



# Transaction Management and Automatic Versioning

Originals of Slides and Source Code for Examples:  
<http://courses.coreservlets.com/Course-Materials/hibernate.html>

**Customized Java EE Training:** <http://courses.coreservlets.com/>  
Servlets, JSP, Struts, JSF/MyFaces/Facelets, Ajax, GWT, Spring, Hibernate/JPA, Java 5 & 6.  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.



**For live Spring & Hibernate training, see courses at <http://courses.coreservlets.com/>.**



**Taught by the experts that brought you this tutorial. Available at public venues, or customized versions can be held on-site at your organization.**

- Courses developed and taught by Marty Hall
  - Java 5, Java 6, intermediate/beginning servlets/JSP, advanced servlets/JSP, Struts, JSF, Ajax, GWT, custom mix of topics
- Courses developed and taught by coreservlets.com experts (edited by Marty)
  - Spring, Hibernate/JPA, EJB3, Ruby/Rails

Contact [hall@coreservlets.com](mailto:hall@coreservlets.com) for details

## Topics in This Section

- **Brush up on transactions, and recognize they're not just for databases**
- **Explore the different ways to include transactions into our applications**
- **Learn a technique for ensuring persistent changes to objects don't inadvertently overwrite other user's actions**

5

© 2009 coreservlets.com



## Transaction Management

Originals of Slides and Source Code for Examples:  
<http://courses.coreservlets.com/Course-Materials/hibernate.html>

**Customized Java EE Training:** <http://courses.coreservlets.com/>  
Servlets, JSP, Struts, JSF/MyFaces/Facelets, Ajax, GWT, Spring, Hibernate/JPA, Java 5 & 6.  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

## Transactions

- **Represents a single unit-of-work**
- **All or nothing – either all of the actions get committed or the entire effort fails**
- **Example:**
  - Transfer Money Between Accounts
  - Step 1: Withdraw \$500 from Savings Account
  - Step 2: Deposit \$500 into Checking Account
- **What happens if the system crashes after Step 1, but before Step 2?**

## Transaction Domain

- **Transactions are only a database concern**
  - Common misconception
- **Application business logic needs to be aware and set transactional boundaries**
  - Tell the system when to start and end a transaction
- **Other resources can also participate in transactions**
  - Java Messaging Service (JMS)
    - Roll back a message if something fails
  - Legacy Systems
  - Anything that leverages JTS (Java Transaction Service)
    - TransactionManager Interface

# Database Transactions: ACID

- **Atomicity**
  - One atomic unit of work
  - If one step fails, it all fails
- **Consistency**
  - Works on a consistent set of data that is hidden from other concurrently running transactions
  - Data left in a clean and consistent state after completion
- **Isolation**
  - Allows multiple users to work concurrently with the same data without compromising its integrity and correctness.
  - A particular transaction should not be visible to other concurrently running transactions.
- **Durability**
  - Once completed, all changes become persistent
  - Persistent changes not lost even if the system subsequently fails

# Database Transactions: Java

- **Programmatic**
  - Handle starting, as well as committing or rolling back transactions manually in your code
- **Declarative**
  - Write your implementation within some external container that abstracts out the transaction management from your code
- **Two-Phase Commits**
  - The handling of safely committing a transaction across two or more resources
    - Distributed transactions
      - Two different databases
      - Database and JMS server

## Transaction Demarcation

- **Setting the bounds of a transaction**
- **Key elements:**
  - Start point
  - End point
  - Proactive instructions to commit or rollback changes made

## Transaction Demarcation

- **What happens if you don't call commit or rollback at the end of your transaction?**
  - Start transaction
  - Execute SQL statements
  - Close transaction
    - NO COMMIT OR ROLLBACK ISSUED
- **What happens to the uncommitted transaction?**

# Transaction Demarcation

- **It depends! The JDBC specification doesn't say anything about pending transactions when close() is called on a connection.**
- **What happens depends on how the vendors implement the specification.**
  - With Oracle JDBC drivers, for example, the call to close() commits the transaction!
  - Most other JDBC vendors take the sane route and roll back any pending transaction when the JDBC connection object is closed and the resource is returned to the pool.
- **Why is this important to us?**
  - **BE SURE TO FULLY IMPLEMENT TRANSACTION DEMARCATION!!**
    - Don't let the vendor make the decision for you
    - May result in unexpected behavior

\*Source: *Java Persistence with Hibernate*

# Java Database Connectivity

- **Connection**
  - setAutoCommit()
  - preparedStatement()
  - createStatement()
  - close()
  - commit()
- **Statement**
  - executeUpdate()
  - executeQuery()
  - close()
- **ResultSet**
  - next()
  - get<Type>() [i.e. getString(), getInt() etc...]
  - close()

# JDBC without Transaction Management

```
public void transferMoney() {  
    ...  
    Connection conn = DriverManager.getConnection(DB_URL);  
  
    PreparedStatement debitAccountStatement =  
        connection.prepareStatement(  
            "UPDATE SAVINGS_ACCOUNT SET  
            BALANCE=? WHERE ACCOUNT_ID=?");  
  
    debitAccountStatement.executeUpdate();  
  
    PreparedStatement creditAccountStatement =  
        connection.prepareStatement(  
            "UPDATE CHECKING_ACCOUNT  
            SET BALANCE=? WHERE ACCOUNT_ID=?");  
  
    creditAccountStatement.executeUpdate();  
  
    ...  
}
```

# Programmatic Transactions in JDBC

1. Obtain a connection
2. Set auto commit on the connection to false
  - `connection.setAutoCommit(false)`
3. Execute series of statements, using the same connection
4. Commit the transaction if all statements successful, rollback otherwise
  - `connection.commit()`
  - `connection.rollback()`

# Programmatic Transactions in JDBC

```
public void transferMoney() {  
    ...  
    Connection conn = DriverManager.getConnection(DB_URL);  
    conn.setAutoCommit(false);  
    try {  
        PreparedStatement debitAccountStatement =  
            connection.prepareStatement(  
                "UPDATE SAVINGS_ACCOUNT SET  
                BALANCE=? WHERE ACCOUNT_ID=?");  
        ...  
        debitAccountStatement.executeUpdate();  
  
        PreparedStatement creditAccountStatement =  
            connection.prepareStatement(  
                "UPDATE CHECKING_ACCOUNT  
                SET BALANCE=? WHERE ACCOUNT_ID=?");  
        ...  
        creditAccountStatement.executeUpdate();  
  
        conn.commit();  
    }  
    ...  
}
```

# Programmatic Transactions in JDBC

```
...  
  
catch (Exception e) {  
    conn.rollback();  
    throw new BusinessException();  
}  
finally {  
    debitAccountStatement.close();  
    creditAccountStatement.close();  
    conn.close();  
}  
  
...
```

## Two Phase Commits - JavaSE

- **Very difficult to write a homegrown implementation for the Java Transaction Service (JTS) efficiently and effectively**
- **Not too many non-EE implementations available**
  - JBoss Transactions
  - Java Open Transaction Manager (JOTM)
- **Best to promote your application to an application server**
  - GlassFish
  - JBoss
  - WebLogic
  - WebSphere
  - Let them handle it for you

## Programmatic Transactions in Hibernate

- **org.hibernate.Transaction**
  - begin();
  - commit();
  - rollback();
  - setTimeout();
  - Obtained through session
    - session.beginTransaction();
    - session.getTransaction();
  - Works in a non-managed plain JDBC environment and also with application servers using JTA
- **javax.transaction.UserTransaction**
  - Java Transaction API (JTA) version
  - Hibernate recommends this as the primary choice whenever it's available

## Programmatic Transactions in Native Hibernate

- 1. Configure Hibernate for programmatic transactions (default behavior)**
  - Transaction factory already defaults to `hibernate.transaction.factory_class`
- 2. Get a handle to the current Session**
  - `HibernateUtil.getSessionFactory().getCurrentSession();`
- 3. Call `session.beginTransaction()`**
  - Session obtains a database connection and immediately sets `autoCommit(false)` for you and keeps it behind the scene
- 4. Modify persistent objects as needed**
  - Hibernate creates SQL statements for you based on modifications
- 5. Call `session.commit()`**
  - Hibernate flushes the persistence context using the collected up SQL statements and commits them to the database
- 6. Call `session.close()`**
  - Returns connection to pool

## Programmatic Transactions in Native Hibernate

- **Need to handle rollback if Hibernate encounters an error**
- **Hibernate throws unchecked exceptions**
  - Subclassed from `RuntimeException`
    - `HibernateException` – generic
    - `JDBCException` – caused by sql statement
      - Call `getSQL()` to see issue
    - Other subtypes, which are mapped from vendor-specific error codes
  - Set up try/catch around database calls and rollback if an exception occurs
    - `session.getTransaction().rollback()`
- **Close session in finally block**

# Programmatic Transactions in Native Hibernate

```
try {
    Session session =
        HibernateUtil.getSessionFactory().getCurrentSession();

    // session obtains connection and sets autoCommit(false)
    session.beginTransaction();

    // ...
    // modify and call persistent methods on various objects
    // ...

    // flush the Persistence Context and commit to the database
    session.getTransaction().commit();

catch (HibernateException he) {
    // roll back if an exception occurs
    session.getTransaction().rollback();
}
finally {
    // close session and return connection to pool
    session.close();
}
```

## Two Phase Commits - JavaSE

- **Still a problem, even using Hibernate**
- **Can continue to look into available non-EE JTS implementations**
  - JBoss Transactions
  - Java Open Transaction Manager (JOTM)
- **Best approach is STILL to use an application server and integrate with its JTS solution**

## Programmatic Transactions in Hibernate with JTA

- **Hibernate recommends leveraging JTA over native Hibernate APIs when available**
- **Within a JTA environment, actions are tied to the JTA system transaction**
- **Need to tell Hibernate about JTA, and the vendor specific TransactionManager**
  - Native Hibernate will still return a session, scoped and bound to the current JTA transaction
    - Though documentation says “with possible different transactional behavior”

## Possible Different Behavior

"The `getCurrentSession()` operation has one downside in a JTA environment. There is one caveat to the use of `after_statement` connection release mode, which is then used by default. Due to a silly limitation of the JTA spec, it is not possible for Hibernate to automatically clean up any unclosed `ScrollableResults` or `Iterator` instances returned by `scroll()` or `iterate()`. You must release the underlying database cursor by calling `ScrollableResults.close()` or `Hibernate.close(Iterator)` explicitly from a finally block"

- **Hibernate is unable to clean up `ScrollableResults` or `Iterators`**
  - **Objects used when querying the database**
  - **With JTA, you must consciously close these yourself**
- **This may be what they are referring to in the documentation**

\*Source: [http://hibernate.org/hib\\_docs/v3/reference/en/html/transactions-demarcation.html](http://hibernate.org/hib_docs/v3/reference/en/html/transactions-demarcation.html)

# Programmatic Transactions in Hibernate with JTA

- 1. Configure Hibernate for programmatic transactions with JTA**
  - `current_session_context_class = jta`
  - `transaction.factory_class = org.hibernate.transaction.JTATransactionFactory`
  - `transaction.manager_lookup_class = <!-- selected on a per-vendor basis -->`
- 2. Configure a data source to get connections from the application server**
  - Vendor specific
- 3. Get a handle to the current Session**
  - `HibernateUtil.getSessionFactory().getCurrentSession();`
- 4. Get a handle to the current system transaction**
  - `UserTransaction utx = (UserTransaction) new InitialContext().lookup("UserTransaction");`
- 5. Start the transaction**
  - `utx.begin();`
- 6. Obtain the current Hibernate Session as usual, and modify persistent objects as needed**
- 7. Commit the transaction**
  - `utx.commit();`
- 8. Call `session.close();`**

# Programmatic Transactions in Hibernate with JTA

```
<session-factory>
...
<property name="current_session_context_class">
  jta
</property>
<propertyname="transaction.factory_class">
  org.hibernate.transaction.JTATransactionFactory
</property>
<property name="transaction.manager_lookup_class">
  <!-- selected on a per-vendor basis -->
  org.hibernate.transaction.JBossTransactionManagerLookup
</property>
...
</session-factory>
```

## JBoss Datasource – lecture6DS.xml

```
<datasources>
  <local-tx-datasource>
    <jndi-name>Lecture6DS</jndi-name>

    <!-- Oracle Configuration -->
    <connection url>
      jdbc:oracle:thin:@localhost:1521:XE
    </connection-url>

    <driver-class>
      oracle.jdbc.driver.OracleDriver
    </driver-class>

    <user-name>lecture6</user-name>

    <password>lecture6</password>

    <min-pool-size>5</min-pool-size>
    <max-pool-size>100</max-pool-size>
  </local-tx-datasource>
</datasources>
```

## Programmatic Transactions in Hibernate with JTA

```
// inject the transaction
@Resource
private UserTransaction utx;

try {
  // start transaction
  utx.begin();

  Session session =
    HibernateUtil.getSessionFactory().getCurrentSession();

  // modify and call persistent methods on various objects
  // ...

  // flush the Persistence Context and commit to the database
  utx.commit();

  catch (HibernateException he) {
    // roll back if an exception occurs
    utx.rollback();
  }
}
```

## Two Phase Commits in Hibernate with JTA

- **Delegate the distributed transaction to the JTA container**
  - Likely an application server
- **In Hibernate:**
  1. Define a SessionFactory for each database
  2. Obtain a session for each database
  3. Call `openSession()` to use the current session on each one
    - Not “`getCurrentSession()`”! We need to manually handle flushing and committing the context ourselves
  4. Make your changes, persisting to each session as needed
  5. Flush both sessions
  6. Commit the transaction
  7. Close both sessions

## Two Phase Commits with JTA

```
@Resource
private UserTransaction utx; // inject the transaction

try {
    utx.begin();

    Session databaseSession1 =
        HibernateUtil.getSessionFactory1().openSession();

    Session databaseSession2 =
        HibernateUtil.getSessionFactory2().openSession();

    // modify and call persistent methods on various objects

    databaseSession1.flush();
    databaseSession2.flush();
    utx.commit(); // commit to both databases!

} catch (RuntimeException he) {
    utx.rollback(); // roll back both databases!
}
finally {
    session1.close(); // close both database sessions
    session2.close();
}
```

## Declarative Transactions

- **Pushes transaction management and responsibility to the app server**
  - Container-managed-transactions
- **Doesn't require extra boilerplate code**
  - Much cleaner and easier to read
  - Easier to follow the business logic
- **No need to manually start and stop transactions!**

## Declarative Transactions

- 1. Configure Hibernate for declaritive transactions**
  - `current_session_context_class = jta`
  - `transaction.factory_class = org.hibernate.transaction.CMTTransactionFactory`
  - `transaction.manager_lookup_class = <!-- selected on a per-vendor basis -->`
- 2. Setup a stateless session EJB to wrap your transactions, setting the transaction attribute on requiring methods**
  - `@TransactionAttribute(TransactionAttributeType.REQUIRED)`
- 3. Obtain the current Hibernate session within the transaction-scoped method**
- 4. Modify persistent objects as needed**
- 5. That's it!**

# EJB3.0 Session Bean

```
@Stateless
public class BankingServiceEJB {
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void transferFunds(Account a, Account b, double x) {

        // NOTICE - NO LOOKING AND STARTING UP TRANSACTION!!!

        Session session =
            HibernateUtil.getSessionFactory().openSession();

        // modify persistent objects
        a.debitAccount(x);
        b.creditAccount(x);

        // -- call saveOrUpdate() etc... on objects
        session.saveOrUpdate(a);
        session.saveOrUpdate(b);

        // NOTICE - NO COMMITTING TRANSACTION!!!
        // NOTICE - NO CLOSING HIBERNATE SESSION!!!
        // NOTICE - NO TRY/CATCH/FINALLY!!!
    }
}
```

© 2009 coreservlets.com



## Automatic Versioning

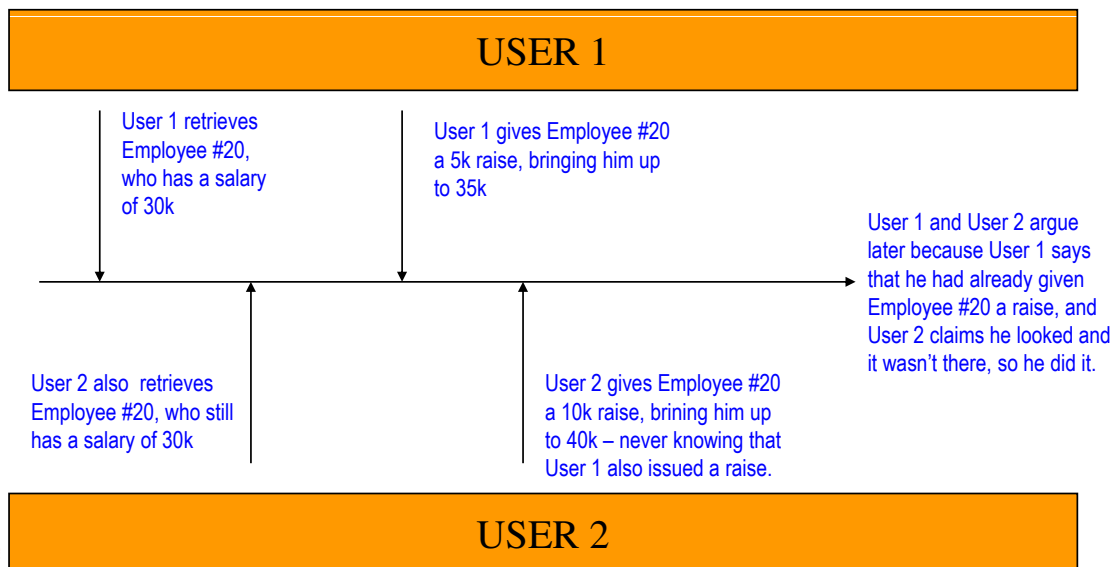
Originals of Slides and Source Code for Examples:  
<http://courses.coreservlets.com/Course-Materials/hibernate.html>

Customized Java EE Training: <http://courses.coreservlets.com/>  
Servlets, JSP, Struts, JSF/MyFaces/Facelets, Ajax, GWT, Spring, Hibernate/JPA, Java 5 & 6.  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

## Overwriting Other Users

- **System allows multiple users to work with and modify the same database record simultaneously**
- **Who ever updates last, wins!**
  - Previous save is unknowingly lost. Winner never even knew it was there.
- **Almost always overlooked by developers**
  - Difficult to reproduce or validate a customer complaint
  - No record of anything failing
  - Needs to be pro-actively thought of

## Overwriting Other Users



## Preventing Overwriting By Locking Records

- **Pessimistic**

- Don't allow other users to even read the record while you have it "checked out"
- We're pessimistic, believing someone will change it on us

- **Optimistic**

- Everyone can access it, and we check to see if it's been updated when we commit changes
- If it's been changed while a user was working on it, alert them so we can act accordingly
- We're optimistic, believing that nobody else will change it, but if it happens, at least we should know about it

## Hibernate's Optimistic Locking

- **Hibernate will automatically check and see if it's been modified while in your possession.**
  - If it's been changed, Hibernate will throw a `StaleObjectStateException`
- **Requires a column to keep track of versions**
  - Dedicated 'Version' Column (usually a number)
  - Can use a Timestamp
    - Caution! – two transactions could occur at the exact same time
- **When performing the update, version column is automatically included as part of the 'where' clause**
  - Update employee set salary=(salary\*1.10), version=version+1 where id=1 and version=1
  - Checks the value returned from the JDBC `executeUpdate()` call. If number of updated rows is zero, exception thrown

# Hibernate's Optimistic Locking

- 1. Decide on a versioning strategy**
  - Version column or date?
- 2. Add an attribute to every domain object Java class to represent the version**
  - `private int version;`
    - No setter/getter required, in fact, application should never programmatically update this column
    - At most, add a getter
- 3. Add a column to every domain object database table to represent the version**
- 4. Update the object mapping files to make Hibernate aware of the version attribute**
  - `<version name="version" access="field" column="version"/>`
  - MUST be placed immediately after the identifier property

# Account.hbm.xml Mapping File

```
<class name="courses.hibernate.vo.Account"
      table="ACCOUNT">

  <id name="accountId" column="ACCOUNT_ID">
    <generator class="native"/>
  </id>

  <version name="version" access="field"
          column="VERSION"/>

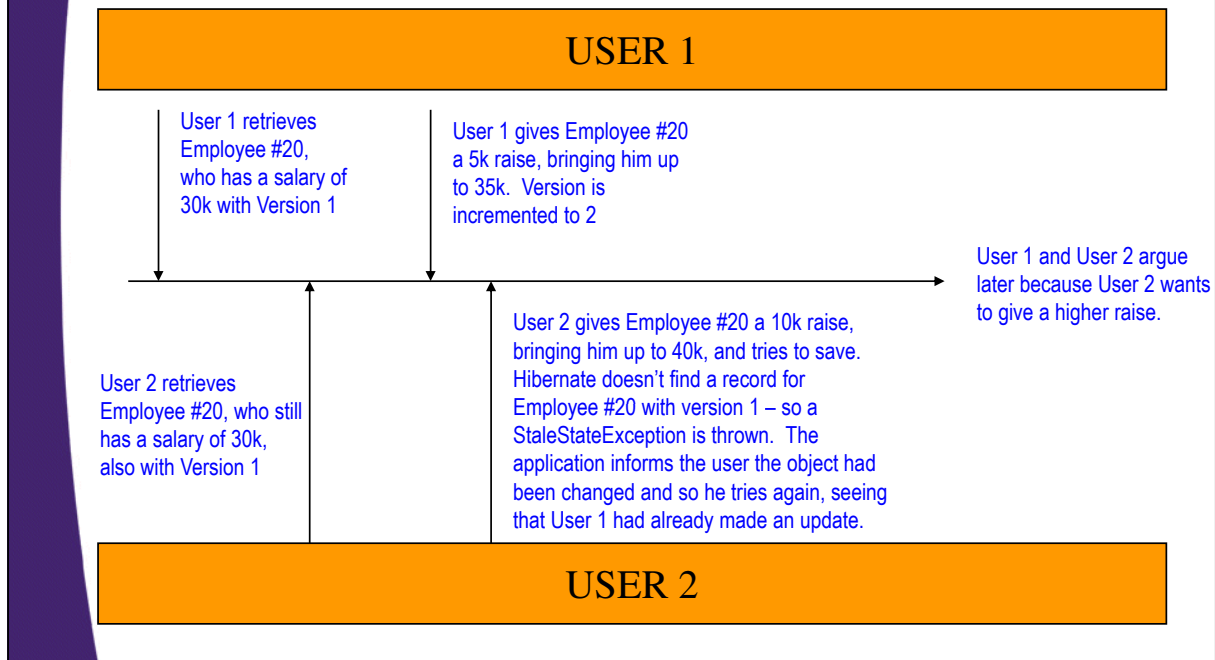
  <property name="creationDate" column="CREATION_DATE"
          type="timestamp"      update="false"/>

  <property name="accountType" column="ACCOUNT_TYPE"
          type="string"         update="false"/>

  <property name="balance" column="BALANCE"
          type="double"/>

</class>
```

# Overwriting Other Users



# Disabling Optimistic Locking

- **Hibernate allows you to disable automatic increment based on changes to a particular property or collection**
  - Used if you don't want to increment the version of an object based on the addition or removal of an associated object
- **In the object's mapping file, add the following to the property or collection mapping:**

```
<many-to-one name="ebills" ... optimistic-lock="false"/>
```

## Hibernate's Pessimistic Locking

- 1. No version or timestamp attributes/columns required**
- 2. Will NOT work for conversations**
  - Not valid for detached objects or sessions
  - Only within the same session
- 3. Does not scale as well**
  - Can become a bottleneck while other users wait
- 4. Once object obtained, call the lock() method**
  - LockMode.UPGRADE
    - Waits for the lock if unavailable
  - LockMode.UPGRADE\_NOWAIT
    - Throws an exception if lock unavailable
  - NOT AVAILABLE IN ALL DATABASE VENDORS
    - If not available, just returns the latest version

## Hibernate's Pessimistic Locking

```
Session session = getSessionFactory.openSession();
session.beginTransaction();

// retrieve an employee object
Employee emp =
    session.get(Employee.class, new Long(1));

// lock the row in the database.
// if already locked, wait for the lock to be released
// ONLY WORKS IF THE DB VENDOR SUPPORTS IT
session.lock(emp, LockMode.UPGRADE);

// now locked, manipulate object
// with no fear of being overwritten

...

session.commit();
session.close();
```



## Wrap-up

**Customized Java EE Training:** <http://courses.coreservlets.com/>  
Servlets, JSP, Struts, JSF/MyFaces/Facelets, Ajax, GWT, Spring, Hibernate/JPA, Java 5 & 6.  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

## Summary

- **In this lecture, we:**
  - Learned about transactions, and realized they are not just about databases – other resources can participate too!
  - Worked through different ways to manage transactions in our applications
    - Programmatic/Declarative
    - Native Hibernate/JTA
  - Unveiled a way to prevent lost database writes for users by using locking in our application, especially the recommended optimistic approach

## Preview of Next Sections

- **Queries, queries and more queries...**
  - Hibernate Query Language
  - Criteria Queries
  - Query by Example
  - Rolling your own SQL

49

© 2009 coreservlets.com



## Questions?

**Customized Java EE Training: <http://courses.coreservlets.com/>**  
Servlets, JSP, Struts, JSF/MyFaces/Facelets, Ajax, GWT, Spring, Hibernate/JPA, Java 5 & 6.  
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.