

Finding Objects

Via Hibernate Query
Language (HQL)



Introducing HQL

- In the last chapter we cheated slightly
 - Via the call `'session.get(Delivery.class,1)'`
 - We retrieved an object by specifying the primary key
 - Even though that key was dynamically generated
- In reality we will mostly find objects via queries
 - E.g. all deliveries of 'Advanced Java' in the next month
- Hibernate supports writing queries in SQL
 - But this means you must think in relational terms, understand the database schema and it harms portability
- Normally we write queries in Hibernate's own language
 - This is Hibernate Query Language (HQL)



Introducing HQL

- HQL is an object oriented version of SQL
 - You use a SQL like syntax to write the query
 - But define what you are after using OO concepts
- Because your query uses relationships between objects you don't need to know the database schema
 - Note that unlike SQL HQL is read-only
- If you were using the JPA you would not use HQL
 - Instead you would use Enterprise JavaBean Query Language
 - EJB-QL was designed for Entity EJB's but is reused in the JPA
 - Once you know one learning the other is straightforward



Running HQL Queries

- Queries are executed by 'Query' objects
 - These are built by sessions via the 'createQuery' method
- A query can be run in several ways
 - The 'list' method returns the results as a list
 - The 'iterate' method returns an iterator to the results
 - The 'uniqueResult' method returns a single reference
- The results of queries can be paginated
 - The 'setMaxResults' method controls the size of the results
 - The 'setFirstResult' method marks the point from which you wish to start returning results (the default is 0)



Basic HQL Queries

- The simplest query is ‘from Course’
 - This returns a Course object for each training course
 - Note that Hibernate works out what the SQL should be based on the mappings you have already specified
- This could be extended to ‘from Course as course’
 - The label ‘course’ is assigned to each Course object in turn
 - This has no effect but is essential in more complex queries
- A more real world example is given in the box below:
 - This returns a Delivery object for each delivery of the title
 - Again notice all the work you don’t have to do

from Delivery as delivery where delivery.course.title = ‘Intro To Java’



Running and Paging HQL Queries

```
public class PagingResults {
    public static void main(String[] args) {
        SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();
        Query query = session.createQuery("from Course");

        int startingPoint = 0;
        int batchSize = 3;
        query.setMaxResults(batchSize);
        List results;
        do {
            results = query.list();
            System.out.println("--- A Batch of Results ---");
            for(Object result : results) {
                System.out.println("\t" + result);
            }
            startingPoint += batchSize;
            query.setFirstResult(startingPoint);
        } while(results.size() == batchSize);
    }
}
```



Basic HQL Queries

- Queries don't have to return persistent objects
 - Instead they can return properties of those objects
 - Obviously this is good for performance
 - Multiple properties are returned as arrays of 'Object'
 - E.g. 'select delivery.course.title from Delivery delivery'
 - This returns the title of the all courses that have deliveries
- Inside the HQL where clause you can place:
 - Mathematical, logical and comparison operators
 - E.g. 'from Course as course where course.title like 'Intro%''
 - Comparison operators such as 'in' and 'between'
 - All the functions defined in EJB-QL




```
public class Demo {
    public static void main(String[] args) throws Exception {
        SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();
        runQuery(session, "from Course");           //all the courses in the Courses table
        runQuery(session, "from Delivery");         //all the deliveries in the Deliveries table
        //all the deliveries of "Intro to Java"
        runQuery(session, "from Delivery as delivery where delivery.course.title = 'Intro To Java'");
        //all the courses whose titles start with Intro
        runQuery(session, "from Course as course where course.title like 'Intro%'");
        //every course for which there are one or more deliveries
        runQuery(session, "select delivery.course from Delivery delivery");
        //the title of every course for which there are one or more deliveries
        runQuery(session, "select delivery.course.title from Delivery delivery");
        //as above but with results grouped by title
        runQuery(session, "select delivery.course.title from Delivery delivery group by delivery.course.title");
    }
    private static void runQuery(Session session, String queryStr) {
        Query query = session.createQuery(queryStr);
        List results = query.list();
        for(Object result : results) {
            System.out.println("\t" + result);
        }
    }
}
```



Parameters in HQL Queries

- Your queries should use parameters
 - Rather than relying on String concatenation
 - This provides better security and allows reuse
- Hibernate supports three types of parameters
 - Named parameters always start with a colon
 - Positional parameters are represented by question marks
 - Entity parameters are references to persistent objects
 - This is a particularly helpful feature
- Parameters are set via the setter methods of 'Query'
 - The 'setParameter' method can be used with any parameter



```
public static void showNamedParameters(Session session) {
    String queryStr = "from Delivery as delivery where delivery.course.title = :title";
    Query query = session.createQuery(queryStr);
    query.setString("title", "Intro to Java");
    List results = query.list();
    displayResults(results,"Named Parameters");
}


public static void showPositionalParameters(Session session) {
    String queryStr = "from Delivery as delivery where delivery.course.title = ?";
    Query query = session.createQuery(queryStr);
    query.setString(0, "Intro to Java");
    List results = query.list();
    displayResults(results,"Positional Parameters");
}

public static void showEntityParameters(Session session) {
    Course course = (Course)session.get(Course.class,"EF56");
    String queryStr = "from Delivery as delivery where delivery.course = :course";
    Query query = session.createQuery(queryStr);
    query.setEntity("course", course);
    List results = query.list();
    displayResults(results,"Entity Parameters");
}
```

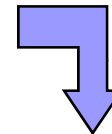


Storing Queries Inside Config Files

- Queries can be stored within OR mapping files
 - This makes your code cleaner and more maintainable
 - 'Session.getNamedQuery' finds your query
- They are placed within a 'query' element
 - Usually in a CDATA section to escape XML characters
 - You can place queries in the mapping file of the associated class, or alternatively separate all queries into a dedicated file
- Query names must be unique within the application
 - However if you nest the 'query' tag within a 'class' tag Hibernate prefixes the query name with the class name
 - E.g. 'demos.Delivery.allDeliveries' rather than 'allDeliveries'




```
<hibernate-mapping>
  <class name="demos.hibernate.courses.v3.Course" table="Courses">
    <!-- Mapping Omitted -->
  </class>
  <query name="allCourses">
    <![CDATA[ from Course ]]>
  </query>
  <query name="introductoryCourses">
    <![CDATA[ from Course as course where course.title like 'Intro%' ]]>
  </query>
</hibernate-mapping>
```

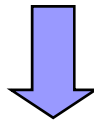


```
public static void main(String[] args) throws Exception {
    SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
    Session session = sessionFactory.openSession();
    runQuery(session.getNamedQuery("allCourses"));
    runQuery(session.getNamedQuery("introductoryCourses"));
}

private static void runQuery(Query query) {
    List results = query.list();
    System.out.println("--- Query \"" + query.getQueryString().trim() + "\" retrieved ---");
    for(Object result : results) {
        System.out.println("\t" + result);
    }
}
```



```
<hibernate-mapping>
  <class name="demos.hibernate.courses.v3.Delivery" table="Deliveries">
    <id name="id" column="DeliveryNum" type="int">
      <generator class="increment" />
    </id>
    <property name="startTime" column="StartDate" type="date" />
    <property name="endTime" column="EndDate" type="date" />
    <many-to-one name="course"
      class="demos.hibernate.courses.v3.Course"
      column="CourseNum" />
    <query name="allDeliveries">
      <![CDATA[ from Delivery ]]>
    </query>
  </class>
</hibernate-mapping>
```




```
runQuery(session.getNamedQuery(Delivery.class.getName() + ".allDeliveries"));
```



Native Queries

- Hibernate supports SQL based queries
 - You should only use this if absolutely necessary
 - Usually to access some feature not exposed in HQL
- Queries are created via 'Session.createQuery'
- An object of type 'SQLQuery' is returned
- Results can still be automatically mapped to objects
 - You supply the 'java.lang.Class' object of a mapped class
 - By calling 'addEntity' on the query object
 - Multiple entities can be specified
- SQL queries can be stored in a mapping file
 - In which case the entity mapping info is placed there as well




```
public class NativeQueries {
    public static void main(String[] args) throws Exception {
        SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();

        String nativeQuery = "SELECT * FROM Courses";
        SQLQuery query = session.createSQLQuery(nativeQuery);
        query.addEntity(Course.class);
        List results = query.list();
        System.out.println("--- Query \"\" + nativeQuery + "\" retrieved ---");
        for(Object result : results) {
            System.out.println("\t" + result);
        }
        System.out.println("\n");
    }
}
```



Building Queries Dynamically

- Sometimes queries cannot be hard-coded
 - E.g. you may wish to allow the user to create or customize the reports they are generating
- Traditional approaches are:
 - Create the query by concatenating within a 'StringBuffer'
 - Encapsulate the above code via the 'Builder Pattern'
- Hibernate offers a neater alternative
 - 'Criteria' objects return all instances of a class by default
 - You can attach 'Criterion' objects to restrict the data retrieved
 - The 'Restrictions' class acts as a factory for 'Criterion' objects



```
public static void main(String[] args) {
    SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
    Session session = sessionFactory.openSession();

    Criteria criteria1 = session.createCriteria(Course.class);
    Criterion criterion1 = Restrictions.eq("type", "Advanced");
    criteria1.add(criterion1);

    runCriteria(session, criteria1);

    Criteria criteria2 = session.createCriteria(Course.class);
    Criterion criterion2 = Restrictions.like("title", "Intro%");
    criteria2.add(criterion2);

    runCriteria(session, criteria2);
}

private static void runCriteria(Session session, Criteria criteria) {
    List results = criteria.list();
    System.out.println("--- Criteria Query Retrieved ---");
    for(Object result : results) {
        System.out.println("\t" + result);
    }
    System.out.println("\n");
}
```



Sessions In Depth

Features and Lifecycle



Session Objects in Depth

- So far we have only used Hibernate to:
 - Save persistent objects we have created
 - Load persistent objects via their primary key and HQL
 - Indirectly load and save objects via associations
- Now we want to look at sessions in detail
 - How they operate and how they should be used
- The methods of the Session class are straightforward
 - Once you are familiar with how Hibernate views your objects
 - Obviously underneath the hood there is a lot of detail
 - Most of which you do not need to be concerned about



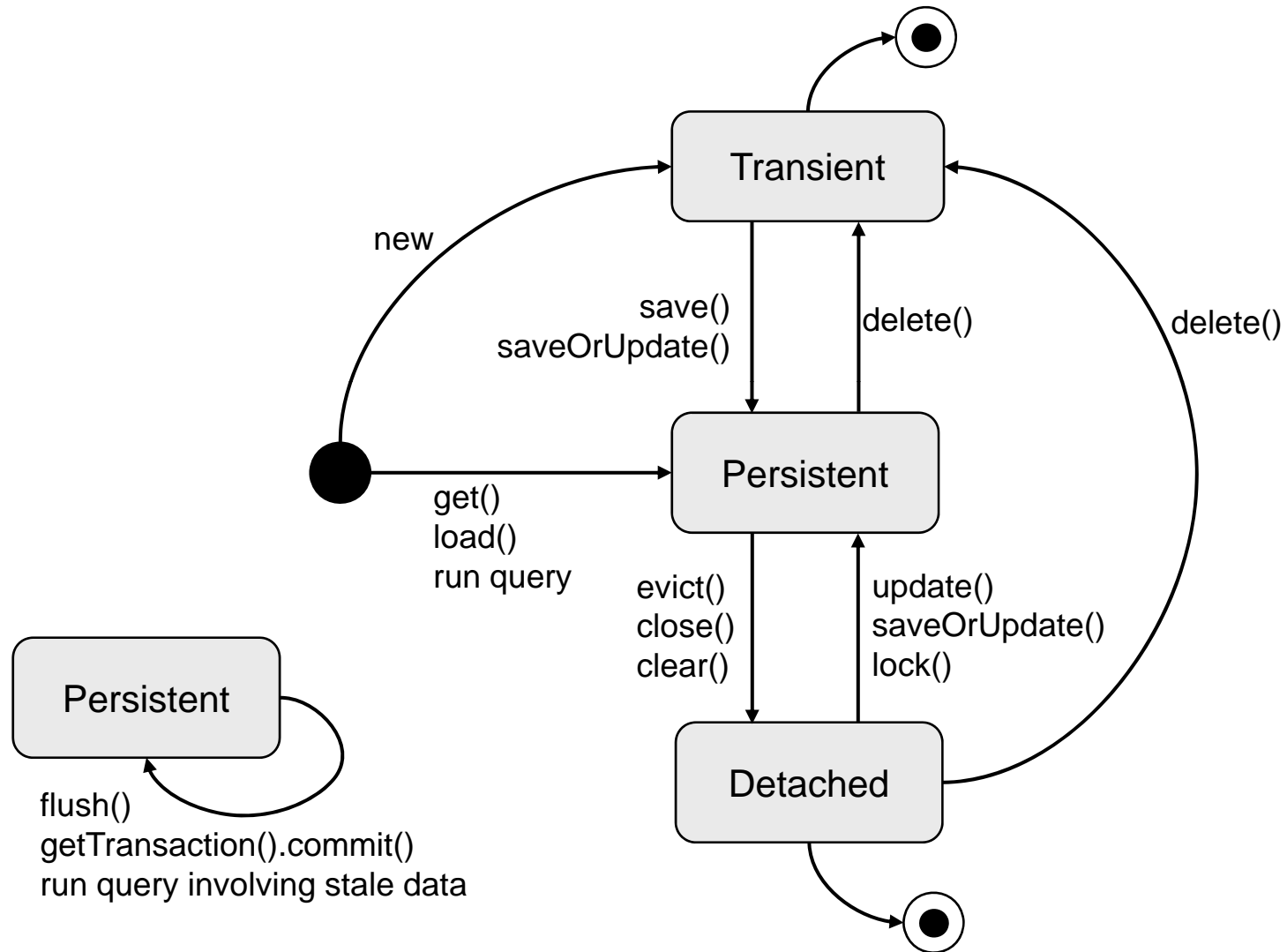
Important Methods of the Session

Method Name	Description
beginTransaction	Creates a new transaction or attaches to an existing one
load (5 versions)	Create a new persistent object
get (4 versions)	Create a new persistent object, returning null if no data exists
save (2 versions)	Save the data from a persistent object, creating a new DB record
update (2 versions)	Save the data from a persistent object into an existing record
saveOrUpdate (2 versions)	Combines the functionality of 'save' and 'update'
refresh (2 versions)	Update the state of a persistent object
delete (2 versions)	Delete the record in in the DB associated with a persistent object
evict	Removes a persistent object from Hibernate's control



The States of a Persistent Object

- Hibernate defines three states for an object
 - Transient, persistent and detached
- The lifecycle of an object is made up of these states
 - As summarized by the state-chart on the following slide
- Transient objects are POJO's created via new
 - Hibernate does not know about them yet
 - Transient objects are not associated with DB records
- Persistent objects are under Hibernate's control
 - Behind the scenes Hibernate has a reference to the object
 - The object has a database identity i.e. a primary key





The States of a Persistent Object

- Persistent objects can be created by:
 - Saving a transient object (which creates a record)
 - Building an object (via the Session or a query object)
- Note that Persistent objects can also appear indirectly
 - As a consequence of working with other persistent objects which have associations (i.e. the object tree)
- Persistent objects can become transient
 - When the 'Session.delete' method is called
 - This removes the associated record in the database
- Persistent objects can become detached
 - This is the third and final state in the lifecycle



The States of a Persistent Object

- Detached objects are no longer controlled by Hibernate
 - The reference Hibernate held to them has been removed
 - Hence their data is no longer being synchronized
- A persistent object can become detached in two ways:
 - When it is manually removed from the session via 'evict'
 - You might want to do this for performance reasons
 - When the session itself is closed or cleared
 - In which case all persistent objects become detached
- Detached objects can become persistent again
 - By re-synchronizing them the associated database record
 - This can be achieved by the 'update' or 'lock' methods
 - We shall discuss the difference between these later



Creating a new Persistent Object

```
@Test
public void creatingNewRecord() throws Exception {
    Session session = factory.openSession();
    session.beginTransaction();

    Course course = new Course();
    course.setNumber("XYZ987");
    course.setTitle("Basic Hibernate");
    course.setType("Intermediate");

    session.save(course);
    session.getTransaction().commit();
    session.close();

    assertEquals("Basic Hibernate", DatabaseUtilities.getCourseTitle(connection,"XYZ987"));
}
```



Finding a Record Based on its Key

```
@Test
public void findingRecordByKey() {
    Session session = factory.openSession();
    session.beginTransaction();

    Course course1 = (Course)session.get(Course.class, "AB12");
    Course course2 = (Course)session.get(Course.class, "IJ90");

    assertEquals("Intro To C++",course1.getTitle());
    assertEquals("XPath and XSLT",course2.getTitle());

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



Deleting Records Through Objects

```
@Test
public void deletingRecords() throws Exception {
    assertTrue(checkCourseExists(connection,"AB12"));
    assertTrue(checkCourseExists(connection,"IJ90"));

    Session session = factory.openSession();
    session.beginTransaction();

    Course course1 = (Course)session.get(Course.class, "AB12");
    Course course2 = (Course)session.get(Course.class, "IJ90");

    session.delete(course1); //Deleting a record via a persistent object
    session.evict(course2);
    session.delete(course2); //NB - Detached objects can be used in delete

    session.getTransaction().commit();
    session.close();

    assertFalse(checkCourseExists(connection,"AB12"));
    assertFalse(checkCourseExists(connection,"IJ90"));
}
```



Returning a Single Object via HQL

```
@Test
public void findingRecordViaHQL() {
    Session session = factory.openSession();
    session.beginTransaction();

    Query query =
        session.createQuery("from Course as course where course.title = 'Intro To C++'");
    Course result = (Course)query.uniqueResult();
    assertEquals("AB12",result.getNumber());
    assertEquals("Beginners",result.getType());
    assertEquals("Intro To C++",result.getTitle());

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



Synchronizing a Persistent Object

- Note no method call is required to tell Hibernate that the objects state has changed

```
@Test
public void persistModifiedObject() throws Exception {
    Session session = factory.openSession();
    session.beginTransaction();

    Course course = (Course)session.get(Course.class, "AB12");
    course.setTitle("Advanced C++");

    session.getTransaction().commit();
    session.close();

    assertEquals("Advanced C++", DatabaseUtilities.getCourseTitle(connection,"AB12"));
}
```



Synchronizing a Persistent Object

- In this case we are modifying an association of the persistent object we retrieved

```
@Test
public void persistModifiedObjectViaAssociation() throws Exception {
    Session session = factory.openSession();
    session.beginTransaction();

    Query query =
        session.createQuery("from Delivery as delivery where delivery.course.number = 'AB12'");
    Delivery delivery = (Delivery)query.list().get(0);
    delivery.getCourse().setTitle("Advanced C++");

    session.getTransaction().commit();
    session.close();
    assertEquals("Advanced C++", DatabaseUtilities.getCourseTitle(connection,"AB12"));
}
```



Sessions and Transactions

- All ORM frameworks depend on transactions
 - To control when information should be saved to the database
 - For a detailed discussion on transactions see the appendices
- A Hibernate session is not the same as a transaction
 - A transaction represents a set of actions that must be performed against one or more resources as an indivisible unit
 - A session represents a logical unit of work within the application
 - Such as that defined on a Workflow or UML Sequence diagram
- Sessions can span any number of transactions
 - This allows us to implement common Use Cases where the user works through several forms, entering data and making choices



```
@Test
public void sessionsCanSpanMultipleConnections() throws Exception {
    Session session = factory.openSession();
    session.beginTransaction();

    Course course1 = (Course)session.get(Course.class, "AB12");
    Course course2 = (Course)session.get(Course.class, "IJ90");
    assertEquals("Intro To C++",course1.getTitle());
    assertEquals("XPath and XSLT",course2.getTitle());

    course1.setTitle("Intermediate C++");

    session.getTransaction().commit();
    assertEquals("Intermediate C++", getCourseTitle(connection,"AB12"));
    session.beginTransaction();

    course1.setTitle("Advanced C++");
    course2.setTitle("Advanced XSLT");

    session.getTransaction().commit();
    session.close();

    assertEquals("Advanced C++", getCourseTitle(connection,"AB12"));
    assertEquals("Advanced XSLT", getCourseTitle(connection,"IJ90"));
}
```



Sessions and Transactions

- By default Hibernate uses JDBC transactions
 - The JDBC library has built in support for local transactions
- Inside a container it uses the Java Transactions API
 - The JTA was created to provide distributed transactions that could span many resources (for full details see the appendices)
 - Resources can include databases, MOM servers and connectors
- ‘beginTransaction’ might not create a new transaction
 - If Hibernate is running within a JEE Container and a JTA transaction is already underway then we join with it
- Similarly ‘commit’ might not finish the transaction
 - Again within a JEE container the transaction will continue on



When Data is Synchronized

- Hibernate doesn't immediately propagate changes in persistent objects to the database
 - So a call to 'save' or 'update' does not cause the generated SQL statements to be run at that time
 - Instead changes are made in batches when required
 - This feature is known as transparent write behind
- Synchronization normally occurs on 'commit'
 - So if an objects state had changed many times during the scope of the transaction only the final values are written
 - No method call is required when an objects state changes
 - Hibernate automatically detects changes to persistent objects
 - This feature is known as automatic dirty checking



When Data is Synchronized

- Synchronization occurs at two other points
 - Via an explicit call to the 'flush' method
 - Before an HQL or SQL query is executed
- Explicit calls to 'flush' are not recommended
 - But may be necessary if you are mixing Hibernate with legacy data access code which uses JDBC directly
- Hibernate detects if a query requires a flush
 - I.e. if unsaved changes made to persistent objects could affect the results of the query you are about to run then a flush occurs
 - You can disable this via 'setFlushMode(FlushMode.COMMIT)'



Object Equality Within Hibernate

- Hibernate does not duplicate persistent objects
 - Within a session you cannot have multiple persistent objects mapped to the same DB record
 - If you try to generate multiple objects based on the same primary key then the same instance is returned
 - This means equality of reference is all you need
- If you are reattaching detached objects within another session then a proper equals method is important
 - E.g. if you try to add objects created in different sessions into the same Set Hibernate must be able to identify duplicates



Object Equality Within Hibernate

```
@Test
public void proveSinglePersistentObjectCreated() {
    Session session = factory.openSession();
    session.beginTransaction();

    Query query = session.createQuery("from Course as course where course.title = 'Intro To C++'");
    Course result1 = (Course)query.uniqueResult();

    Course result2 = (Course)session.get(Course.class, "AB12");
    Course result3 = (Course)session.get(Course.class, "AB12");

    assertEquals(result1,result2);
    assertEquals(result2,result3);
    assertEquals(result1,result3);

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



Reattaching Detached Objects

- Use 'update' when you know (or suspect) that the object will have changed since being detached
 - This forces Hibernate to update the values in the database
 - If a persistent object already exists with the same primary key then an exception will be thrown
 - Hence calls to update should be placed after creating the session
- Only use 'lock' if you know the object has not changed
 - Any changes made before 'lock' are not written to the DB
- The 'merge' method copies values in a detached object into a persistent object with the same primary key
 - This was introduced in Hibernate V2 to cope with the case where you wanted to do an 'update' but couldn't guarantee uniqueness



```
@Test
public void updatingDetachedObjectsViaUpdate() throws Exception {
    Session session1 = factory.openSession();
    session1.beginTransaction();

    Course course = (Course)session1.get(Course.class, "AB12");
    assertEquals("Intro To C++", DatabaseUtilities.getCourseTitle(connection, "AB12"));

    session1.getTransaction().commit();
    session1.close();


    course.setTitle("Advanced C++");

    Session session2 = factory.openSession();
    session2.beginTransaction();

    session2.update(course);

    session2.getTransaction().commit();
    session2.close();

    assertEquals("Advanced C++", DatabaseUtilities.getCourseTitle(connection, "AB12"));
}
```



```
@Test
public void updatingDetachedObjectsViaMerge() throws Exception {
    Session session1 = factory.openSession();
    session1.beginTransaction();

    Course course1 = (Course)session1.get(Course.class, "AB12");
    assertEquals("Intro To C++", DatabaseUtilities.getCourseTitle(connection,"AB12"));

    session1.getTransaction().commit();
    session1.close();

    course1.setTitle("Advanced C++");

    Session session2 = factory.openSession();
    session2.beginTransaction();

    Course course2 = (Course)session2.get(Course.class, "AB12");
    session2.merge(course1);
    assertEquals("Advanced C++",course2.getTitle());

    session2.getTransaction().commit();
    session2.close();

    assertEquals("Advanced C++", DatabaseUtilities.getCourseTitle(connection,"AB12"));
}
```



```
@Test
public void updatingDetachedObjectsViaLock() throws Exception {
    Session session = factory.openSession();
    session.beginTransaction();

    Course course = (Course)session.get(Course.class, "AB12");
    assertEquals("Intro To C++", course.getTitle());

    session.evict(course);

    //without this call any changes made to the object will not be
    // written to the database as the object is detached
    session.lock(course, LockMode.NONE);

    course.setTitle("Advanced C++");

    session.getTransaction().commit();
    session.close();

    assertEquals("Advanced C++", DatabaseUtilities.getCourseTitle(connection, "AB12"));
}
```



```
@Test
public void changesMadeToDetachedObjectLostWithLock() throws Exception {
    Session session = factory.openSession();
    session.beginTransaction();

    Course course = (Course)session.get(Course.class, "AB12");
    assertEquals("Intro To C++", course.getTitle());

    session.evict(course);

    //this change is made while the object is detached, if we
    // reattach it with 'lock' Hibernate will assume no changes
    // have been made and the DB will not be updated
    course.setTitle("Advanced C++");

    session.lock(course, LockMode.NONE);

    session.getTransaction().commit();
    session.close();

    assertEquals("Intro To C++", getCourseTitle(connection, "AB12"));
}
```



Cascading Relationships

- We have seen that Hibernate manages trees of objects
 - All dirty objects are flushed, including ones that were never directly referenced (e.g. `delivery.getCourse.setTitle("XXX")`)
- However we have not discussed non-persistent objects
 - If 'A' references 'B' and neither are in the cache what happens to 'B' when we call 'save', 'update' or 'delete' on 'A'?
- By default associations of transient or detached objects are not affected by an 'update', 'save' or 'delete'
 - This is changed via the optional 'cascade' attribute
- The 'cascade' attribute has six possible values
 - Allowing you to select which operations should be chained



Values for Cascading Persistence

Cascade Value	Description
none	The default - associations are never followed
all	Associations are followed on all operations
save-update	Associations are followed when saving a transient object or when updating a detached object
delete	Associations are only followed when deleting detached objects
delete-orphan	Persistent objects will be deleted which have been removed from their association with the detached object
all-delete-orphan	The combination of 'all' and 'delete-orphan'