Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

**for** ( int i=0; i<100000; i++ ) {

Customer customer = **new** Customer(.....);

session.save(customer);

}

tx.commit();

session.close();

This fails with exception OutOfMemoryException after around 50000 rows on most systems. The reason is that Hibernate caches all the newly inserted Customer instances in the session-level cache. There are several ways to avoid this problem.

Before batch processing, enable JDBC batching. To enable JDBC batching, set the property hibernate.jdbc.batch\_size to an integer between 10 and 50.

When you make new objects persistent, employ methods flush() and clear() to the session regularly, to control the size of the first-level cache.

Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

**for** ( int i=0; i<100000; i++ ) {

Customer customer = **new** Customer(.....);

session.save(customer);

**if** ( i % 20 == 0 ) { //20, same as the JDBC batch size

//flush a batch of inserts and release memory:

session.flush();

session.clear();

}

}

tx.commit();

session.close();

When you retriev and update data, flush() and clear() the session regularly. In addition, use method scroll() to take advantage of server-side cursors for queries that return many rows of data.

**Example 4.3. Using scroll()**

Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")

.setCacheMode(CacheMode.IGNORE)

.scroll(ScrollMode.FORWARD\_ONLY);

int count=0;

**while** ( customers.next() ) {

Customer customer = (Customer) customers.get(0);

customer.updateStuff(...);

**if** ( ++count % 20 == 0 ) {

//flush a batch of updates and release memory:

session.flush();

session.clear();

}

}

tx.commit();

session.close();