

ID2212 Network Programming with Java

Lecture 9

Java Database Connectivity (JDBC)

Java Persistence API (JPA)

Leif Lindbäck and Vladimir Vlassov

KTH/ICT/SCS

HT 2015

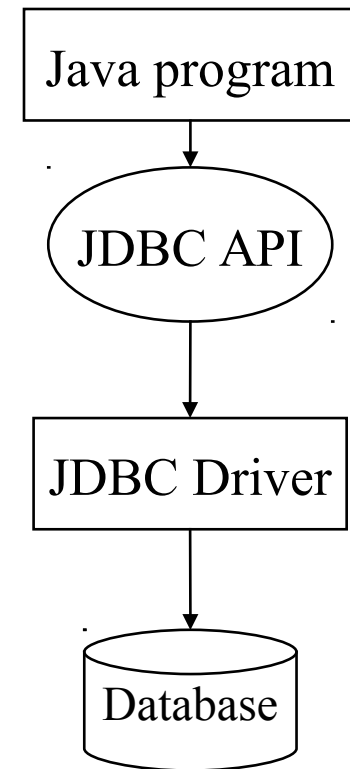
JDBC: Java Database Connectivity

`java.sql`

<https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/index.html>

Java Database Connectivity (JDBC)

- An API for unified connectivity to relational databases
 - Establish a connection with a data source
 - Execute SQL queries on the data source
 - Get and process results



Database

- A *database* is essentially a smart container for tables.
- A *table* is a named container comprised of rows.
- A *row* is (conceptually) a container comprised of columns.
- A *column* is a single data item having a name, type, and value.

SQL

- SQL (Structured Query Language)
 - An industry-standard language for creating, updating and, querying relational DBMS.
 - Developed by IBM in the 1970s
 - A single SQL statement can be very expressive and can initiate high-level actions, such as sorting and merging.

SQL Primer

- Create a table in SQL:

```
CREATE TABLE <table name> (<column element> [,  
    <column element>]...)
```

- where **column element** is of the form:

```
<column name> <data type> [DEFAULT <expression>] [<column  
    constraint> [, <column constraint>]...]
```

- where **column constraint** is of the form:

```
NOT NULL | UNIQUE | PRIMARY KEY
```

- Example:

```
CREATE TABLE participants (ID char(5), NAME char(64), GENDER  
    char(1), COUNTRY char(32), BIRTHDAY date, HEIGHT double,  
    WEIGHT double, SUBJECT char(32));
```

- Drop a table:

```
DROP TABLE <table name>
```

SQL Primer (cont)

- Retrieve a set of columns from one or more tables:

```
SELECT [ALL | DISTINCT] <select list> FROM <table reference list>  
WHERE <search condition list> [ORDER BY <column designator>  
[ASC | DESC] [, <column designator> [ASC | DESC]]...]
```

– Example:

```
SELECT NAME, COUNTRY from participants WHERE GENDER='F';
```

SQL Primer (cont)

- Insert rows:

```
INSERT INTO <table name> [(<column name> [, <column name>]...)] VALUES (<expression> [, <expression>]...)
```

– Example:

```
INSERT INTO participants VALUES (50044, 'Wahlström, Robert', 'M', 'Sweden', 1979-05-03, 177.0, 61.0, 'Skijumping');
```

- Update rows:

```
UPDATE <table name> SET <column name = {<expression> | NULL}> [, <column name = {<expression> | NULL}>]... WHERE <search condition>
```

- Delete rows:

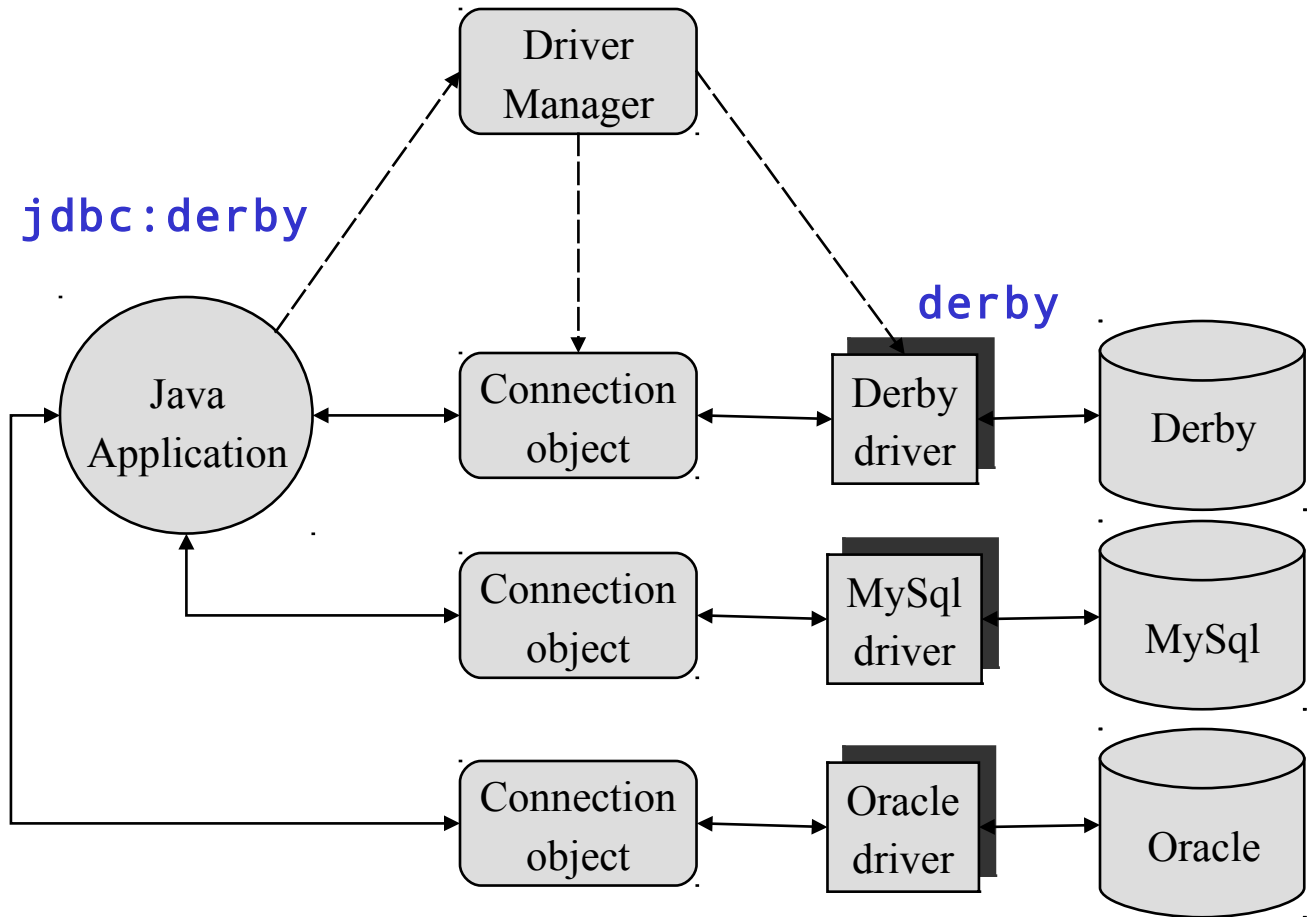
```
DELETE FROM <table name> WHERE <search condition>
```


JDBC Code Fragment

```
// Connect to the data source
Connection connection =
DriverManager.getConnection(
    "jdbc:derby://localhost:1527/mydb", "user",
    "pass");
// Create SQL statement
Statement stmt = connection.createStatement();
// Send a query to the data source, get results
ResultSet rs = stmt.executeQuery("SELECT a, b, c
FROM Table1");
// Process results
    while (rs.next()) {
        int x = rs.getInt("a");
        String s = rs.getString("b");
        float f = rs.getFloat("c");
    }
```

JDBC Programming Concepts

- Data Source
- Driver
- Driver Manager
- Connection
- Statement
- Result set
- Metadata
- Transactions



A Data Source

- A database, a file system, a (tab-separated-value) file.
- A data source is pointed to by an URL of the form **`jdbc:<Sub-Protocol>:<Datasource-Name>`**
 - For example:
 - **`jdbc:derby://localhost:1527/myDataBase`**
 - **`jdbc:mysql://localhost:3306/myDataBase`**
 - A sub-protocol name
 - indicates the type of data source, e.g. **`derby`**
 - defines a driver to handle the data source.
 - used by DriverManager to lookup a driver.
- User name and password might be required to connect to the data source.

A JDBC Driver

- An object that opens connection to a data source and handles the connection.
- A JDBC driver class
 - implements the JDBC **Driver** interface and can convert program (and typically SQL) requests for a particular database.
- Loading a driver class. Two options
 - Put name of the driver in the jdbc.drivers System property, e.g.
`jdbc.drivers=org.apache.derby.jdbc.ClientXADataSource`
 - Will be checked by the DriverManager
 - Load class explicitly, e.g.
`Class.forName("org.apache.derby.jdbc.ClientXADataSource");`

Driver Manager

- Driver Manager
 - Parses URL of a data source, look for a driver to handle the source, returns a Connection object, e.g.

```
Connection connection =  
    DriverManager.getConnection (url,  
        username, password);
```

Connection

- Represents a session with a data source.
- Used
 - to create and prepare SQL statements and calls,
 - to retrieve the meta data regarding the connection's database,
 - to commit or to drop (rollback) all changes made to the connection's database.
- Any number of SQL statements can be executed over the connection.
- An application can have one or more connections to a single data source or to several databases.

Connection (cont)

- To connect to a data source, you supply the following information
 - URL of a data source (database, file),
 - class names of drivers,
 - user name and password (both are optional)
- This info can be
 - “hard-coded” in the code,
 - passed as arguments to the application,
 - loaded as Properties at run time from a configuration file of the form:

```
jdbc.drivers=org.apache.derby.jdbc.ClientXADataSource  
jdbc.url=jdbc:derby://localhost:1527/myDataBase  
db.username=user  
db.password=pass
```

Using Configuration Properties.

Connecting to a Data Source.

```
FileInputStream in = new FileInputStream(configFileName);
Properties props = new Properties();
props.load(in);

String drivers = props.getProperty("jdbc.drivers");
System.setProperty("jdbc.drivers", drivers);
String url = props.getProperty("jdbc.url");
String username = props.getProperty("jdbc.username");
String password = props.getProperty("jdbc.password");
in.close();
Connection connection =
    DriverManager.getConnection(url, username, password);
```


Database Meta-Data

- If necessary, query the Connection for meta-data about the database structure:
 - tables, supported SQL grammar, stored procedures,
 - capabilities of the connection (e.g. supported isolation levels), etc.

```
DatabaseMetaData dbm = connection.getMetaData();
```

- The DatabaseMetaData interface defines various get and checking methods, e.g.

```
ResultSet rs =  
    dbm.getTables(null, null, null, null);  
System.out.println("Table Name\tTable Type");  
while (rs.next()) {  
    System.out.println(rs.getString(3) + "\t" +  
        rs.getString(4));  
}
```

Statement

- Create a SQL statement object from the Connection object for sending commands and SQL statements to the data source.
 - Statement is like an envelope for SQL,
 - Connection is like the transport to deliver the statement to the driver,
 - The driver forwards the SQL to the database and returns results.

Statement (cont'd)

- Create a statement using the `Connection` object
 - `createStatement()`
 - Creates a `Statement` object for sending SQL statements to the database
 - `prepareStatement(String sql)`
 - Creates a `PreparedStatement` object for sending parameterized SQL statements to the database.
 - `prepareCall(String sql)`
 - Creates a `CallableStatement` object for calling stored procedures.

Executing A Statement

- Four methods of **Statement** for sending SQL to the database and executing database calls:
 - **ResultSet executeQuery(String sql)**
 - Executes an SQL statement that returns a single ResultSet object.
 - **int executeUpdate(String sql)**
 - Executes an SQL INSERT, UPDATE or DELETE statement.

Result Set

- Result Set

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

```
Statement stmt = con.createStatement();  
ResultSet rs =  
    stmt.executeQuery("SELECT a, b FROM TABLE2");
```

- Organized into logical rows and columns of data.
- Maintains a cursor to a current row

Result Set (cont'd)

- The **ResultSet** interface contains methods for
 - getting values from the set by name or position,
 - traversing to the next, previous, first, and last row of the set,
 - deleting current row, jumping to the insert row, and so on,
 - getting result set meta-data.

Result Set Meta-Data

- Result set meta-data

- number of columns, names and types of columns.

Get from the result set of an execute method. For example:

```
ResultSetMetaData rsmd = rs.getMetaData();
int columnCount = rsmd.getColumnCount();
// Iterate through the columns
// and print each column name
for (int i = 1; i <= columnCount; i++) {
    String columnName = rsmd.getColumnName(i);
    System.out.print(columnName + "\t");
}
System.out.println("");
```

Iterating Though A Result Set

```
// Execute a SELECT query, get result set and
// meta-data.
ResultSet rs = stmt.executeQuery(sqlStr);
ResultSetMetaData rsmd = rs.getMetaData();
// Get the column count.
int columnCount = rsmd.getColumnCount();
// Iterate through each row printing the values.
// Print a $ if the column type is CURRENCY.
while (rs.next()) {
    for (int i =1; i <= columnCount; i++) {
        if (rsmd.getColumnTypeName(i).
            equals("CURRENCY")) {
            System.out.print("$");
        }
        System.out.print(rs.getString(i) +"\t");
    }
    System.out.println("");
}
```


PreparedStatement

- Represents a precompiled SQL statement prepared using the Connection object.

```
PreparedStatement pstmt = con.prepareStatement(
    "UPDATE EMPLOYEES SET SALARY = ? WHERE ID = ?");
pstmt.setBigDecimal(1, 153833.00);
pstmt.setInt(2, 110592);
int insCount = pstmt.executeUpdate();
System.out.println( "Updated " + insCount + "rows");
```

PreparedStatement, Cont'd

- **PreparedStatement** has the following advantages above **Statement**:
- Faster execution since the statement is not interpreted and compiled at each call.
- More secure since SQL injection is not possible when using a prepared statement.

Summary

- Steps for accessing and working with a data source
 - Load (specify) a JDBC driver, URL of the source, username and password
 - Create a connection to the data source pointed to by the URL:

```
Connection con =  
    DriverManager.getConnection(url, user, password)
```

- If necessary query the Connection for meta-data about the database:

```
DatabaseMetaData dbm = con.getMetaData();
```

- Create a SQL statement from the connection

```
Statement stmt = con.createStatement();
```

- Use the statement object to execute SQL query(ies)

```
ResultSet rs =  
    stmt.executeQuery("SELECT a, b FROM TABLE2");
```

- Check for SQLWarning, if any, or ignore

```
SQLWarning warning = stmt.getWarnings();
```

- Get and process the results from the query

- Finally close the database connection: `con.close();`

Transactions

- A transaction is a group of operations that are:
 - **Atomic**, either all or no of the operations are performed.
 - **Consistent**, The data is left in a valid state.
 - **Isolated**, transactions do not affect each other even if they are concurrent.
 - **Durable**, once a transaction has finished the data is saved, no matter what happens afterwards.
- These four properties are referred to as ACID.

Transactions, Cont'd

- There are two operations that can end a transaction:
 - **Commit**, all changes made during the transaction are saved permanently.
 - **Rollback**, All changes made during the transaction are undone and the data is left in the same state it had before the transaction started.

Auto Commit

- By default, an active database connection is set to *auto commit*
 - all connection's SQL statements are executed and committed as individual transactions.
 - The commit occurs when the statement completes or the next execute occurs,
 - If a statement returns a ResultSet, the statement completes when the last row of the ResultSet has been retrieved or the ResultSet has been closed.

Managing Transactions

- To enable/disable auto commit, call on the **Connection**
 - **setAutoCommit(boolean)**
- If auto commit is disabled, call on the **Connection**
 - **rollback()**
 - To drop all changes made since the previous commit/rollback and releases any database locks currently held by the Connection.
 - **commit()**
 - To make all changes made since the previous commit/rollback permanent and releases any database locks currently held by the Connection.

Transaction Isolation Level

- Specifies to which extent transactions avoid sharing data. Different isolation levels allow different sets of the following phenomena.
 - *Phantom read* - finding data (in where clause) added by another transaction
 - *Dirty read* - reading data not committed yet by another transaction
 - *Non-repeatable read* - rereading different data within the same transaction

Transaction Isolation Level (cont'd)

- To control isolation level of the Connection, use
 - `getTransactionIsolation()`
 - `setTransactionIsolation(int level)`

- May use also

`DatabaseMetaData.
supportsTransactionIsolationLevel(int)`

Transaction Isolation Level (cont'd)

- Levels are defined as integer constants in the **Connection** interface

TRANSACTION_NONE

- Transactions are not supported.

TRANSACTION_READ_UNCOMMITTED

- Dirty reads, non-repeatable reads and phantom reads can occur.

TRANSACTION_READ_COMMITTED

- Dirty reads are prevented; non-repeatable and phantom reads can occur.

TRANSACTION_REPEATABLE_READ

- Dirty and non-repeatable reads are prevented; phantom reads can occur.

TRANSACTION_SERIALIZABLE

- Dirty reads, non-repeatable reads and phantom reads are prevented.

JPA: Java Persistence API

`javax.persistence`

JPA Home Page:

`https://docs.oracle.com/javaee/7/tutorial/doc/partpersist.htm#BNBPY`

What is JPA?

- Persists plain Java objects, no need to write SQL
- Object/relational (O/R) mapping, relations between objects are managed by JPA.
 - Possible to store and load entire object graphs with one command.
- Uses post-compilation (when needed)

The first example (1/2)

- The **@Entity** and **@Id** annotations are all that is needed to turn a plain Java object into an entity managed by JPA.

```
package account;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Account {
    @Id
    private int acctNo;
    private String firstName;
    private String lastName;
    private int balance;
```

The first example (2/2)

```
public Account() {}

public Account(int acctNo, String firstName,
               String lastName, int balance) {
    this.acctNo = acctNo;
    this.firstName = firstName;
    this.lastName = lastName;
    this.balance = balance;
}

public int getAcctNo() {
    return acctNo;
}

// More business methods.
```

Main JPA Concepts

- Entity
 - A persistent abstraction.
 - Represented as Java class in the program and (typically but not necessarily) as table in the database.
 - An *entity instance* is a Java object in the program and a row in the database table(s).
 - Either fields or properties (JavaBeans style) are persisted. If fields or properties are persisted is decided by the location of annotations (close to fields or close to properties).
 - Must have no-argument **public** or **protected** constructor.
 - Fields may not be **public** and may not be accessed by other objects than the entity instance itself.
 - Must have the **@Entity** annotation.
 - Object/Relational (O/R) mapping with annotations to map objects to underlying relational data store.

Main JPA Concepts (cont)

- Primary key
 - Identifies an entity instance, must be unique for each instance.
 - A simple (non-composite) primary key must correspond to a single persistent field or property of the entity class.
 - The **@Id** annotation is used to denote a simple primary key.
- Context
 - A set of managed entity instances that exist in a particular data store.
 - The scope under which entity instances exist.
- Entity manager
 - An interface that defines the methods used to interact with the context, for example **create**, **remove** and **find**.
 - Each **EntityManager** instance is associated with a single context.

Main JPA Concepts (cont)

- Persistence unit
 - Defines the entities that are managed by an entity manager.
 - Defines where to store the entities persistently.
- Relation
 - A relation between entity instances that is persisted together with the entity instances.
- Query
 - The data store can be searched for entity instances using the **find** method in **EntityManager** or using the JPA Query Language (JPQL).
- Transaction
 - JPA is transaction aware. Transaction can be either container-managed or application-managed.

How to Start JPA

Applications that are not container-managed, for example Java SE applications, must use the classes `javax.persistence.Persistence` and `javax.persistence.EntityManagerFactory` to create an entity manager:

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

Entity Instance's Life cycle

- The life cycle of an entity instance is managed by the **EntityManager**.
- Entity instances are in one of four states: *new*, *managed*, *detached*, or *removed*.

Entity Instance's Life cycle (cont)

- *New* entity instances have no persistent identity and are not yet associated with a persistence context.
- *Managed* entity instances have a persistent identity and are associated with a persistence context.

Entity Instance's Life cycle (cont)

- ***Detached*** entity instances have a persistent identity and are not currently associated with a persistence context.
- ***Removed*** entity instances have a persistent identity, are associated with a persistent context, and are scheduled for removal from the data store.

How to Create a New Entity

```
@PersistenceContext
EntityManager em;
...
public LineItem createLineItem(Order order, Product product,
                                int quantity) {
    LineItem li = new LineItem(order, product,
                                quantity); // new
    order.getLineItems().add(li);
    em.persist(li); // managed
}
```

The entity (**li**) is *new* after this statement.

The entity is *managed* after this statement.

How to Find and Remove an Existing Entity

```
public void removeOrder(Integer orderId) {  
    try {  
        Order order = em.find(Order.class,  
                                orderId);  
        em.remove(order);  
    }  
}
```

- Entities are looked up with the **EntityManager** method **find** (more on queries below).
- Entities are removed with the **EntityManager** method **remove**.

Container-Managed Transactions

- The preferred way.
- Can only be used when JPA entities stays in a transaction aware container (e.g EJB or Spring)
- Transactions propagate from the calling container and are not handled by JPA code.
- Use declarative transaction demarcation in the container.

Application-Managed Transactions

- The only choice when there is no transaction aware container. This is the case with plain Java SE applications.
- Transaction must be started and stopped programmatically through the **EntityManager** interface.
- Easy to make mistakes!

Application-Managed Transaction

Example

```
EntityManager em = emFactory.createEntityManager();  
EntityTransaction transaction = em.getTransaction();  
transaction.begin();  
  
// Update entities here.  
  
em.getTransaction().commit();
```

Synchronization With Database

- The state of persistent entities is synchronized to the database when the transaction with which the entity is associated commits.
- To force synchronization of the managed entity to the database before transaction commit, invoke the **flush** method of the **EntityManager**.

Relationships

- Relationships are persisted by JPA and recreated when an entity instance is read from the database.
- Can be unidirectional or bidirectional.
- Can be one-to-one, one-to-many, many-to-one or many-to-many
- Entity updates (adding/removing entities or changing entity state) can cascade along relations when synchronizing with the database.

Relationship Example

```
@Entity
public class Employee {
    private Cubicle assignedCubicle;

    @OneToOne
    public Cubicle
    getAssignedCubicle() {
        return assignedCubicle;
    }

    public void setAssignedCubicle(
        Cubicle cubicle) {
        assignedCubicle = cubicle;
    }
    ...
}
```

```
@Entity
public class Cubicle {
    private Employee residentEmployee;

    @OneToOne(mappedBy="assignedCubicle")
    public Employee getResidentEmployee()
    {
        return residentEmployee;
    }

    public void setResidentEmployee(
        Employee employee) {
        residentEmployee = employee;
    }
    ...
}
```

Relationship Direction

- Unidirectional relationships can only be navigated in one direction.
 - Have relationship annotation only on one side.
- Bidirectional relationships can be navigated in both directions.
 - Have relationship annotations on both sides.
 - Inverse (not owning) side specifies that it is mapped by the property or field on the owning side:
@OneToOne(mappedBy="assignedCubicle")

Persisting Relationships

- The relationship is persisted based on the owning side.
- The owning side has the foreign key.

Relationship Multiplicities

- The following annotations exist:
 - **OneToOne**
 - **OneToMany**
 - **ManyToOne**
 - **ManyToMany**
- For **OneToOne** and **ManyToMany** relationships, any side may be the owning side.
- For **OneToMany** and **ManyToOne** relationships, the many side must be the owning side.

OneToMany/ManyToOne Example

(1/2)

```
@Entity
public class Employee {
    private Department department;

    @ManyToOne
    public Department getDepartment() {
        return department;
    }

    public void setDepartment(Department department)
    {
        this.department = department;
    }
    ...
}
```

OneToMany/ManyToOne Example

(2/2)

```
@Entity
public class Department {
    private Collection<Employee> employees = new HashSet();

    @OneToMany(mappedBy="department")
    public Collection<Employee> getEmployees() {
        return employees;
    }

    public void setEmployees(Collection<Employee> employees) {
        this.employees = employees;
    }
    ...
}
```

Cascading Updates

- Updates to the database may cascade along relationships.
 - Specified by the **cascade** element of the relationships annotations. The following cascade types can be specified:
 - **ALL**, Cascade all operations
 - **MERGE**, Cascade merge operation
 - **PERSIST**, Cascade persist operation
 - **REFRESH**, Cascade refresh operation
 - **REMOVE**, Cascade remove operation

Cascading Updates Example

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

Queries

- Query methods are in the **EntityManager**.
- The **find** method can be used to find instances by primary key:

```
em.find(Order.class, orderId);
```

Java Persistence Query Language, JPQL

- JPQL is a language with many similarities to SQL.
- JPQL is used to create, search, update or delete JPA entities.
- Has object-like syntax, the query below declares the variable **c**, which has the type **Customer** (must be an entity). Then searches for all instances of **Customer** that has the property **name** equal to the parameter **custName**. The **custName** parameter must be assigned a value before the query is executed.

```
SELECT c FROM Customer c  
WHERE c.name LIKE :custName
```

JPQL Example 1

- The `createQuery` method is used to create dynamic queries, queries that are defined directly within an application's business logic.

```
public EntityManager em;

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

JPQL Example 2

- The `createNamedQuery` method is used to create static queries, queries that are defined in meta data using the `NamedQuery` annotation.

```
@NamedQuery(  
    name="findCustomersByName",  
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName"  
)  
  
public EntityManager em;  
  
public List findWithName(String name) {  
    Query query = em.createNamedQuery("findCustomersByName");  
    query.setParameter("custName", name);  
    return query.getResultList();  
}
```


Criteria API

- The criteria API provides a way to generate queries in an object-oriented way with ordinary method calls, as opposed to the string manipulation used by JPQL.
- The advantage over JPQL is that it is type safe and that it is not required to know field names at compile time.
- The disadvantage is that notably more code is required to generate queries and that it is harder to read the queries.
- The Criteria API is not part of this course.