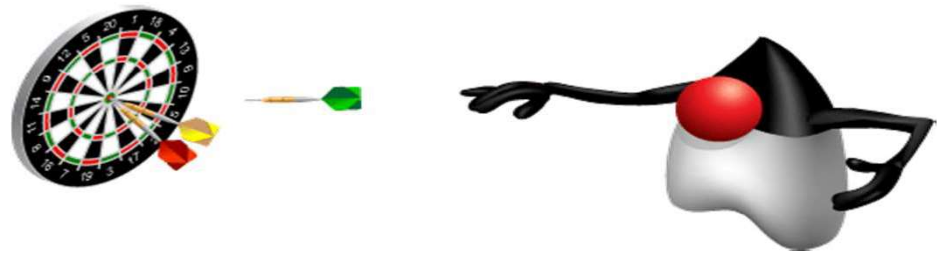
9

Hibernate Generators & Hibernate Transactions

Objectives

After completing this lesson, you should be able to do the following:

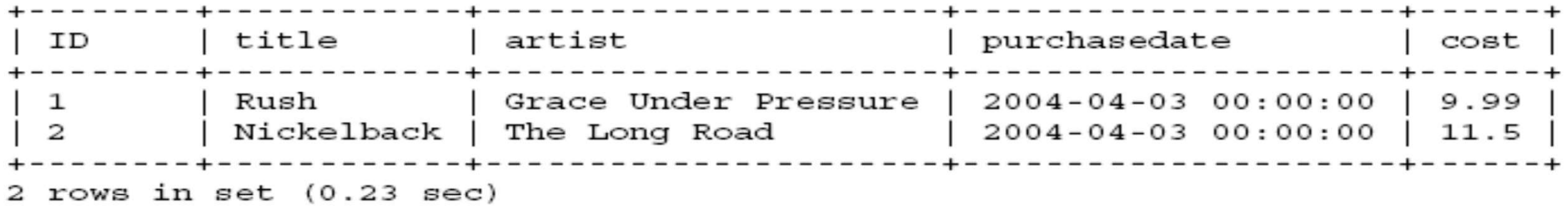
- Hibernate Generators
- Brush up on transactions, and recognize they're not just for databases
- Explore the different ways to include transactions into our applications
- Learn a technique for ensuring persistent changes to objects don't inadvertently overwrite other user's Actions



Generators

Increment

- The increment generator is probably the most familiar creator of IDs. Each time the generator needs to generate an ID, it performs a select on the current database, determines the current largest ID value, and increment to the next value.



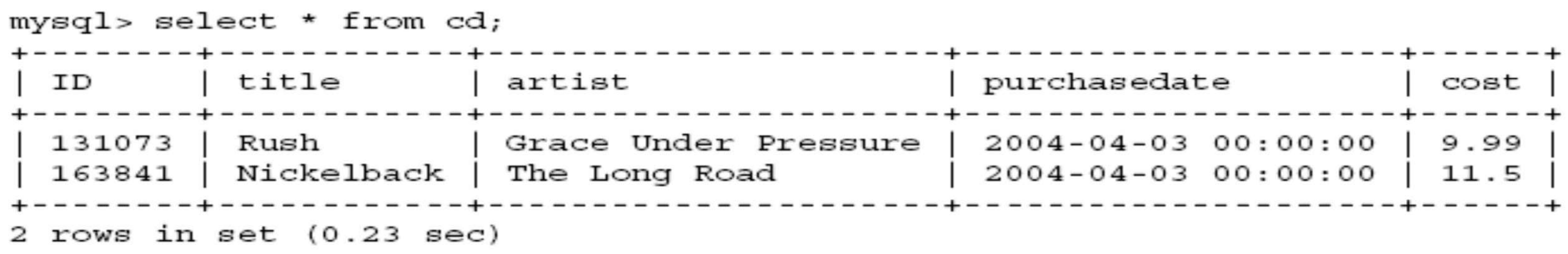
```
+---------+------------+------------------------+----------------------+------+
| ID      | title      | artist                 | purchasedate         | cost |
+---------+------------+------------------------+----------------------+------+
| 1       | Rush       | Grace Under Pressure   | 2004-04-03 00:00:00  | 9.99 |
| 2       | Nickelback | The Long Road          | 2004-04-03 00:00:00  | 11.5 |
+---------+------------+------------------------+----------------------+------+
2 rows in set (0.23 sec)
```

Identity

- If the database has an identity column associated with it, Hibernate can take advantage of the column to indicate when an object hasn’t been added to the database. Supported databases include DB2, MySQL, Microsoft SQL Server, Sybase, and HypersonicSQL.

- ***Sequence***
  - If the database has a sequence column associated with it, Hibernate can take advantage of the column to determine if the object has been added to the database. Supported databases include DB2, PostgreSQL, Oracle, SAP DB, McKoi, and InterBase.

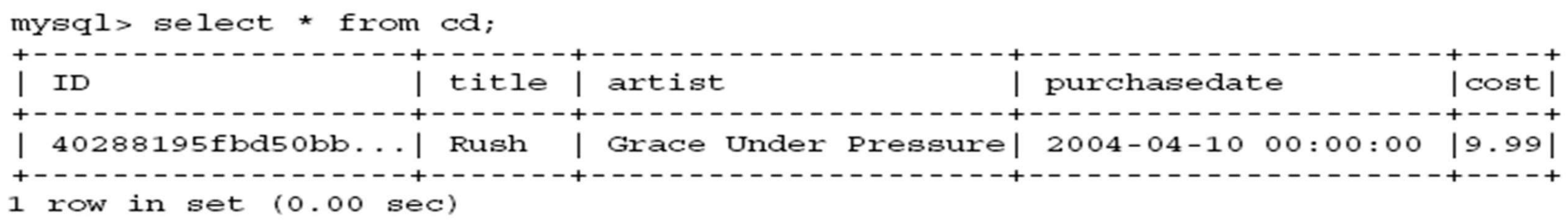
- ***Hilo***
  - The hilo generator generates unique IDs for a database table. The IDs won’t necessarily be sequential. This generator must have access to a secondary table that Hibernate uses to determine a seed value for the generator. The default table is hibernate-unique-key, and the required column is next-value.



```
mysql> select * from cd;
+---------+-------------+-----------------------+----------------------+------+
| ID      | title       | artist                | purchasedate         | cost |
+---------+-------------+-----------------------+----------------------+------+
| 131073  | Rush        | Grace Under Pressure  | 2004-04-03 00:00:00  | 9.99 |
| 163841  | Nickelback  | The Long Road         | 2004-04-03 00:00:00  | 11.5 |
+---------+-------------+-----------------------+----------------------+------+
2 rows in set (0.23 sec)
```

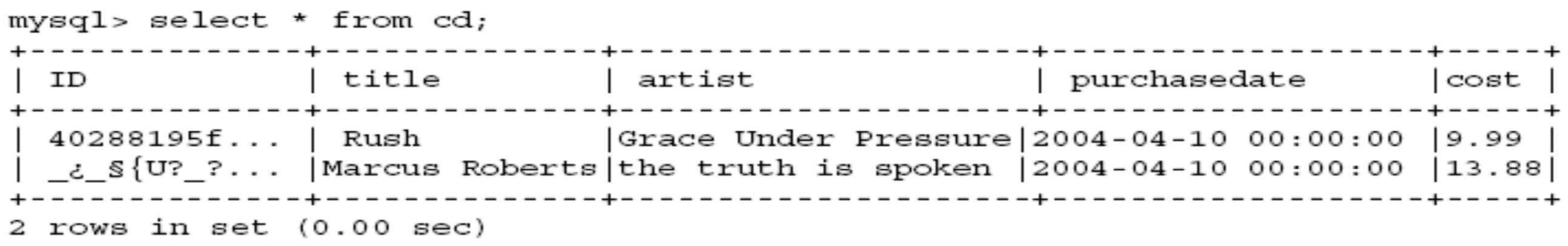
- ***seqhilo***
  - With the seqhilo generator, Hibernate combines the sequence and hilo generators. Supported databases include DB2, PostgreSQL, Oracle, SAP DB, McKoi, and InterBase. Be sure you use this generator with the correct database.

- ***Uuid.hex***
  - This generator creates a unique strings based on appending the following values: the machine's IP address, the startup time of the current JVM, the current time, and the counter value. It supports all databases



```
mysql> select * from cd;
+--------------------+-------+-----------------------+----------------------+----+
| ID                 | title | artist                | purchasedate         |cost|
+--------------------+-------+-----------------------+----------------------+----+
| 40288195fbd50bb...| Rush  | Grace Under Pressure| 2004-04-10 00:00:00 |9.99|
+--------------------+-------+-----------------------+----------------------+----+
1 row in set (0.00 sec)
```

- ***Uuid.string***
  - This generator is like Uuid.hex, but the result is a string 16 characters long. It supports all databases except PostgreSQL, and it has no parameters. Possible column types are string, varchar, and text.



```
mysql> select * from cd;
+--------------+--------------+---------------------+--------------------+------+
| ID           | title        | artist              | purchasedate       |cost  |
+--------------+--------------+---------------------+--------------------+------+
| 40288195f... | Rush         |Grace Under Pressure|2004-04-10 00:00:00 |9.99  |
| _¿_§{U?_?... |Marcus Roberts|the truth is spoken  |2004-04-10 00:00:00 |13.88|
+--------------+--------------+---------------------+--------------------+------+
2 rows in set (0.00 sec)
```

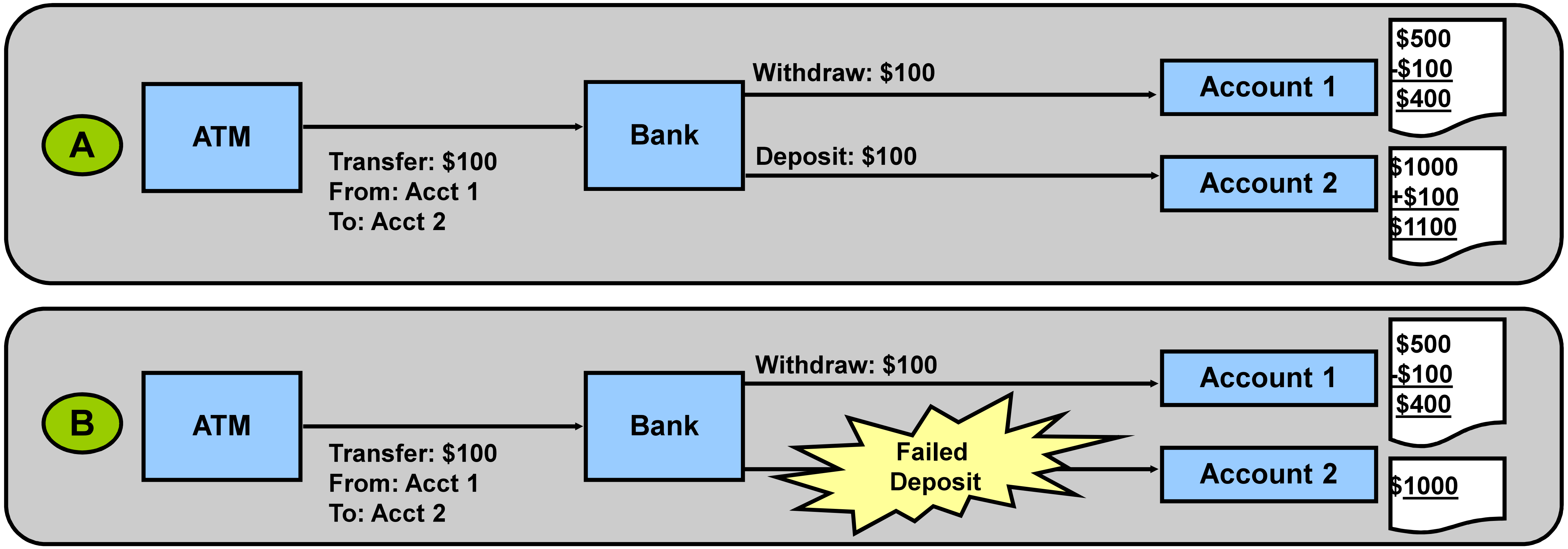
What Is a Transaction?

A transaction:

- Is a single, logical unit of work or a set of tasks that are executed together
- May access one or more shared resources (such as databases)
- Must be atomic, consistent, isolated, and durable (ACID)

Example of a Transaction

- Successful transfer (A)
- Unsuccessful transfer (accounts are left in an inconsistent state) (B)



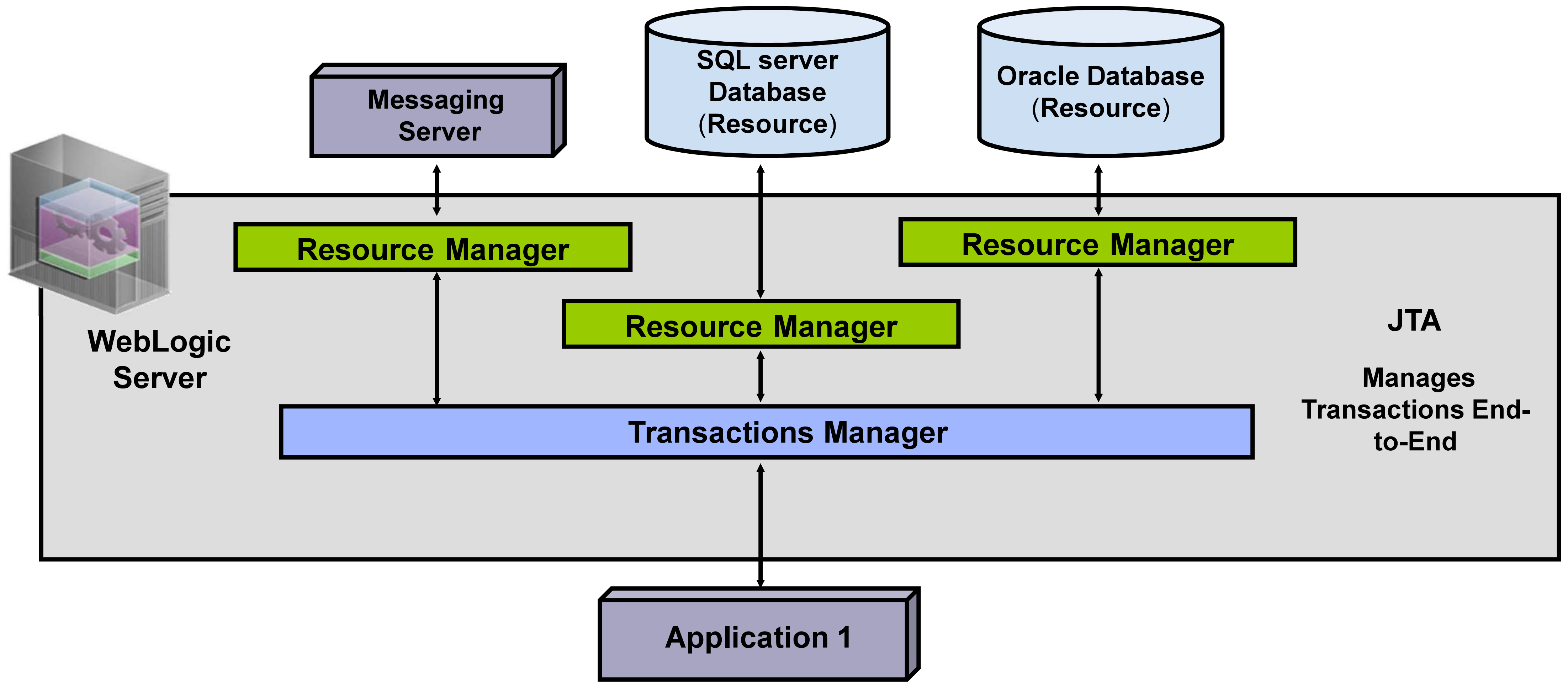
Transactions

- **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

Transaction Management



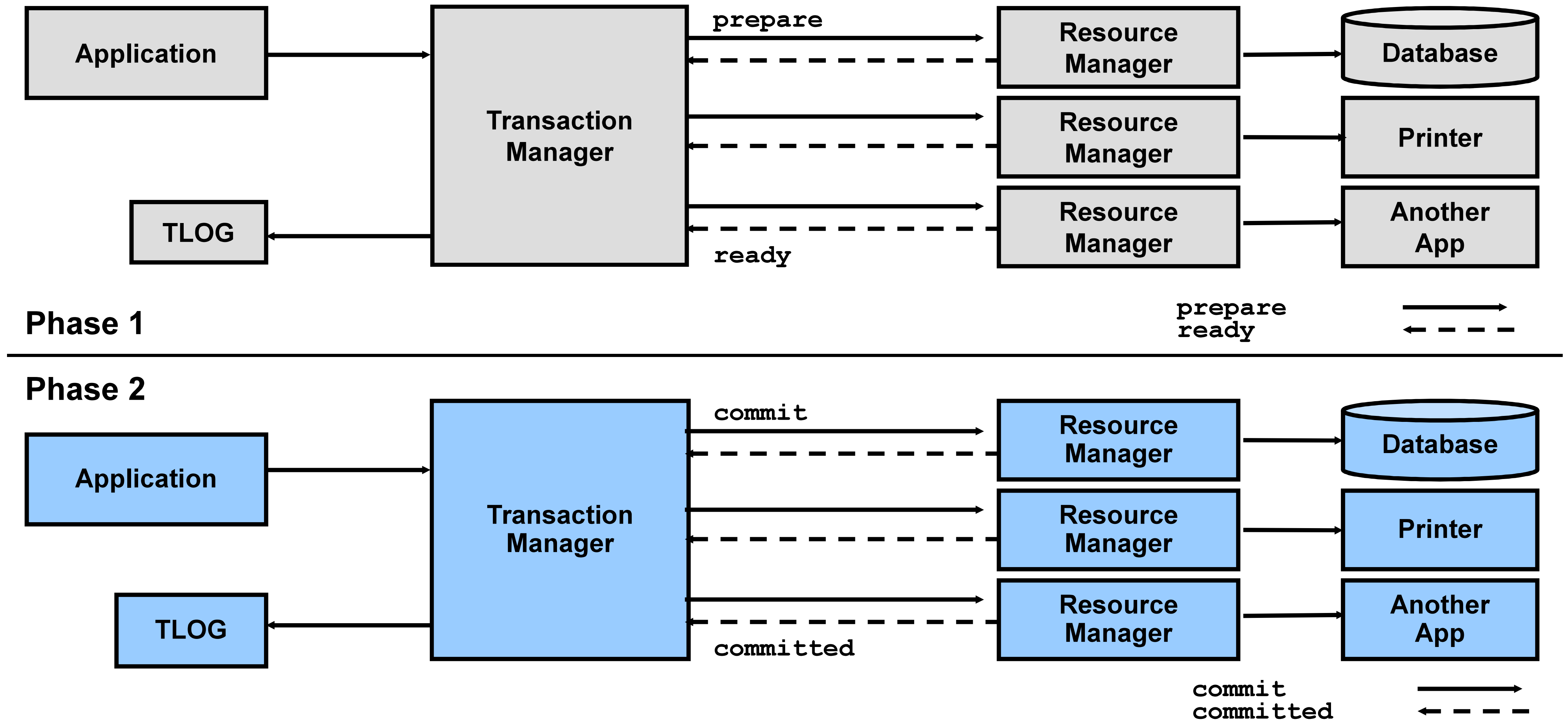
Database Transactions: ACID

- **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 concurrentlyrunning transactions.
- **Durability**
  - Once completed, all changes become persistent
  - Persistent changes not lost even if the system subsequently fails

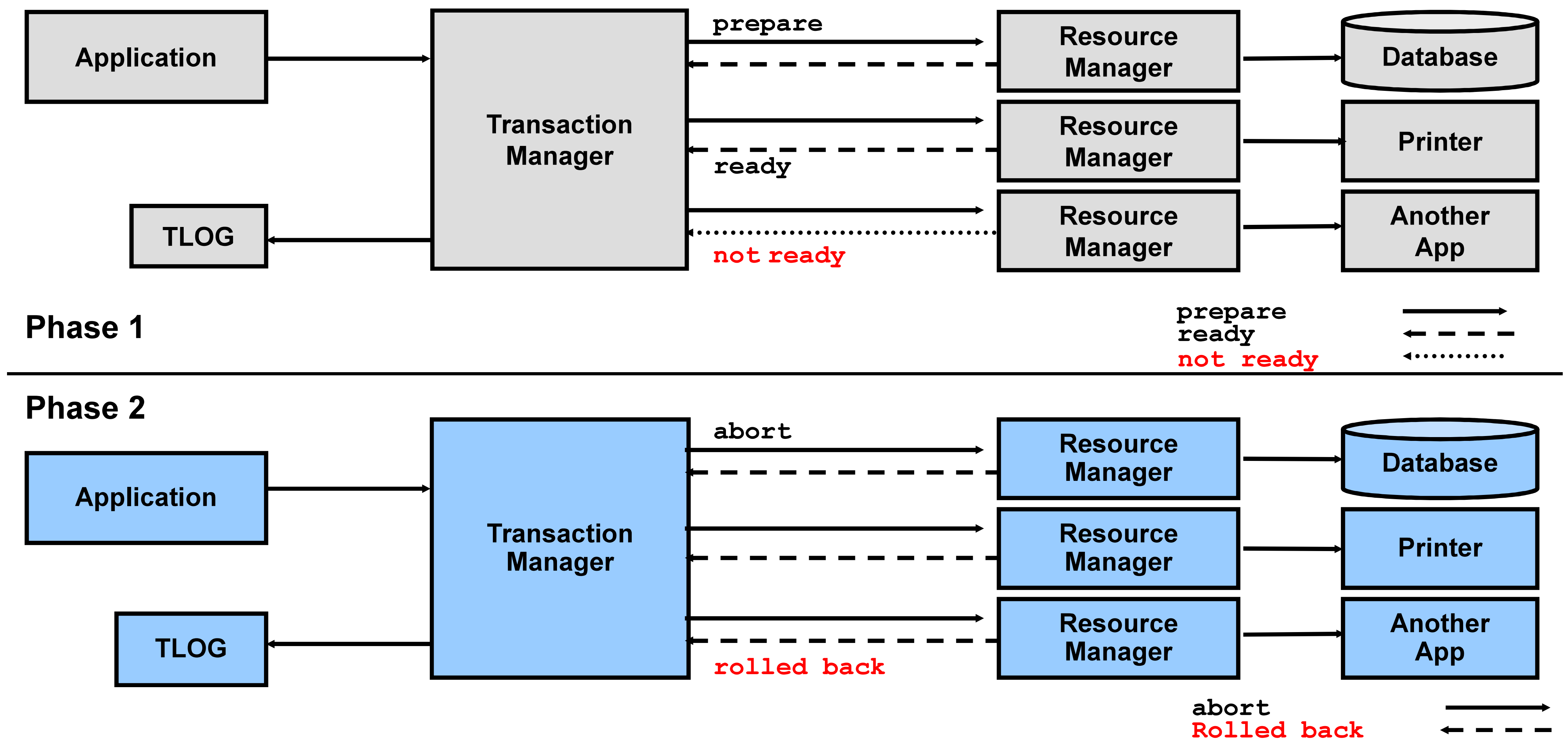
Two-Phase Commit Protocol

- The Two-Phase Commit (2PC) protocol uses two steps to commit changes within a distributed transaction.
  - Phase 1 asks RMs to prepare to make the changes
  - Phase 2 asks RMs to commit and make the changes permanent, or to roll back the entire transaction
- A global transaction ID (XID) is used to track all changes associated with a distributed transaction.

Successful Two-Phase Commit



Unsuccessful Two-Phase Commit



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
    - »

Java Transaction API (JTA)

- Hibernate uses JTA to implement and manage transactions.
- JTA provides the following support:
  - Creates a unique transaction identifier (XID)
  - Supports an optional transaction name
  - Tracks objects involved in transactions
  - Notifies databases of transactions
  - Orchestrates 2PC using XA
  - Executes rollbacks
  - Executes automatic recovery procedures when failure
  - Manages time-outs

Programmatic Transactions

- **org.hibernate.Transaction**
  - begin();
  - commit();
  - rollback();
  - 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

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

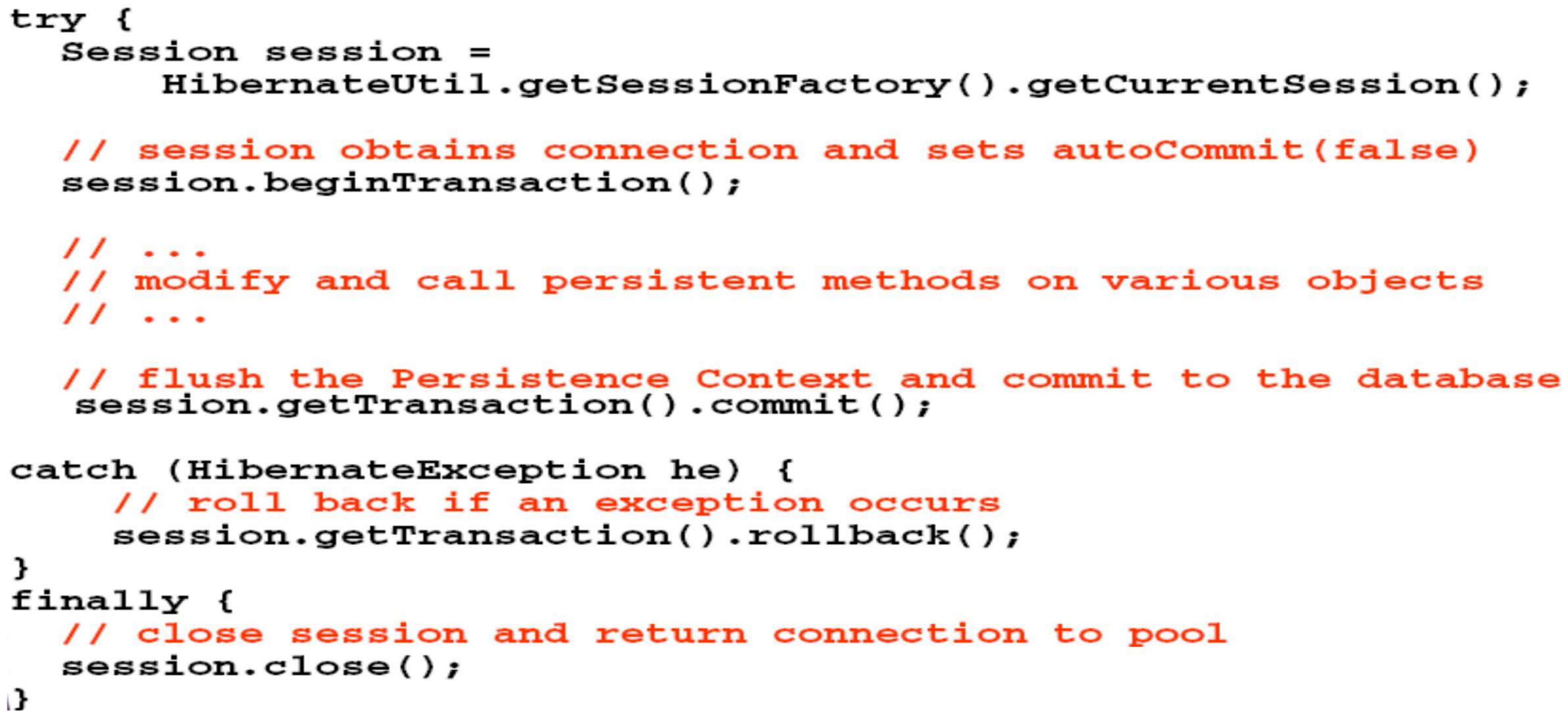
5. Call tranx.commit()

– 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

Sample



```
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();
}
```

Summary

In this lesson, you should have learned how to:

- Hibernate Generators
- 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