Preparation of the material was supported by the project „Increasing Internationality in Study Programs of the Department of Computer Science II", project number VP1–2.2–ŠMM-07-K-02-070, funded by The European Social Fund Agency and the Government of Lithuania.

Java Technologies

Lecture VIII

Valdas Rapševičius

Vilnius University
Faculty of Mathematics and Informatics

2014.05.15

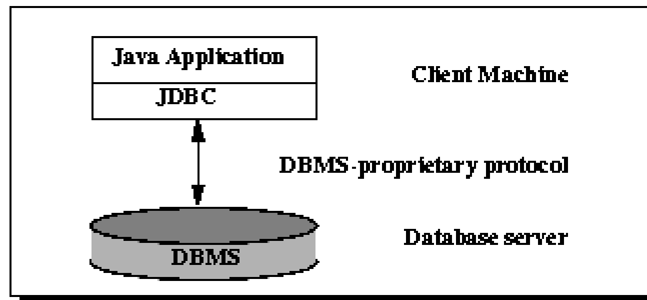
Session Outline

You will learn the Java Persistent APIs

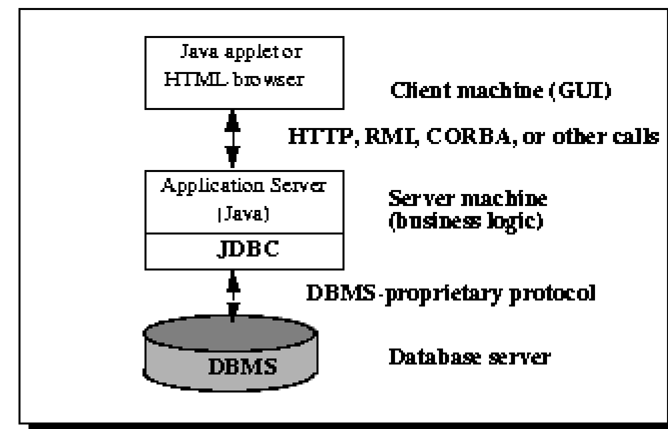
- JDBC (java.sql, javax.sql)
  - Connection
  - Statement
  - ResultSet
  - RowSet
- JPA (javax.persistence)
  - Mapping Entities
  - PersistenceUnit, EntityManager
  - Queries
    - JPQL
    - Criteria API (+ Metamodel)
  - Locking, Caching

Database

- Two-tier Architecture for Data Access.

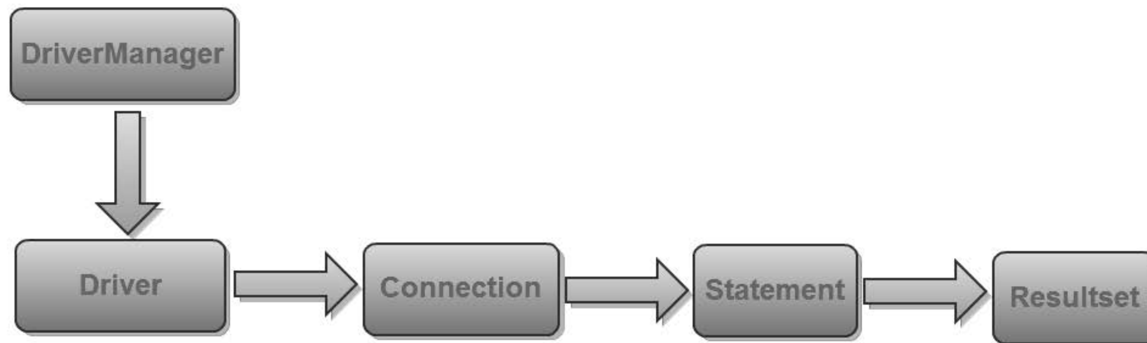


- Three-tier Architecture for Data Access.



JDBC

- The JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database
- JDBC helps you to write Java applications that manage these 3 programming activities:
  - Connect to a data source, like a database
  - Send queries and update statements to the database
  - Retrieve and process the results received from the database in answer to your query



JDBC: Main Interfaces (1)

`interface java.sql.Connection`

A connection (session) with a specific database. SQL statements are executed and results are returned within the context of a connection

`interface java.sql.Statement`

The object used for executing a static SQL statement and returning the results it produces

`public interface PreparedStatement extends Statement`

An object that represents a precompiled SQL statement

`public interface CallableStatement extends PreparedStatement`

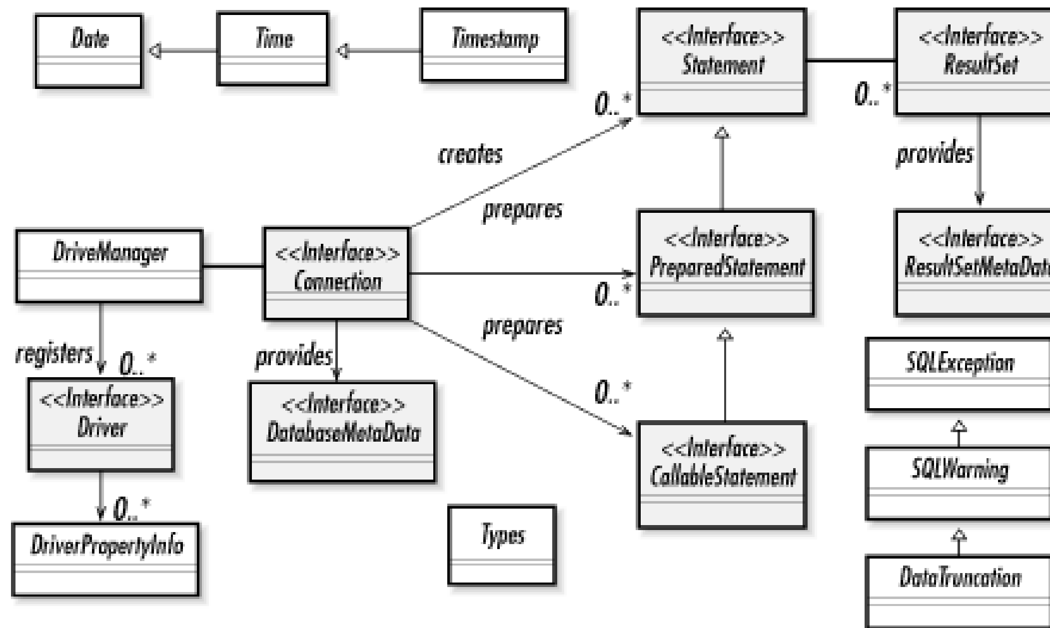
The interface used to execute SQL stored procedures

`interface java.sql.ResultSet`

A table of data representing a database result set, which is usually generated by executing a statement that queries the database

Subinterfaces: `CachedRowSet, FilteredRowSet, JdbcRowSet, JoinRowSet, RowSet, SyncResolver, WebRowSet`

JDBC: Main Classes (2)



JDBC: Connection

```
// Optional properties
Properties properties = new Properties();
properties.put("user", username);
properties.put("password", password);


// Lookup Driver class
Class.forName("com.mysql.jdbc.Driver").newInstance();


// Database specific URL
String url = "jdbc:mysql://localhost/test?";


// Connect to the database, use the Connection
try (Connection conn = DriverManager.getConnection(url)) { ... }
try (Connection conn = DriverManager.getConnection(url, username, password)) { ... }
try (Connection conn = DriverManager.getConnection(url, properties)) { ... }

// From the JNDI container
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/mydatabase");
try (Connection con = ds.getConnection(username, password)) { ... }
try (Connection con = ds.getConnection()) { ... }
```

ResultSet: Retrieve Values

```
String query = "select COF_NAME, PRICE, SALES, TOTAL from COFFEES";

// Open statement
try (Statement stmt = conn.createStatement()) {

  // Get result set
  try (ResultSet rs = stmt.executeQuery(query)) {

    // Loop results
    while (rs.next()) {

      String coffeeName = rs.getString("COF_NAME");
      Float price = rs.getFloat(2);
      Integer sales = rs.getInt("SALES");
      Integer total = rs.getInt("TOTAL");

      System.out.println(coffeeName + " " + price + " " + sales + " " + total);

    }
  }

} catch (SQLException e ) {

  System.printStackTrace(e);

}
```

ResultSet: Value Modification

```
String selectSql = "SELECT * FROM COFFEES";

// createStatement(int resultSetType, int resultSetConcurrency)

try (Statement stmt = conn.createStatement(
                      ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE)) {

  try (ResultSet rs = stmt.executeQuery(selectSql)) {

    while (rs.next()) {

      Float price = rs.getFloat("PRICE");
      rs.updateFloat("PRICE", f * 1.1);
      rs.updateRow();

    }
  }

} catch (SQLException e ) {
  System.printStackTrace(e);
}
```

ResultSet: New Row

```
String selectSql = "SELECT * FROM COFFEES";

try (Statement stmt = conn.createStatement(
          ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE)) {

  try (ResultSet rs = stmt.executeQuery(selectSql)) {

    rs.moveToInsertRow();
    rs.updateString("COF_NAME", coffeeName);
    rs.updateFloat("PRICE", price);
    rs.updateInt("SALES", sales);
    rs.updateInt("TOTAL", total);
    rs.insertRow();

    rs.beforeFirst();

  }

} catch (SQLException e ) {
  System.printStackTrace(e);
}
```

PreparedStatement

```
conn.setAutoCommit(false);

String updateSql = "update COFFEES set SALES=? where COF_NAME=?";

try (PreparedStatement stmt = conn.prepareStatement(updateString)) {

  // First update
  stmt.setInt(1, 100);
  stmt.setString(2, "French_Roast");
  stmt.executeUpdate();

  // Second update
  stmt.setInt(1, 101);
  stmt.setString(2, "Espresso");
  stmt.executeUpdate();

  conn.commit();

} catch (SQLException ex) {

  System.printStackTrace(ex);
  conn.rollback();

}
```

JDBC: Batch Update

```
conn.setAutoCommit(false);

try (Statement stmt = conn.createStatement()) {

  stmt.addBatch( "INSERT INTO COFFEES VALUES('Amaretto', 49, 9.99, 0, 0)");
  stmt.addBatch( "INSERT INTO COFFEES VALUES('Hazelnut', 49, 9.99, 0, 0)");
  stmt.addBatch( "INSERT INTO COFFEES VALUES('Amaretto_decaf', 49, 10.99, 0, 0)");
  stmt.addBatch( "INSERT INTO COFFEES VALUES('Hazelnut_decaf', 49, 10.99, 0, 0)");

  int [] updateCounts = stmt.executeBatch();
  conn.commit();

} catch(SQLException ex) {

  System.printStackTrace(ex);
  conn.rollback();

}
```

JDBC: RowSet

- A JDBC RowSet object holds tabular data in a way that makes it more flexible and easier to use than a result set
- Oracle has defined five RowSet interfaces for some of the more popular uses of a RowSet, and standard reference are available for these RowSet interfaces
- Programmers are free write their own versions of the javax.sql.RowSet interface, to extend the implementations of the five RowSet interfaces, or to write their own implementations

RowSet: Types

- **Connected** RowSet Objects
  - Only one of the standard RowSet implementations is a connected RowSet object: `JdbcRowSet`. Always being connected to a database, a JdbcRowSet object is most similar to a ResultSet object and is often used as a wrapper to make an otherwise non-scrollable and read-only ResultSet object scrollable and updatable.
- **Disconnected** RowSet Objects
  - A `CachedRowSet` object has all the capabilities of a JdbcRowSet object plus it can also do the following:
    - Obtain a connection to a data source and execute a query
    - Read the data from the resulting ResultSet object and populate itself with that data
    - Manipulate data and make changes to data while it is disconnected
    - Reconnect to the data source to write changes back to it
    - Check for conflicts with the data source and resolve those conflicts
  - A `WebRowSet` object has all the capabilities of a CachedRowSet object plus it can also do the following:
    - Write itself as an XML document
    - Read an XML document that describes a WebRowSet object
  - A `JoinRowSet` object has all the capabilities of a WebRowSet object (and therefore also those of a CachedRowSet object) plus it can also do the following:
    - Form the equivalent of a SQL JOIN without having to connect to a data source
  - A `FilteredRowSet` object likewise has all the capabilities of a WebRowSet object (and therefore also a CachedRowSet object) plus it can also do the following:
    - Apply filtering criteria so that only selected data is visible. This is equivalent to executing a query on a RowSet object without having to use a query language or connect to a data source.

RowSet: Create

```
JdbcRowSet rs = new JdbcRowSetImpl();
rs.setCommand("select * from COFFEES");
rs.setUrl("jdbc:myDriver:myAttribute");
rs.setUsername(username);
rs.setPassword(password);
rs.execute();

RowSetFactory rsf = RowSetProvider.newFactory();
JdbcRowSet rs = rsf.createJdbcRowSet();
rs.setUrl("jdbc:myDriver:myAttribute");
rs.setUsername(username);
rs.setPassword(password);
rs.setCommand("select * from COFFEES");
jdbcRs.execute();
```

RowSet: Operations

```
rs.absolute(3);
rs.updateFloat("PRICE", 10.99f);
rs.updateRow();

rs.moveToInsertRow();
rs.updateString("COF_NAME", "HouseBlend");
rs.updateFloat("PRICE", 7.99f);
rs.insertRow();

rs.last();
rs.deleteRow();

try {

  // Cached RowSet
  crs.acceptChanges();

} catch (SyncProviderException spe) {
  SyncResolver resolver = spe.getSyncResolver();
}
```

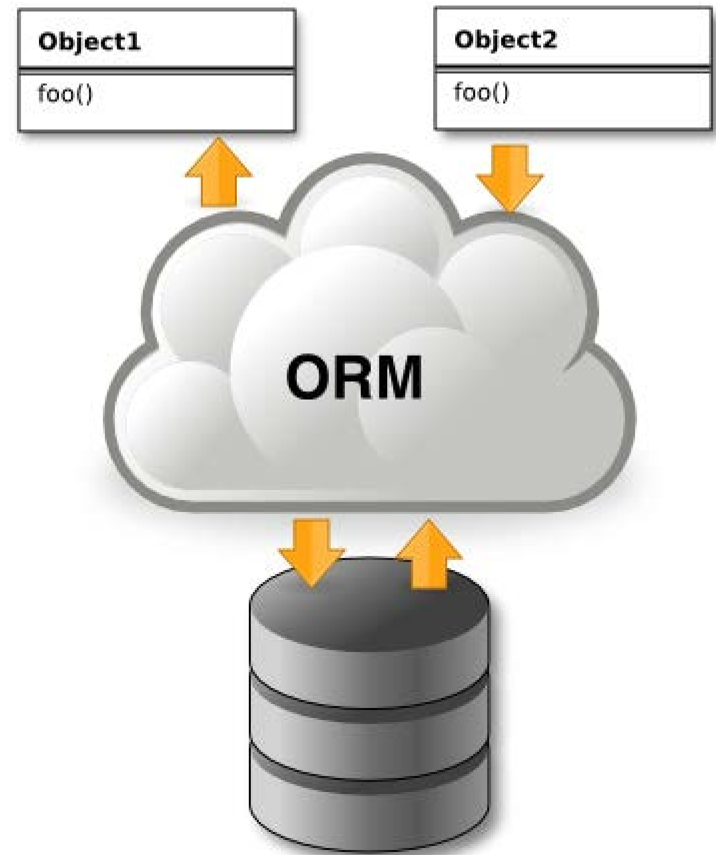
JDBC: Advanced Data Types

Advanced Data Type	**get*DataType* method**	**set*DataType* method**	**update*DataType* method**
BLOB	getBlob	setBlob	updateBlob
CLOB	getClob	setClob	updateClob
NCLOB	getNClob	setNClob	updateNClob
ARRAY	getArray	setArray	updateArray
XML	getSQLXML	setSQLXML	updateSQLXML
Structured type	getObject	setObject	updateObject
REF(structured type)	getRef	setRef	updateRef
ROWID	getRowId	setRowId	updateRowId
DISTINCT	getBigDecimal	setBigDecimal	updateBigDecimal
DATALINK	getURL	setURL	updateURL

ORM

- Object-relational mapping (ORM) is a software that provides services by converting data between incompatible type systems in OO programming languages.

- Java ORM JPA2 projects:
  - Hibernate (GLPL)
  - EclipseLink (EPL)
  - Oracle Toplink (Oracle)
  - Apache OpenJPA (ALv2.0)
  - MyBatis (ALv2.0)



JPA

The Java Persistence API provides Java developers with an object/relational mapping facility for managing relational data in Java applications

JPA 1.0 at May 2006 (JSR 220)
JPA 2.0 at December 2009 (JSR-000317)
JPA 2.1 at April 2013 (JSR-000338)

Java Persistence consists of 4 areas:

- Object/relational mapping metadata
- The Java Persistence API
- The Query language
- The Java Persistence Criteria API

Entity

- An entity is a lightweight persistence domain object. Typically, an entity represents a table in a relational database, and each entity instance corresponds to a row in that table. The primary programming artifact of an entity is the entity class, although entities can use helper classes.

- The persistent state of an entity is represented through either persistent fields or persistent properties. These fields or properties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store.

- An entity class must follow these requirements
  - The class must be annotated with the *javax.persistence.Entity* annotation.
  - The class must have *a public or protected, no-argument constructor*. The class may have other constructors
  - The class must *not be declared final*. No methods or persistent instance variables must be declared final
  - If an entity instance is passed by value as a detached object, such as through a session bean's remote business interface, the class must implement the *Serializable* interface
  - Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes
  - Persistent instance variables must be declared private, protected, or package-private and can be accessed directly only by the entity class's methods. Clients must access the entity's state through accessor or business methods.

Table Mapping

```
@Entity
@Table(name="CUST", schema="RECORDS")
public class Customer { ... }

@Entity
  @Table(name="CUST")
    @SecondaryTable(name="CUST_DETAIL",
        pkJoinColumns={
            @PrimaryKeyJoinColumn(name="CUST_ID"),
            @PrimaryKeyJoinColumn(name="CUST_TYPE")})
    public class Customer { ... }

@Entity
@NamedQuery(name="findAllCustomersWithName",
  query="SELECT * FROM CUST c WHERE c.name = :custName")
public class Customer { ... }
```

Entity: Primary Key

Every entity must have a primary key (PK). An entity may have either a simple or a composite primary key. Simple primary keys use the *javax.persistence.Id* annotation to denote the primary key property or field.

```
package lt.vu.mif.model;
import javax.persistence.*;

@Entity
@Table(name="FLIGHTS")
public class Flight implements Serializable {

     Long id;

     @Id
     @Column(name="FID")
     public Long getId() { return id; }
     public void setId(Long id) { this.id = id; }

}
```

Entity: Properties

Every non static non transient property (field or method depending on the access type) of an entity is considered persistent, unless you annotate it as @Transient. Not having an annotation for your property is equivalent to the appropriate @Basic annotation. The @Basic annotation allows you to declare the fetching strategy for a property:

```
private String firstname;

String getName() {... }

@Basic
@Column(name="NAME", nullable=false, length=128)
String getName() { ... } // persistent property

@Basic(fetch = FetchType.LAZY)
@Column(name="COMMENT", nullable=true, length=512)
private String comment;

@Temporal(TemporalType.TIME)
java.util.Date getDepartureTime() { ... }

//enum persisted as String in database
@Enumerated(EnumType.STRING)
Starred getGender() { ... }
```

Entity: Transient Properties

Every non static non transient property (field or method depending on the access type) of an entity is considered persistent, unless you annotate it as @Transient. Not having an annotation for your property is equivalent to the appropriate @Basic annotation. The @Basic annotation allows you to declare the fetching strategy for a property:

```
public transient int counter;

@Transient
private String comment;

@Transient
String getName() {... }
```

Multiplicity

- Multiplicities are of the following types:
  - **One-to-one**: Each entity instance is related to a single instance of another entity. For example, to model a physical warehouse in which each storage bin contains a single widget, StorageBin and Widget would have a one-to-one relationship. One-to-one relationships use the *javax.persistence.OneToOne* annotation on the corresponding persistent property or field
  - **One-to-many**: An entity instance can be related to multiple instances of the other entities. A sales order, for example, can have multiple line items. In the order application, Order would have a one-to-many relationship with LineItem. One-to-many relationships use the *javax.persistence.OneToMany* annotation on the corresponding persistent property or field.
  - **Many-to-one**: Multiple instances of an entity can be related to a single instance of the other entity. This multiplicity is the opposite of a one-to-many relationship. In the example just mentioned, the relationship to Order from the perspective of LineItem is many-to-one. Many-to-one relationships use the *javax.persistence.ManyToOne* annotation on the corresponding persistent property or field.
  - **Many-to-many**: The entity instances can be related to multiple instances of each other. For example, each college course has many students, and every student may take several courses. Therefore, in an enrollment application, Course and Student would have a many-to-many relationship. Many-to-many relationships use the *javax.persistence.ManyToMany* annotation on the corresponding persistent property or field.

Relationship Direction

Bidirectional Relationships

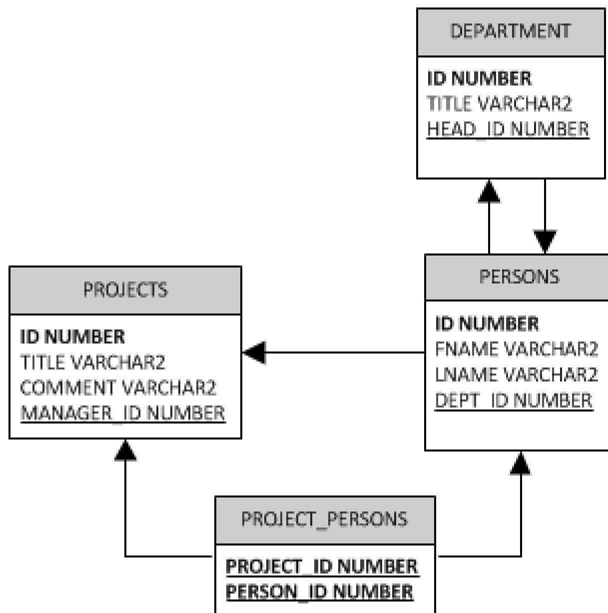
- In a **bidirectional** relationship, each entity has a relationship field or property that refers to the other entity. Through the relationship field or property, an entity class's code can access its related object

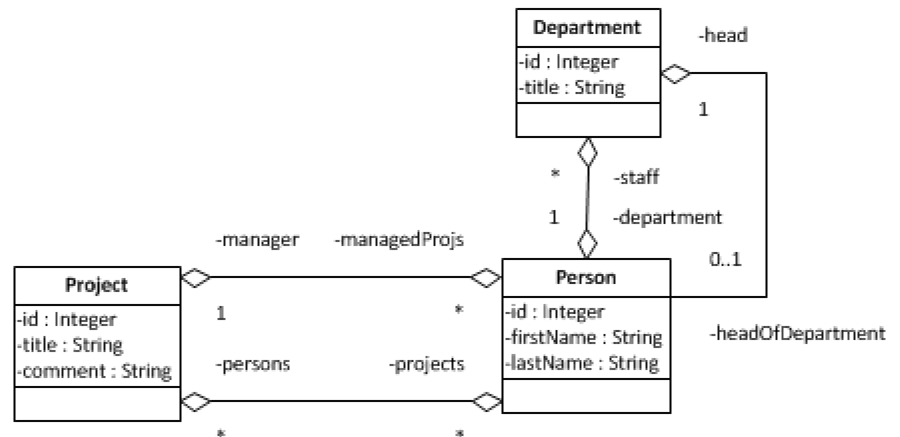
- Bidirectional relationships must follow these rules
  - The inverse side of a bidirectional relationship must refer to its owning side by using the mappedBy element of the *@OneToOne*, *@OneToMany*, or *@ManyToMany* annotation. The mappedBy element designates the property or field in the entity that is the owner of the relationship.
  - The many side of many-to-one bidirectional relationships must not define the mappedBy element. The many side is always the owning side of the relationship.
  - For one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.
  - For many-to-many bidirectional relationships, either side may be the owning side.

Unidirectional Relationships

- In a **unidirectional** relationship, only one entity has a relationship field or property that refers to the other. For example, LineItem would have a relationship field that identifies Product, but Product would not have a relationship field or property for LineItem. In other words, LineItem knows about Product, but Product doesn't know which LineItem instances refer to it.

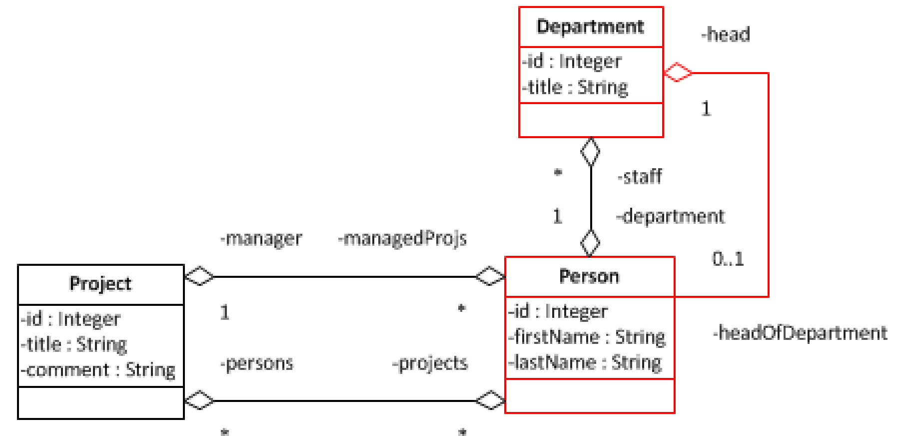
Example Schema

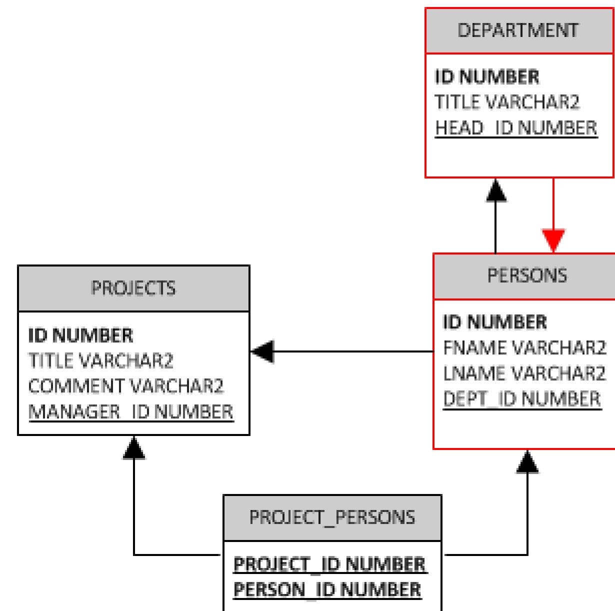




@OneToOne

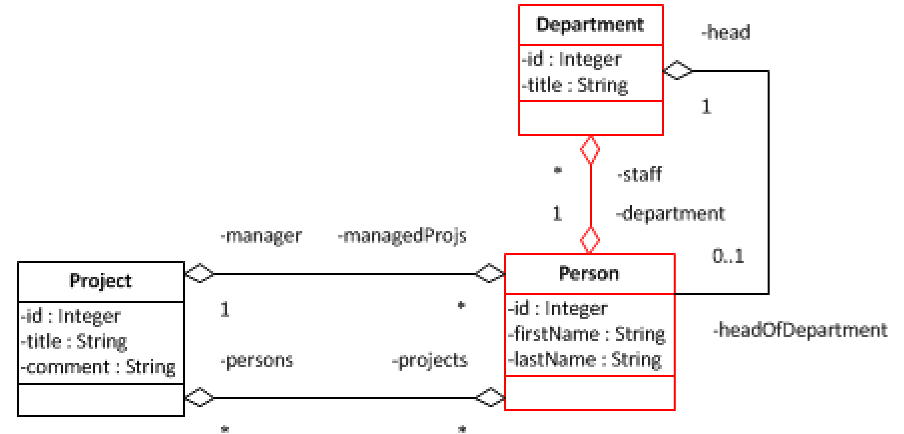
```
@Entity
@Table("DEPARTMENT")
public class Department {

  @Id
  @Column(name="ID")
  private Integer id;
  ...

  @OneToOne(fetch=FetchType.LAZY)
  @JoinColumn(name="HEAD_ID")
  private Person head;
  ...

}

@Entity
@Table("PERSONS")
public class Person {

  @Id
  private Integer id;
  ...

  @OneToOne(mappedBy="head")
  private Department headOfDepartment;
  ...

}
```

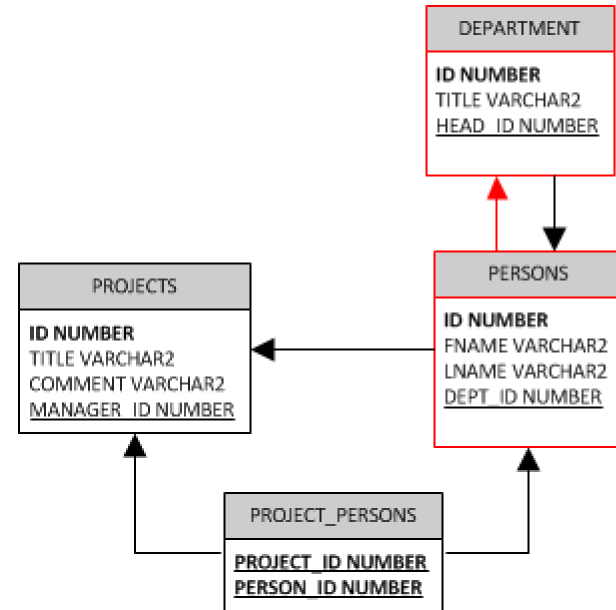




@OneToMany / @ManyToOne

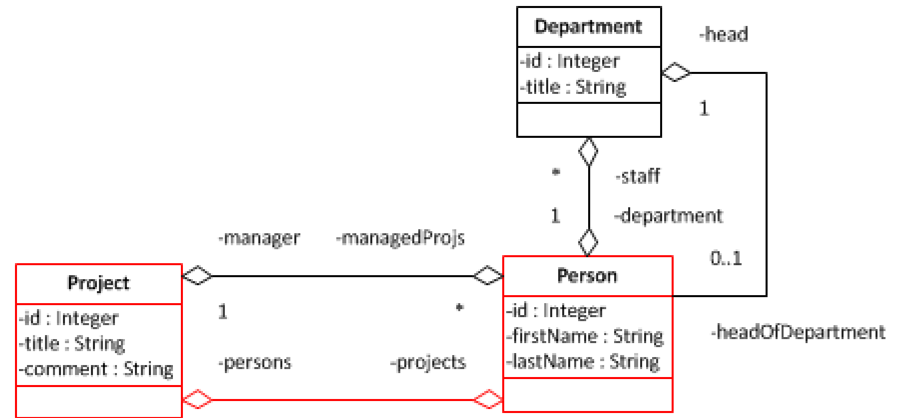
```
@Entity
@Table("DEPARTMENT")
public class Department {

  @Id
  @Column(name="ID")
  private Integer id;
  ...

  @OneToMany(mappedBy="department")
  private List<Person> staff;
  ...

}

@Entity
@Table("PERSONS")
public class Person {

  @Id
  private Integer id;
  ...

  @ManyToOne
  @JoinColumn(name="DEPT_ID")
  private Department department;
  ...

}
```

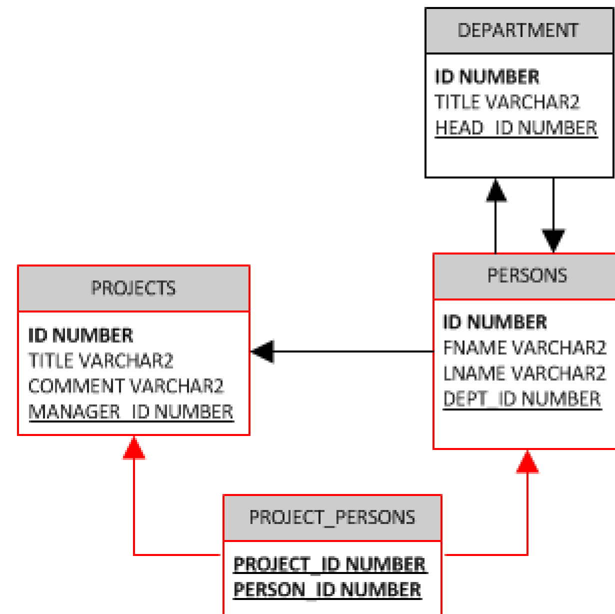




@ManyToMany

```
@Entity
@Table("PROJECTS")
public class Project {

  ...

  @ManyToMany
  @JoinTable(name="PROJECT_PERSONS",
    joinColumns={
      @JoinColumn(name="PROJECT_ID",
                  referencedColumnName="ID")},
    inverseJoinColumns={
      @JoinColumn(name="PERSON_ID",
                  referencedColumnName="ID")})
  private List<Person> persons;
  ...

}

@Entity
@Table("PERSONS")
public class Person {

  ...

  @ManyToMany(mappedBy="persons")
  private List<Project> projects;
  ...

}
```





Cascading Operations

The *javax.persistence.CascadeType* enumerated type defines the cascade operations that are applied in the cascade element of the relationship annotations.

Cascade Operation	Description
ALL	All cascade operations will be applied to the parent entity's related entity. All is equivalent to specifying cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}
DETACH	If the parent entity is detached from the persistence context, the related entity will also be detached.
MERGE	If the parent entity is merged into the persistence context, the related entity will also be merged.
PERSIST	If the parent entity is persisted into the persistence context, the related entity will also be persisted.
REFRESH	If the parent entity is refreshed in the current persistence context, the related entity will also be refreshed.
REMOVE	If the parent entity is removed from the current persistence context, the related entity will also be removed.

```
@OneToMany(cascade=REMOVE, mappedBy="customer")
public Set<Order> getOrders() { return orders; }
```

Embeddable

Embeddable classes are used to represent the state of an entity but don't have a persistent identity of their own, unlike entity classes. Instances of an embeddable class share the identity of the entity that owns it. Embeddable classes exist only as the state of another entity. An entity may have single-valued or collection-valued embeddable class attributes.

Embeddable classes have the same rules as entity classes but are annotated with the *javax.persistence.Embeddable* annotation instead of @Entity.

```
@Embeddable
public class ZipCode {
	String zip;
	String plusFour;
	...
}
```

```
@Entity
public class Address {
	@Id
	protected long id;
	String street;
	String city;
	@Embedded
	ZipCode zipCode;
	String country;
	...
}
```

Inheritance

Entity:

```
@Entity
public abstract class Employee {

        @Id
        protected Integer uid;
        ...

}

@Entity
public class FullTimeEmployee extends Employee {

        protected Integer salary;
        ...

}

@Entity
public class PartTimeEmployee extends Employee {

        protected Float hourlyWage;
        ...

}
```

Not entity:

```
@MappedSuperclass
public class Employee {

        @Id
        protected Integer employeeId;
        ...

}

@Entity
public class FullTimeEmployee extends Employee {

        protected Integer salary;
        ...

}

@Entity
public class PartTimeEmployee extends Employee {

        protected Float hourlyWage;     ...

}
```

Inheritance Strategy

You can configure how the Java Persistence provider maps inherited entities to the underlying datastore by decorating the root class of the hierarchy with the annotation *javax.persistence.Inheritance*. The following mapping strategies are used to map the entity data to the underlying database:

- The Single Table per Class Hierarchy Strategy

  With this strategy, which corresponds to the default *InheritanceType.SINGLE_TABLE*, all classes in the hierarchy are mapped to a single table in the database. This table has a **discriminator column** containing a value that identifies the subclass to which the instance represented by the row belongs.

- The Table per Concrete Class Strategy

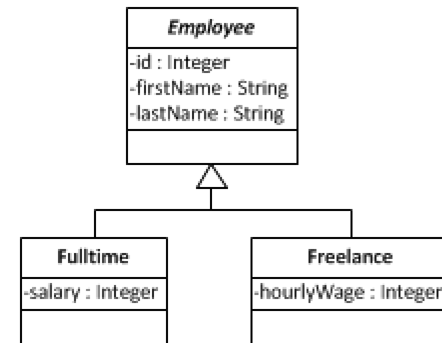
  In this strategy, which corresponds to *InheritanceType.TABLE_PER_CLASS*, each concrete class is mapped to a separate table in the database. All fields or properties in the class, including inherited fields or properties, are mapped to columns in the class's table in the database.

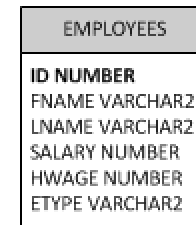
- The Joined Subclass Strategy

  In this strategy, which corresponds to *InheritanceType.JOINED*, the root of the class hierarchy is represented by a single table, and each subclass has a separate table that contains only those fields specific to that subclass. That is, the subclass table does not contain columns for inherited fields or properties. The subclass table also has a column or columns that represent its primary key, which is a foreign key to the primary key of the superclass table.

InheritanceType.SINGLE_TABLE

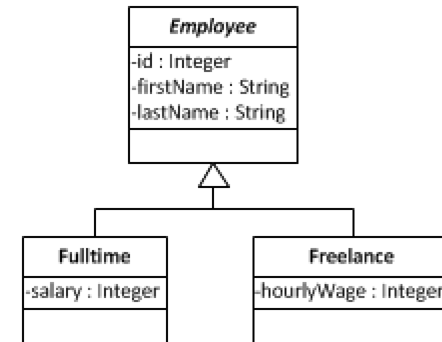
```
@Entity
@Table("EMPLOYEES")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="ETYPE")
public abstract class Employee {

  ...

  @Id
  private Integer id;
  ...

}

@Entity
@DiscriminatorValue("FT")
public class Fulltime extends Employee {

  ...

}

@Entity
@DiscriminatorValue("FL")
public class Freelance extends Employee {

  ...

}
```

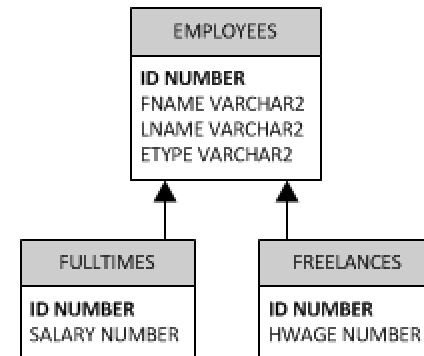




InheritanceType.JOINED

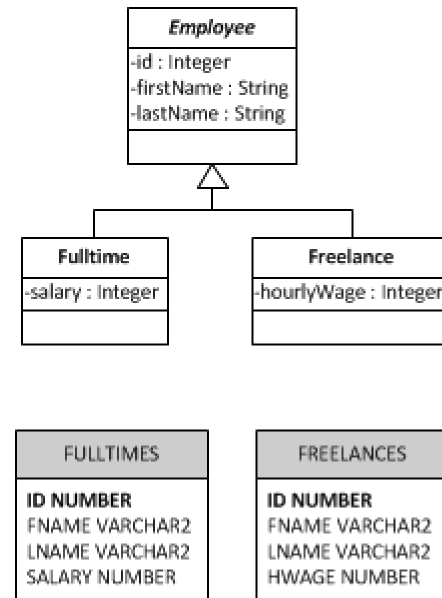
```
@Entity
@Table("EMPLOYEES")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="ETYPE")
public abstract class Employee {

  ...

  @Id
  private Integer id;
  ...

}

@Entity
@DiscriminatorValue("FT")
@Table(name="FULLTIMES")
public class Fulltime extends Employee {

  ...

}

@Entity
@DiscriminatorValue("FL")
@Table(name="FREELANCES")
public class Freelance extends Employee {

  ...

}
```



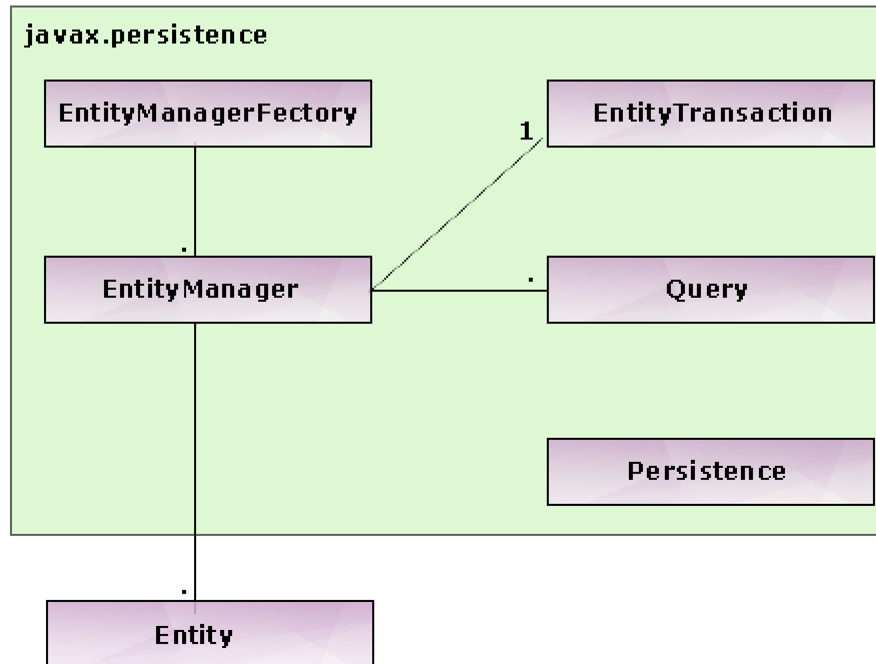


InheritanceType.TABLE_PER_CLASS

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Employee {

  ...

  @Id
  private Integer id;
  ...

}

@Entity
@Table(name="FULLTIMES")
public class Fulltime extends Employee {

  ...

}

@Entity
@Table(name="FREELANCES")
public class Freelance extends Employee {

  ...

}
```



JPA Architecture



Persistence Unit

- A persistence unit defines a set of all entity classes that are managed by EntityManager instances in an application. This set of entity classes represents the data contained within a single data store.
- Persistence units are defined by the `persistence.xml` configuration file

```
<persistence>
  <persistence-unit name="OrderManagement">

    <description>This unit manages orders and customers.
      It does not rely on any vendor-specific features and
      can therefore be deployed to any persistence provider.
    </description>

    <jta-data-source>jdbc/MyOrderDB</jta-data-source>
    <jar-file>MyOrderApp.jar</jar-file>
    <class>com.widgets.Order</class>
    <class>com.widgets.Customer</class>

  </persistence-unit>
</persistence>
```

EntityManager

The *javax.persistence.EntityManager* API creates and removes persistent entity instances, finds entities by the entity's primary key, and allows queries to be run on entities.

- Container-Managed Entity Managers

  An EntityManager instance's persistence context is automatically propagated by the container to all application components that use the EntityManager instance within a single Java Transaction API (JTA) transaction.
  To obtain an EntityManager instance, inject the entity manager into the application component:

```
@PersistenceContext
EntityManager em;
```

- Application-Managed Entity Managers

  With an application-managed entity manager, on the other hand, the persistence context is not propagated to application components, and the lifecycle of EntityManager instances is managed by the application.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("JpaTest");
EntityManager em = emf.createEntityManager();
…
em.close();
emf.close();
```

Managing Entities

```
@PersistenceContext
EntityManager em;
```

- Finding Entities Using the EntityManager

```
Customer cust = em.find(Customer.class, custID);
```

- Persisting Entity Instances

```
Customer cust = new Customer();
em.persist(cust);
```

- Removing Entity Instances

```
Order order = em.find(Order.class, orderId);
em.remove(order);
```

- Synchronizing Entity Data to the Database

To force synchronization of the managed entity to the data store, invoke the flush method of the EntityManager instance. If the entity is related to another entity and the relationship annotation has the cascade element set to PERSIST or ALL, the related entity's data will be synchronized with the data store when flush is called. If the entity is removed, calling flush will remove the entity data from the data store.

Querying Entities

The Java Persistence API provides the following methods for querying entities.

- The Java Persistence query language (JPQL) is a simple, string-based language similar to SQL used to query entities and their relationships.
- The Criteria API is used to create typesafe queries using Java programming language APIs to query for entities and their relationships.

Both JPQL and the Criteria API have advantages and disadvantages.

- Just a few lines long, JPQL queries are typically more concise and more readable than Criteria queries. Developers familiar with SQL will find it easy to learn the syntax of JPQL. JPQL named queries can be defined in the entity class using a Java programming language annotation or in the application's deployment descriptor. JPQL queries are not typesafe, however, and require a cast when retrieving the query result from the entity manager. This means that type-casting errors may not be caught at compile time. JPQL queries don't support open-ended parameters.
- Criteria queries allow you to define the query in the business tier of the application. Although this is also possible using JPQL dynamic queries, Criteria queries provide better performance because JPQL dynamic queries must be parsed each time they are called. Criteria queries are typesafe and therefore don't require casting, as JPQL queries do. The Criteria API is just another Java programming language API and doesn't require developers to learn the syntax of another query language. Criteria queries are typically more verbose than JPQL queries and require the developer to create several objects and perform operations on those objects before submitting the query to the entity manager.

JPQL Examples

select_statement ::= select_clause from_clause
[where_clause][groupby_clause][having_clause][orderby_clause]
update_statement :: = update_clause [where_clause]
delete_statement :: = delete_clause [where_clause]

```
em.createQuery("SELECT p FROM Player p").getResultList();

em.createQuery("SELECT c FROM Customer c WHERE c.name LIKE :custName")
                .setParameter("custName", name) .getResultList();

SELECT DISTINCT p1 FROM Player p1, Player p2 WHERE p1.salary > p2.salary
AND p2.name = :name

UPDATE Player p SET p.status = 'inactive' WHERE p.lastPlayed <
:inactiveThresholdDate

DELETE FROM Player p WHERE p.status = 'inactive' AND p.teams IS EMPTY
```

Criteria API

```
//
// Equivalent to SELECT p FROM Pet p
//

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);
Root<Pet> pet = cq.from(Pet.class);
cq.select(pet);

TypedQuery<Pet> q = em.createQuery(cq);

List<Pet> allPets = q.getResultList();
```

Using Metamodel

```
CriteriaBuilder cb = em.getCriteriaBuilder();
Metamodel m = em.getMetamodel();

CriteriaQuery<Pet> cq = cb.createQuery(Pet.class);

Root<Pet> pet = cq.from(Pet.class);
Date firstDate = new Date(...);
Date secondDate = new Date(...);

// With Metamodel classes

cq.where(cb.equal(pet.get("name"), "Fido"));
cq.where(cb.between(pet.get(Pet_.birthDate), firstDate, secondDate));
cq.where(cb.like(pet.get(Pet_.name), "*do"));

// Without Metamodel classes

EntityType<Pet> Pet_ = m.entity(Pet.class);
cq.where(cb.between(
        pet.get(Pet_.getSingularAttribute("birthDate", Date.class)),
        firstDate, secondDate));
cq.where(cb.like(pet.get(Pet_. getSingularAttribute("name",
String.class)),"*do"));
```

Locking

Entity data is *concurrently accessed* if the data in a data source is accessed at the same time by multiple applications. Special care must be taken to ensure that the underlying data's integrity is preserved when accessed concurrently.

- Optimistic Locking

  By default, persistence providers use optimistic locking, where, before committing changes to the data, the persistence provider checks that no other transaction has modified or deleted the data since the data was read. This is accomplished by a version column in the database table, with a corresponding version attribute in the entity class. When a row is modified, the version value is incremented. The original transaction checks the version attribute, and if the data has been modified by another transaction, a javax.persistence.OptimisticLockException will be thrown, and the original transaction will be rolled back. When the application specifies optimistic lock modes, the persistence provider verifies that a particular entity has not changed since it was read from the database even if the entity data was not modified.

  The javax.persistence.Version annotation is used to mark a persistent field or property as a version attribute of an entity. By adding a version attribute, the entity is enabled for optimistic concurrency control. The version attribute is read and updated by the persistence provider when an entity instance is modified during a transaction. The application may read the version attribute, but must not modify the value.

```
@Version
protected int version;
```

- Pessimistic Locking

  Pessimistic locking goes further than optimistic locking. With pessimistic locking, the persistence provider creates a transaction that obtains a long-term lock on the data until the transaction is completed, which prevents other transactions from modifying or deleting the data until the lock has ended. Pessimistic locking is a better strategy than optimistic locking when the underlying data is frequently accessed and modified by many transactions.

  Caution - Using pessimistic locks on entities that are not subject to frequent modification may result in decreased application performance.

Lock Modes

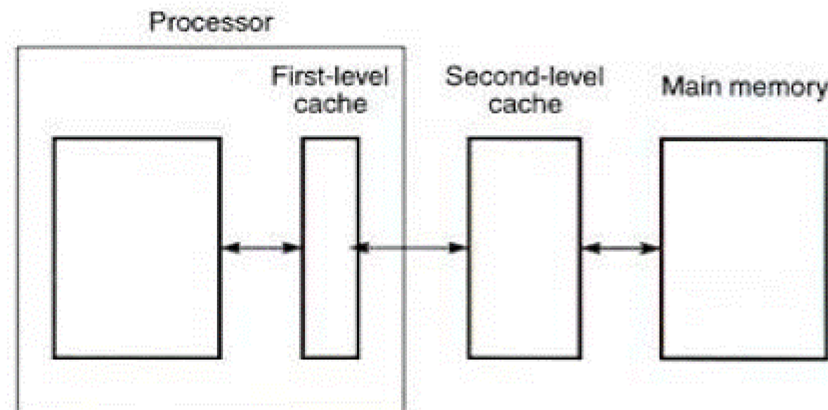
```
EntityManager em = ...;
Person person = ...;
em.lock(person, LockModeType.OPTIMISTIC);

Person person = em.find(Person.class, personPK, LockModeType.PESSIMISTIC_WRITE);
Person person = em.find(Person.class, personPK);
em.refresh(person, LockModeType.OPTIMISTIC_FORCE_INCREMENT);
Query q = em.createQuery(...);
q.setLockMode(LockModeType.PESSIMISTIC_FORCE_INCREMENT);
@NamedQuery(name="lockPersonQuery", query="SELECT p FROM Person p WHERE p.name LIKE
:name", lockMode=PESSIMISTIC_READ)
```

Lock Mode	**Description**
OPTIMISTIC	Obtain an optimistic read lock for all entities with a version attribute.
OPTIMISTIC_FORCE_INCREMENT	Obtain an optimistic read lock for all entities with a version attribute, and increment the version attribute value.
PESSIMISTIC_READ	Immediately obtain a long-term read lock on the data to prevent the data from being modified or deleted. Other transactions may read the data while the lock is maintained, but may not modify or delete the data. The persistence provider is permitted to obtain a database write lock when a read lock was requested, but not vice versa.
PESSIMISTIC_WRITE	Immediately obtain a long-term write lock on the data to prevent the data from being read, modified, or deleted.
PESSIMISTIC_FORCE_INCREMENT	Immediately obtain a long-term lock on the data to prevent the data from being modified or deleted, and increment the version attribute of versioned entities.
READ	A synonym for OPTIMISTIC. Use of LockModeType.OPTIMISTIC is to be preferred for new applications.
WRITE	A synonym for OPTIMISTIC_FORCE_INCREMENT. Use of LockModeType.OPTIMISTIC_FORCE_INCREMENT is to be preferred for new applications.
NONE	No additional locking will occur on the data in the database.

Second-level Cache

- A second-level cache is a local store of entity data managed by the persistence provider to improve application performance. A second-level cache helps improve performance by avoiding expensive database calls, keeping the entity data local to the application. A second-level cache is typically transparent to the application, as it is managed by the persistence provider and underlies the persistence context of an application. That is, the application reads and commits data through the normal entity manager operations without knowing about the cache

- Note: Persistence providers are not required to support a second-level cache. Portable applications should not rely on support by persistence providers for a second-level cache.



Cache Mode

- Persistence Unit:

```
<persistence-unit name="examplePU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>jdbc/__default</jta-data-source>
    <shared-cache-mode>DISABLE_SELECTIVE</shared-cache-mode>
</persistence-unit>
```

- EntityManagerFactory:

```
EntityManagerFactor emf =
    Persistence.createEntityManagerFactory("myExamplePU",
                new Properties()
                .add("javax.persistence.sharedCache.mode","ENABLE_SELECTIVE"));
```

Cache Mode Setting	Description
ALL	All entity data is stored in the second-level cache for this persistence unit.
NONE	No data is cached in the persistence unit. The persistence provider must not cache any data.
ENABLE_SELECTIVE	Enable caching for entities that have been explicitly set with the @Cacheable annotation.
DISABLE_SELECTIVE	Enable caching for all entities except those that have been explicitly set with the @Cacheable(false) annotation.
UNSPECIFIED	The caching behavior for the persistence unit is undefined. The persistence provider's default caching behavior will be used.

@Cacheable

```
@Cacheable(true)
@Entity
public class Person{ ... }

@Cacheable(false)
@Entity
public class Person{ ... }

@Cacheable(timeToLive=30)
@Entity
public class OrderStatus { ... }
```

Cache Control

```
EntityManager em = ...;
em.setProperty("javax.persistence.cache.storeMode", "BYPASS");

Map<String, Object> props = new HashMap<String, Object>();
props.put("javax.persistence.cache.retrieveMode", "BYPASS");
String personPK = ...;
Person person = em.find(Person.class, personPK, props);

CriteriaQuery<Person> cq = ...;
TypedQuery<Person> q = em.createQuery(cq);
q.setHint("javax.persistence.cache.storeMode", "REFRESH");

Cache cache = em.getEntityManagerFactory().getCache();
if (cache.contains(Person.class, personPK)) {
        // the data is cached
} else {
        // the data is NOT cached
}

cache.evict(Person.class, personPK);

cache.evict(Person.class);

cache.evictAll();
```

Session Conclusions

- Old plain JDBC
  - JDBC Drivers (version, provider)
  - Connection
  - Statement
  - ResultSet, RowSet
- JPA!
  - Mapping Entities
    - Aggregation
    - Inheritence
  - FetchType, Cascade
  - PersistenceUnit
    - EntityManagerFactory
    - EntityManager
  - Queries
    - JPQL
    - Criteria API (+ Metamodel)
  - Locking, Caching