerable amount of code.

What is ORM?

- Stands for Object-Relational Mapping.
- Maps objects to relational tables automatically and transparently.
- Uses metadata or XML entries to map objects to relational table.
- Provides API for performing CRUD (create, read, update and delete).
- Provides a language for queries.
- Provides a way to interact with transaction manager, dirty checking, caching and optimizing fetching.

Why ORM?

Productivity

Allows you to concentrate on business logic leaving persistence plumbing.

Maintainability

Fewer lines of code, allows code to be more maintainable.

Performance

Provides better performance with better fetching strategy and caching.

Vendor independence

Abstracts application from underlying database differences.

JPA (Java Persistence API)

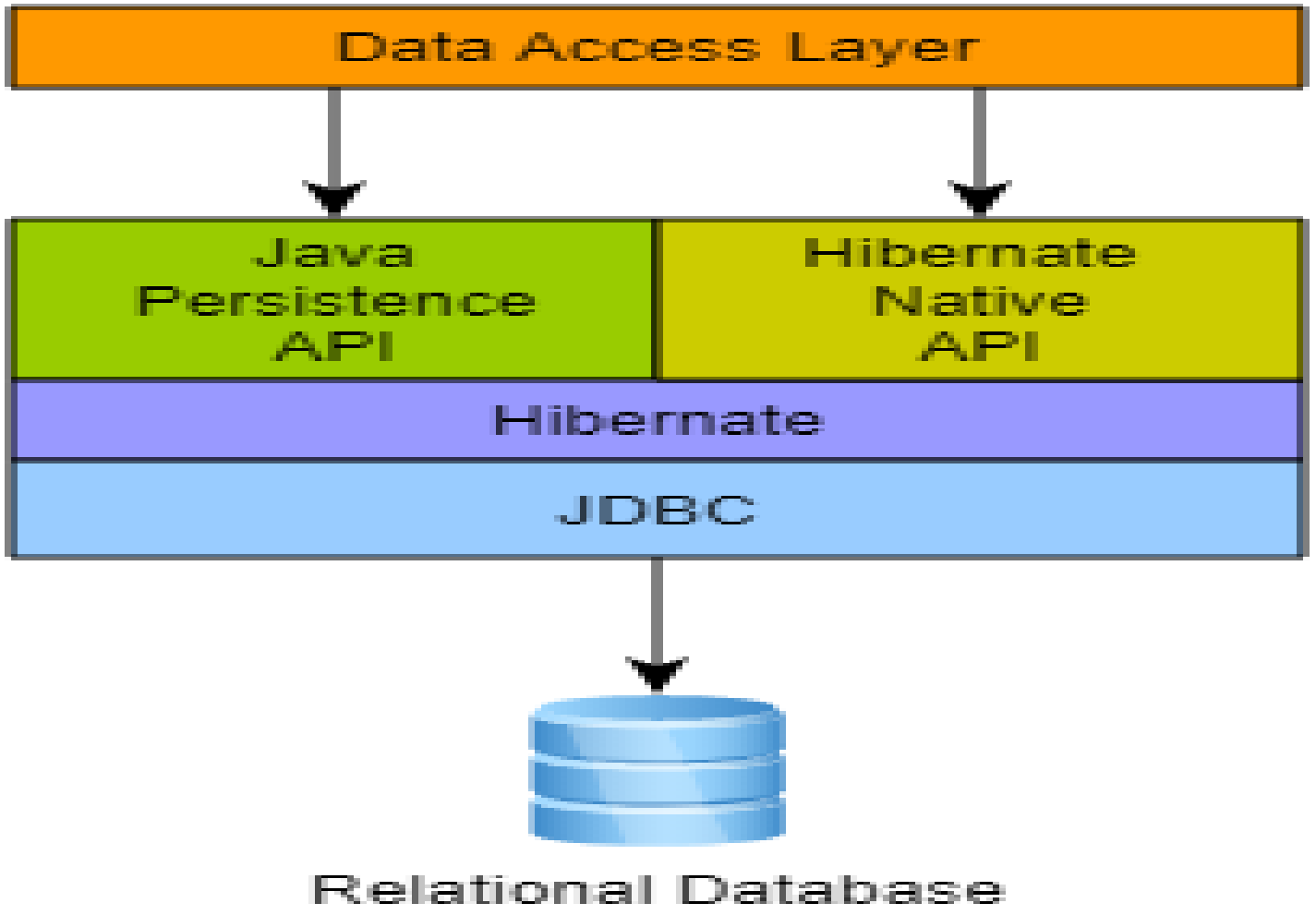
- The Java Persistence API (JPA) is a Java standards–based solution for persistence.
- Persistence uses an object/relational mapping approach to bridge the gap between an object-oriented model and a relational database.
- The Java Persistence API can also be used in Java SE applications outside of the Java EE environment.
- Java Persistence consists of the following areas:
  - ✓ The Java Persistence API
  - ✓ The query language
  - ✓ Object/relational mapping metadata

- Current version is Java Persistence API 2.2.

Hibernate as ORM

- Hibernate is Object/relational mapping framework for enabling transparent POJO persistence.
- Developed by Gavin King.
- It is a professional Open Source project and a critical component of the JBoss Enterprise Middleware System (JEMS) suite of products.
- Available at **hibernate.org.**
- Hibernate is a standard implementation of the JPA specification, with a few additional features that are specific to Hibernate.
- Allows you to build persistent objects following common OO programming concepts:
  - Association
  - Inheritance
  - Polymorphism
  - Composition

Hibernate as ORM



Version of Hibernate

- Hibernate **5.6.5** released on 26-Jan-2022 is the latest version.
- It is compatible with Java 8, 11 or 17.
- Implements **JPA 2.2**

JPA using Maven

- Maven is a dependency management tool.
- The main Hibernate ORM artifact is named **hibernate-core**. Add it to POM.XML

```
<dependency>
        <groupId>com.microsoft.sqlserver</groupId>
        <artifactId>mssql-jdbc</artifactId>
        <version>11.2.0.jre17</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.6.5.Final</version>
</dependency>
<dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>5.6.5.Final</version>
</dependency>
<dependency>
        <groupId>org.hibernate.javax.persistence</groupId>
        <artifactId>hibernate-jpa-2.1-api</artifactId>
        <version>1.0.2.Final</version>
</dependency>
```

A simple JPA Application

Here are the steps to build a simple Hibernate Application:

- ❑ Add required .jar files to your project using in **pom.xml**.
- ❑ Define the domain model – **Entity** classes.
- ❑ Setup your persistence configuration - **persistence.xml**.
- ❑ Create **EntityManager** using **EntityManagerFactory**.
- ❑ Write code to use EntityManager to manage entities.

Persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence   …>
 <persistence-unit name="mssqlserver" transaction-type="RESOURCE_LOCAL">
  <!-- Persistence provider -->
  <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
  <class>jpademo.Category</class>
  <exclude-unlisted-classes>true</exclude-unlisted-classes>
  <properties>
  <property name="javax.persistence.jdbc.driver" value="com.microsoft.sqlserver.jdbc.SQLServerDriver" />
  <property name="javax.persistence.jdbc.url"
         value="jdbc:sqlserver://srikanthlaptop\\sqlexpress:1433;user=sa;password=sa;database=msdb;
         encrypt=true;trustServerCertificate=true" />
  <property name="javax.persistence.jdbc.user" value="sa" />
  <property name="javax.persistence.jdbc.password" value="sa" />
  <property name="hibernate.dialect" value="org.hibernate.dialect.SQLServer2012Dialect" />
  <property name="hibernate.hbm2ddl.auto" value="none" />
  <property name="hibernate.show_sql" value="true" />
  <property name="hibernate.format_sql" value="true" />
  </properties>
 </persistence-unit>
</persistence>
```

Category.java

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="categories")
public class Category {
        @Id
        @Column(name="catcode")
        private String code;

        @Column(name="catdesc")
        private String description;

        // getter and setter methods
}
```

A simple Application – Add User

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("mssqlserver");
EntityManager em = emf.createEntityManager();

Category c = new Category();
c.setCode("c2");
c.setDescription("Category 2");

em.getTransaction().begin();
em.persist(c);
em.getTransaction().commit();

em.close();
emf.close();
```

ListCategories.java

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("mssqlserver");
EntityManager em = emf.createEntityManager();

List<Category> cats = em.createQuery("from Category", Category.class)
                        .getResultList();

for (Category cat : cats) {
    System.out.printf("%s - %s\n", cat.getCode(), cat.getDescription());
}

em.close();
emf.close();
```

Annotations Reference

- ❑ `@Entity`
- ❑ `@Table`
- ❑ `@Id`
- ❑ `@Transient`
- ❑ `@Column`
- ❑ `@Embedded`
- ❑ `@Embeddable`
- ❑ `@EmbeddedId`
- ❑ `@GeneratedValue`
- ❑ `@AttributeOverride`

@Column Annotation

```
@Column(
    name="columnName";                                          1
    boolean unique() default false;                             2
    boolean nullable() default true;                            3
    boolean insertable() default true;                          4
    boolean updatable() default true;                           5
    String columnDefinition() default "";                       6
    String table() default "";                                  7
    int length() default 255;                                   8
    int precision() default 0; // decimal precision             9
    int scale() default 0; // decimal scale                     10
```

1. name (optional): the column name (default to the property name)
2. unique (optional): set a unique constraint on this column or not (default false)
3. nullable (optional): set the column as nullable (default true).
4. insertable (optional): whether or not the column will be part of the insert statement (default true)
5. updatable (optional): whether or not the column will be part of the update statement (default true)
6. columnDefinition (optional): override the sql DDL fragment for this particular column (non portable)
7. table (optional): define the targeted table (default primary table)
8. length (optional): column length (default 255)
8. precision (optional): column decimal precision (default 0)
10. scale (optional): column decimal scale if useful (default 0)

Entity Life Cycle Annotation

Annotation	When method is called
@PrePersist	Before persist is called for a new entity
@PostPersist	After persist is called for a new entity
@PreRemove	Before an entity is removed
@PostRemove	After an entity has been deleted
@PreUpdate	Before the update operation
@PostUpdate	After an entity is updated
@PostLoad	After an entity has been loaded

JDBC connection properties

Usually, you want to have the **SessionFactory** to be created and pool JDBC connections. As soon as you do something that requires access to the database, a JDBC connection will be obtained from the pool. For this to work, we need to pass some JDBC connection properties to Hibernate.

Property Name	**Purpose**
hibernate.connection.driver_class	jdbc driver class
hibernate.connection.url	jdbc URL
hibernate.connection.username	database user username
hibernate.connection.password	database user password
hibernate.connection.pool_size	maximum number of pooled connections

Entity Classes

- Must have no-argument constructor.
- Optionally contains identifier property – primitive type, wrapper classes, String, Date, or user-defined for composite primary key.
- Prefers non-final classes.
- By default, Hibernate persists JavaBeans style properties, and recognizes method names of the form getFoo, isFoo and setFoo.
- Properties need not be declared public - Hibernate can persist a property with a default, protected or private get / set pair.

DataSource properties

For use inside an application server, you should always configure Hibernate to obtain connections from an application server Datasource registered in JNDI. You'll need to set at least one of the following properties:

Property name	**Purpose**
hibernate.connection.datasource	Datasource JNDI name
hibernate.jndi.url	URL of the JNDI provider (optional)
hibernate.jndi.class	Class of the JNDI InitialContextFactory (optional)
hibernate.connection.username	Database user username (optional)
hibernate.connection.password	Database user password (optional)

SQL Dialects

You should always set the **hibernate.dialect** property to the correct **org.hibernate.dialect.Dialect** subclass for your database.

RDBMS	Dialect
DB2	org.hibernate.dialect.DB2Dialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
MySQL	org.hibernate.dialect.MySQLDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 9i	org.hibernate.dialect.Oracle9iDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Sybase	org.hibernate.dialect.SybaseDialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Informix	org.hibernate.dialect.InformixDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Ingres	org.hibernate.dialect.IngresDialect
Progress	org.hibernate.dialect.ProgressDialect

Basic Value Types

The built-in *basic mapping types* may be roughly categorized into the following categories.

Data Type	Description
integer, long, short, float, double, character, byte, boolean, yes_no, true_false	Type mappings from Java primitives or wrapper classes to appropriate (vendor-specific) SQL column types.
String	A type mapping from java.lang.String to VARCHAR (or Oracle VARCHAR2).
date, time, timestamp	Type mappings from java.util.Date and its subclasses to SQL types DATE, TIME and TIMESTAMP (or equivalent).
calendar, calendar_date	Type mappings from java.util.Calendar to SQL types TIMESTAMP and DATE (or equivalent).
big_decimal, big_integer	Type mappings from java.math.BigDecimal and java.math.BigInteger to NUMERIC (or Oracle NUMBER).
locale, timezone, currency	Type mappings from java.util.Locale, java.util.TimeZone and java.util.Currency to VARCHAR. Instances of Locale and Currency are mapped to their ISO codes. Instances of TimeZone are mapped to their ID.
class	A type mapping from java.lang.Class to VARCHAR (or Oracle VARCHAR2). A Class is mapped to its fully qualified name.

Basic Value Types

Data Type	Description
binary	Maps byte arrays to an appropriate SQL binary type.
text	Maps long Java strings to SQL CLOB or TEXT type.
serializable	Maps serializable Java types to an appropriate SQL binary type. You may also indicate the Hibernate type serializable with the name of a serializable Java class or interface that does not default to a basic type.
clob, blob	Type mappings for the JDBC classes are java.sql.Clob and java.sql.Blob. These types may be inconvenient for some applications, since the blob or clob object may not be reused outside of a transaction. (Furthermore, driver support is patchy and inconsistent.)
imm_date, imm_time, imm_timestamp, imm_calendar, imm_calendar_date, imm_serializable, imm_binary	Type mappings for what are usually considered mutable Java types, where Hibernate makes certain optimizations appropriate only for immutable Java types, and the application treats the object as immutable. For example, you should not call Date.setTime() for an instance mapped as imm_timestamp. To change the value of the property, and have that change made persistent, the application must assign a new (non-identical) object to the property.

@GeneratedValue Annotation

- Provides for the specification of generation strategies for the values of primary keys.
- The **GeneratedValue** annotation is applied to a primary key property or field of an entity
- The parameter **strategy** specifies strategy that the persistence provider must use to generate the annotated entity primary key.
- Parameter **generator** provides the name of the primary key generator to use as specified in the SequenceGenerator or TableGenerator annotation.
- Enum **GenerationType** provides constants related to generation strategy.

AUTO	Indicates that the persistence provider should pick an appropriate strategy for the particular database.
IDENTITY	Indicates that the persistence provider must assign primary keys for the entity using a database identity column.
SEQUENCE	Indicates that the persistence provider must assign primary keys for the entity using a database sequence.
TABLE	Indicates that the persistence provider must assign primary keys for the entity using an underlying database table to ensure uniqueness.

```
@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQ_STORE")

@GeneratedValue(strategy=GenerationType.IDENTITY)
```

Composite Primary Key

- A composite primary key, also called a composite key, is a combination of two or more columns to form a primary key for a table.
- In JPA, we have two options to define the composite keys: the **@IdClass** and **@EmbeddedId** annotations.
- In order to define the composite primary keys, we should follow some rules:
  - ✓ The composite primary key class must be public.
  - ✓ It must have a no-arg constructor.
  - ✓ It must define the *equals()* and *hashCode()* methods.
  - ✓ It must be *Serializable.*

TitleAuthorPK.java

```
@Embeddable
public class TitleAuthorPK  implements Serializable {
	@Column(name = "au_id")
	private String authorId;
	@Column(name = "title_id")
	private String titleId;

	// constructor with and without args

	// equals() and hashCode() methods

	// getter and setter methods
}
```

TitleAuthor.java

```
@Entity(name = "TitleAuthorCompositePK")
@Table(name = "TitleAuthor")
public class TitleAuthor {
	@EmbeddedId
	private TitleAuthorPK key;

	@Column(name = "au_ord")
	private int order;

	// getter and setter methods

}
```

Collection Mapping

```
@ElementCollection
@CollectionTable
```

```
@ElementCollection
@CollectionTable(name = "Programmer_Skills",
                 joinColumns = @JoinColumn(name = "programmer_id"))
private Set<String> skills = new HashSet<String>();
```

EntityTransaction Methods

- EntityManager.getTransaction() method returns the EntityTransaction object.
- A single-threaded, short-lived object used by the application to specify atomic units of work.
- Abstracts application from underlying JDBC or JTA transaction.
- A EntityManager might span several Transactions in some cases.
- However, transaction demarcation, either using the underlying API or Transaction, is never optional.

Method	**Meaning**
void begin()	Begin a new transaction.
void commit()	Commit the current resource transaction, writing any unflushed changes to the database.
boolean isActive()	Indicate whether a resource transaction is in progress.
void rollback()	Roll back the current resource transaction.
void setRollbackOnly	Mark the current resource transaction so that the only possible outcome of the transaction is for the transaction to be rolled back.
boolean getRollbackOnly	Determine whether the current resource transaction has been marked for rollback.

Association Annotation

- `@OneToOne`
- `@JoinColumn`
- `@JoinTable`
- `@OneToMany`
- `@ManyToOne`
- `@ManyToMany`

Instance States

The following are the different states in which an instance could be.

Transient

- The instance is not, and has never been associated with any Entity Manager.
- It has no persistent identity (primary key value).
- It has no corresponding row in the database.

Managed / Persist

- The instance is currently associated with a Entity Manager.
- It has a persistent identity (primary key value) and likely to have a corresponding row in the database.
- Changes to objects in this state are automatically saved to the database.

Detached

- The instance was once associated with a Entity Manager, but that context was closed, or the instance was serialized to another process.
- It has a persistent identity and, perhaps, a corresponding row in the database.
- No longer managed by Entity Manager.
- Can be reattached using method like merge().

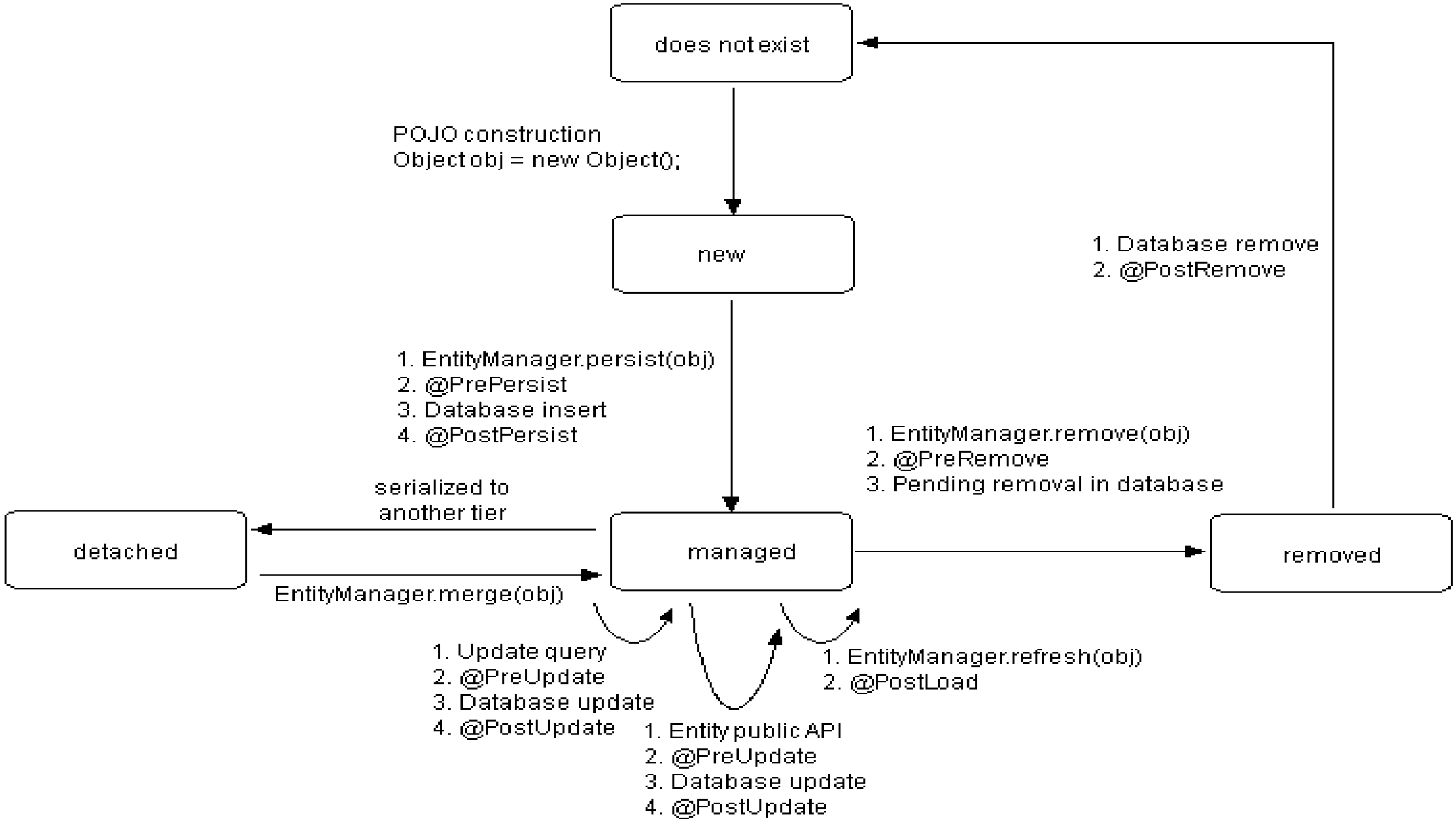
Removed

- A previously persistent object that is deleted from the database using remove() method.
- Java instance may still exist, but it is ignored by Entity Manager.

Instance States

```
EntityManagerFactory emf =
         Persistence.createEntityManagerFactory("mssqlserver_msdb");
EntityManager em = emf.createEntityManager();

Category c = new Category();
c.setCode("c10");
c.setDescription("Category 10");

// Entity c is Transient

em.getTransaction().begin();
em.persist(c);
// Entity c is Persistent
em.getTransaction().commit();

em.close();
emf.close();

// Entity c is Detached
```

Instance States



State Transition

- Transient instances may be made persistent by calling **persist**() method.
- Persistent instances may be made removed by calling **remove**() method.
- Any instance returned by **find**() method is persistent.
- Detached instances may be made persistent by calling **merge**() method.

Removing an Object

The following code shows how to delete an object thereby causing corresponding row to be deleted from table.

```
Category c = em.find(Category.class, "c1");

if (c == null) {
      System.out.println("Sorry! Category not found!");
}
else {
       em.getTransaction().begin();
       em.remove(c);
       em.getTransaction().commit();
}
```

Composition using Component Class

- A component is a contained object that is persisted as a value type, not an entity reference.
- The term "component" refers to the object-oriented notion of composition.

```
@Embeddable
public class Address {
	private String address, city;
```

```
@Entity
public class Customer {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private int id;
	private String name;

	@Embedded
	private Address home;
```

Composition using Component Class

```
Customer c = new Customer();
c.setName("Larry Page");
Address a = new Address();
a.setAddress("33-303-33");
a.setCity("California");

c.setHome(a);

em.getTransaction().begin();
em.persist(c);
em.getTransaction().commit();
```

customer

id	address	city	name
3	33-303-33	California	Larry Page

Collection Mapping

- Hibernate requires that persistent collection-valued fields be declared as an interface type.
- The actual interface might be Set, Collection, List, Map, SortedSet, SortedMap.
- Use **@ElementCollection** and **@CollectionTable** annotations to specify collection and associated table.

```
@Entity
public class Programmer {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private int id;
	private String name;

	@ElementCollection
	@CollectionTable(name = "Programmer_Skills",
	                 joinColumns = @JoinColumn(name = "programmer_id"))
	private Set<String> skills = new HashSet<String>();
```

Collection Mapping

```
Programmer p = new Programmer();
p.setName("Hunter");
p.getSkills().add("Java");
p.getSkills().add("Python");

em.getTransaction().begin();
em.persist(p);
em.getTransaction().commit();
```

programmer

id	name
1	Hunter

programmer_skills

programmer_id	skills
1	Java
1	Python

Bulk Updates

- JPA provides methods for bulk SQL-style DML statement execution which are performed through the JPQL Language.
- It is more optimal to run directly in the database (not in memory) as it avoids loading potentially thousands of records into memory to perform the exact same action.
- Changes made to database records are NOT reflected in any in-memory objects.

```
Query q = em.createQuery
    ("UPDATE Product p SET p.price = p.price + 100 WHERE p.price > 1000");

em.getTransaction().begin();
int count = q.executeUpdate();
em.getTransaction().commit();
```

Flushing Context and Flushing Mode options

- Submits the stored SQL statements to the database.
- Flushing occurs :
  - ✓ When transaction.commit() is called
  - ✓ When session.flush() is called explicitly
  - ✓ Before a query is executed, if stored statements would affect the results of the query
- You can determine whether changes are to be committed using isDirty() method.
- Methods **setFlushMode**() and **getFlushMode**() set and get flush mode of the current session.

Option	Description
ALWAYS	Every query flushes the session before the query is executed.
AUTO (default)	Hibernate manages the query flushing to guarantee that the data returned by the query is up to date.
COMMIT	Hibernate flushes the session on transaction commits.
NEVER	Application needs to manage the session flushing with flush() methods. Hibernate never flushes the session itself.

Association Mapping

- Relationship between entity classes can be represented in Hibernate.
- We must decide the cardinality and direction of the relationship.
- Relationship between Entities is represented in mapping file.
- Entities also need to have attributes to represent relationships between entities.

Cardinality or Multiplicity

- Multiplicity deals with how many instances could be present on each side of the relationship.
  - ✓ one-to-one
  - ✓ many-to-one
  - ✓ one-to-many
  - ✓ many-to-many

Directionality of Relationship

Unidirectional

- Can only traverse objects from one side of the relationship.
- Given an Account object, we can obtain related transaction objects.
- Given a Transaction object, we cannot obtain related Account object.

Bidirectional

- Can traverse objects from both sides of the relationship.
- Given an Account object, we can obtain related Transaction objects.
- Given a Transaction object, we can obtain related Account object.

One to One Relationship

- Expresses a relationship between two entities where each instance of the first entity is related to a single instance of the second or vice versa.
- Can be expressed in the database in two ways:
  - ✓ Giving each of the respective tables the same primary key values
  - ✓ Using foreign key constraint from one table onto a unique identifier column of the other

One to One Relationship

```
@Entity
@Table(name = "employees")
public class Employee {
	// code
	@OneToOne(cascade = CascadeType.ALL)
	@JoinColumn(name = "project_id", referencedColumnName = "project_id")
	private Project project;
	// code
}
```

```
@Entity
@Table(name = "projects")
public class Project {
	// code
	@OneToOne(mappedBy = "project")
	private Employee employee;
	// code
}
```

One to One Relationship

```
Employee e = new Employee();
e.setName("Employee1");
Project p = new Project();
p.setTitle("Project1");

e.setProject(p);

em.getTransaction().begin();
em.persist(e);
em.getTransaction().commit();
```

employees

emp_id	emp_name	project_id
1	Employee1	1

projects

project_id	project_title
1	Project1

Many To One/One To Many Relationship

- An entity refers to another entity through a reference.
- Relationship is defined in many side.
- Foreign key is used in many-side table to represent relationship in database.
- A collection is used on the one side class to contain references to many objects.
- Inverse side of the relationship is the one that doesn’t store relationship.

Category.java

```
@Entity
@Table(name = "categories")
public class Category {
	@Id
	@Column(name = "catcode")
	private String code;
	@Column(name = "catdesc")
	private String description;

	@OneToMany(cascade = CascadeType.ALL, mappedBy = "category")
	private List<Product> products = new ArrayList<Product>();

	// getter and setter methods

}
```

The value of ***mappedBy*** is the name of the association-mapping attribute on the owning side.

Product.java

```
@Entity
@Table(name = "products")
public class Product {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name = "prodid")
        private int id;

        @ManyToOne
        @JoinColumn(name="catcode")
        private Category category;


}
```

The ***@JoinColumn*** annotation defines that actual physical mapping on the owning side.

ListProducts.java

```
List<Product> prods = em.createQuery("from Product", Product.class).getResultList();

for (Product p : prods) {
   System.out.printf("%s - %s\n", p.getCategory().getDescription(), p.getName());
}
```

ListCategories.java

```
List<Category> cats = em.createQuery("from Category", Category.class).getResultList();

for (Category cat : cats) {
    System.out.printf("%s  %s\n", cat.getCode(), cat.getDescription());
    for(Product p : cat.getProducts()) {
       System.out.println(p.getName());
    }
}
```

Many To Many Relationship

- A single employee deals with many projects and a single project has many employees.
- On both sides, map the Collection objects.
- Either side can be made inverse.

Many To Many Relationship

```
@Entity(name = "Author")
@Table(name = "authors")
public class Author {
      // code
      @ManyToMany(mappedBy = "authors")
      Set<Title> titles;
      // code
}
```

```
@Entity
@Table(name = "titles")
public class Title {
      @ManyToMany
      @JoinTable(name = "titleauthor",
                  joinColumns = @JoinColumn(name = "title_id"),
                  inverseJoinColumns = @JoinColumn(name = "au_id"))
      Set<Author> authors;
```

Many To Many Relationship

```
List<Author> authors = em.createQuery("from Author", Author.class).getResultList();

for (Author author : authors) {
	System.out.println(author.getName());
	for (Title title : author.getTitles()) {
		System.out.println(title.getTitle().indent(5));
	}
}
```

Cascading

Propagate the persistence action not only to the object submitted, but also to any objects associated with that object.

none

- Default behavior. No cascading is done.

save-update

- Saves or updates associated objects.
- Associated objects can be transient or detached.

delete

- Deletes associated persistent instances.

delete-orphan

- Enables deletion of associated objects when they're removed from a collection.
- Enabling this tells Hibernate that the associated class is NOT SHARED, and can therefore be deleted when removed from its associated collection.

Implementing Inheritance

- Hibernate supports the three basic inheritance mapping strategies:
  - ✓ table per class hierarchy
  - ✓ table per subclass
  - ✓ table per concrete class

Inheritance Annotation

- @Inheritance
- @DiscriminatorColumn
- @DiscriminatorValue
- @PrimaryKeyJoinColumn
- @MappedSuperclass

Table per Class Hierarchy

A single table for the whole class hierarchy. Discriminator column contains key to identify the base type.

Advantage

Offers best performance even for deep hierarchy since single select may suffice.

Disadvantage

Changes to members of the hierarchy require column to be altered, added or removed from the table.

Implementing Inheritance - Table per class hierarchy

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="person_type", discriminatorType = DiscriminatorType.INTEGER)
public class Person {

}
```

```
@Entity
@DiscriminatorValue("1")
public class Player extends Person {
```

```
@Entity
@DiscriminatorValue("2")
public class Student extends Person {
```

Implementing Inheritance - Table per class hierarchy

person

id	name	game	college	person_type
2	Dhoni	Cricket	*NULL*	1
3	Jason	*NULL*	Stanford	2

Table per subclass

Foreign key relationship exists between common table and subclass tables.

Advantages

- ❑ Does not require complex changes to the schema when a single parent class is modified.
- ❑ Works well with shallow hierarchy.

Disadvantages

- ❑ Can result in poor performance – as hierarchy grows, the number of joins required to construct a leaf class also grows.

Implementing Inheritance - Table per subclass

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Course {
```

```
@Entity
public class OfflineCourse extends Course {
```

```
@Entity
public class OnlineCourse extends Course {
```

Implementing Inheritance - Table per subclass

course

id	name
1	AWS
2	Java Langauge

offline_course

id	location
2	Srikanth Technologies, Dwarkanagar

online_course

id	url
1	https://meet.goto.com/81883833

Table per concrete subclass

Map each of the concrete classes as normal persistent class.

Advantages

- Easiest to implement

Disadvantages

- Data belonging to a parent class is scattered across a number of different tables, which represent concrete classes
- A query of parent class is likely to cause a large number of SELECT operations
- Changes to a parent class can touch large number of tables

Querying In Hibernate

The following are the various options provided by JPA for querying the data from data source:

- JPQL – Hibernate Query Language
- Named Queries
- Query By Example (QBE)
- Criteria
- Native SQL

JPQL (Java Persistence Query Language)

- Looks very much like SQL.
- JPQL is fully object-oriented, understanding notions like inheritance, polymorphism and association.
- Entity manager may turn one JPQL statement into several SQL statements.
- Queries are case-insensitive, except for names of Java classes and properties.

JPQL

- SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY Clauses
- DISTINCT to eliminate duplicates
- Arithmetic operators +, -, *, /
- Relational operators =, >=, <=, <>, !=, like
- Logical operations and, or, not
- Parentheses ( ), indicating grouping
- Other operators are – IN, NOT IN, LIKE, BETWEEN ..AND, IS NULL, IS NOT NULL, IS EMPTY, IS NOT EMPTY
- CASE expression

JPQL Functions

CONCAT(String, String)	String
LENGTH(String)	int
LOCATE(String, String [, start])	int
SUBSTRING(String, start, length)	String
TRIM([[LEADING TRAILING BOTH] char) FROM] (String)	String
LOWER(String)	String
UPPER(String)	String

ABS(number)	int, float, or double
MOD(int, int)	int
SQRT(double)	double
SIZE(Collection)	int

CURRENT_DATE	java.sql.Date
CURRENT_TIME	java.sql.Time
CURRENT_TIMESTAMP	java.sql.Timestamp

Query Interface

- An object-oriented representation of a Hibernate query.
- A particular page of the result set may be selected by calling setMaxResults(), setFirstResult().
- Supports named query parameters (:name) and JDBC style parameters (?).
- You may not mix and match JDBC-style parameters and named parameters in the same query.
- Queries are executed by calling **getResultList**() method of EntityManager.
- A query may be re-executed by subsequent invocations. Its lifespan is, however, bounded by the lifespan of the Session that created it.

Query Interface

Method	Meaning
int executeUpdate()	Executes the update or delete statement.
String[]getNamedParameters()	Returns the names of all named parameters of the query.
List list()	Returns the query results as a List.
set<Type>(int position, Type value) set<Type>(String name,Type value)	Assigns value to parameter of type <Type>
setComment(String comment)	Adds a comment to the generated SQL.
setFetchSize(int fetchSize)	Sets a fetch size for the underlying JDBC query.
setFirstResult(int firstResult)	Sets the first row to retrieve.
setMaxResults(int maxResults)	Sets the maximum number of rows to retrieve.
setParameter(int position, Object val)	Binds a value to a JDBC-style query parameter.
Object uniqueResult()	Returns a single instance that matches the query, or null if the query returns no results.

JPQL Examples

The following are the example of JQPL queries:

- `from Title`
- `from Title t where t.price> 500`
- `select title, price from Title order by price desc`
- `select title, t.publisher.name from Title`
- `from Title where title like '%and%'`
- `select max(price) from Title`
- `select t.publisher.name, max(t.price) from Title t group by t.publisher.name`
- `select upper(t.title)  || ' - ' ||  upper(t.publisher.name) from Title t`
- `select name from Publisher p where size(p.titles) > 2`
- `select title, price from Title where  price > ( select avg(price) from Title )`

HQL Examples

Processing results of a Query - Objects

When you do not use projection, JPA returns a collection of objects.

```
List result = em.createQuery("from Title").getResultList();
for(Object obj : result) {
  System.out.println(obj.toString());
}
```

Processing results of a Query - Fields

When you use projection with select then JPA returns an array of objects for each row.

```
List result = em.createQuery("select title, price from Title").getResultList();
for(Object obj : result) {
   Object [] values = (Object[]) obj;
   for (Object o : values) {
       System.out.print(o.toString() + "  ");
   }
}
```

Named Queries

- Define query at entity using @**NamedQuery** annotation.
- Use **createNamedQuery**() method to create a query from named query.

```
@Entity
@Table(name = "titles")
@NamedQuery(name = "CostlyTitles", query = "from Title where price > 15")
public class Title {
```

```
Query query = em.createNamedQuery("CostlyTitles", Title.class);
List<Title> titles = query.getResultList();
```

Native Query

- You may also express queries in the native SQL dialect of your database.
- JPA allows you to specify handwritten SQL (including stored procedures) for all create, update, delete, and load operations.
- Used for very complicated queries or taking advantage of some database features, like hints.

```
Query query =   em.createNativeQuery
                ("select * from titles where ytd_sales > 10000", Title.class);
List<Title> titles = query.getResultList();
```

Fetching

- A fetching strategy is the strategy Hibernate will use for retrieving associated objects if the application needs to navigate the association.
- Fetching strategy will have performance impact.
- Default fetch mode is **vulnerable to N+1 selects problem** where “N” is number of SELECTs used to retrieve the associated entity or collection.
- Unless you explicitly disable lazy fetching by specifying lazy="false", the subsequent select will only be executed when you actually access the association.
- Fetch strategies may be declared in the mapping files, or over-ridden by a particular HQL or Criteria query.

```
String cmd = "from Title t join fetch t.publisher as p";
Query query = em.createQuery(cmd);
List results = query.getResultList();
```

Concurrency – What is the problem?

- User A retrieves data of product 101 at 10:10:10
- User B retrieve data of product 101 at 10:10:11
- User A and B both modify the product information in the memory
- User B writes changes back at 10:10:13 and commits changes
- User A tries to write changes back at 10:10:14 and commit changes
- At this point there are two options:
  - Option 1: Allows user A to proceed overwriting changes made by user B
  - Option 2: Stop user A from making changes as data has changed since he retrieved the data

Optimistic vs. Pessimistic Locking

Optimistic locking

- Based on detecting changes on entities by checking their version attribute.
- No locking takes place at the database level.

Pessimistic locking

- Pessimistic locking mechanism involves locking entities on the database level.
- Each transaction can acquire a lock on data.
- As long as it holds the lock, no transaction can read, delete or make any updates on the locked data.

Optimistic Locking

- You can use version based optimistic concurrency control in which a number or timestamp is used to detect changes to row in the table.
- In order to use optimistic locking, we need to have an entity including a property with ***@Version*** annotation.
- While using it, each transaction that reads data holds the value of the version property.
- Before the transaction wants to make an update, it checks the version property again.

Version based concurrency

- Each entity instance has a **version**, which can be a number or timestamp.
- Persistence provider increments the version number whenever it makes changes to object.
- It uses version to detect changes and throws **OptimisticLockException** exception

```
@Entity
@Table(name = "accounts")
public class Account {
       // code
       @Version
       private Integer version;
      // code
}
```

Version based concurrency

```
update
        accounts
    set
        balance=?,
        customer=?,
        version=?       // updates version
    where
        acno=?
        and version=?   // checks whether retrieved version is same in table
```

Pessimistic Locking

- We can use a pessimistic lock to ensure that no other transactions can modify or delete reserved data.
- There are two types of locks we can retain: an exclusive lock and a shared lock. We could read but not write in data when someone else holds a shared lock. In order to modify or delete the reserved data, we need to have an exclusive lock.
- JPA specification defines three pessimistic lock modes
  - ✓ *PESSIMISTIC_READ* allows us to obtain a shared lock and prevent the data from being updated or deleted.
  - ✓ *PESSIMISTIC_WRITE* allows us to obtain an exclusive lock and prevent the data from being read, updated or deleted.
  - ✓ *PESSIMISTIC_FORCE_INCREMENT* works like *PESSIMISTIC_WRITE*, and it additionally increments a version attribute of a versioned entity.
- They all are retained until the transaction commits or rolls back.

```
entityManager.find(Product.class, prodid, LockModeType.PESSIMISTIC_READ);
```

```
entityManager.lock(product, LockModeType.PESSIMISTIC_WRITE); // lock explicitly
```