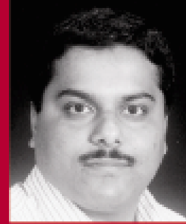
PROGRAMMER TO PROGRAMMER ™



PROFESSIONAL EJB

Rahim Adatia, Faiz Arni, Kyle Gabhart, John Griffen, Matjaz B Juric, Jeremiah Lott, Tim McAllister, Aaron Mulder, Vaidyanathen Nagarajan, Daniel O'Conner, Ted Osborne, P G Sarang, Andre Tost, Dave Young

Table of Contents

Introduction **1**
Chapter 1: The Enterprise JavaBeans Architecture **7**
Chapter 2: EJB Development **29**
Chapter 3: Developing Session Beans **63**
Chapter 4: Developing EJB 1.1 Entity Beans **105**
Chapter 5: The EJB 2.0 Entity Model **149**
Chapter 6: Developing EJB 2.0 CMP Entity Beans **185**
Chapter 7: Asynchronous EJBs **263**
Chapter 8: Resource Management and the EJB Environment **297**
Chapter 9: Transactions and EJB **363**
Chapter 10: Security in EJB **417**
Chapter 11: EJB Design Strategies **471**
Chapter 12: Common EJB Design Patterns **501**
Chapter 13: UML Modeling and EJBs **527**
Chapter 14: Testing Enterprise JavaBeans **587**
Chapter 15: EJB Performance and Scalability **665**
Chapter 16: Advanced Bean-Managed Persistence **715**
Chapter 17: The EJB Container **801**
Chapter 18: J2EE Applications **843**
Chapter 19: COM-Based EJB Clients **905**
Chapter 20: Integrating EJBs and CORBA **929**
Chapter 21: Wireless EJB Clients **959**
Chapter 22: EJBs as Web Services **999**
Appendix A: The Recipe Beans **1051**
Appendix B: The J2EE Reference Implementation **1075**
Appendix C: WebLogic Server 6.0 **1107**
Appendix D: IBM WebSphere Application Server 4.0 **1127**
Appendix E: SilverStream Application Server **1155**
Appendix F: Sybase's EAServer **1179**
Appendix G: JBoss **1197**
Index **1215**

9

Transactions and EJB

Transactions play an integral role in the development of any non-trivial application. This is certainly true of EJB-based systems. The management of transactions is known as **transactional processing**.

Transactional processing can be a very complex feature to implement. However, EJB developers can consider themselves fortunate that the EJB specification requires that the container provide some measure of assistance. This doesn't necessarily mean that they do not have to be concerned with how transactions work. On the contrary, understanding transactions is vital to the development of any enterprise-class system.

This chapter aims to develop a thorough understanding of transactions, transactional systems, and how to effectively use them to improve the scalability, reliability, and dependability of EJB-based applications. It does so by examining how transactions are used within EJB systems. We begin with a discussion of just what transactions are, how they work, and what we can expect from them. Then we move on to cover the following topics:

- How transactions work in distributed systems, an area of interest in itself
- How Java supports transactions
- The API for transactions: JTA, the Java Transaction API
- EJBs and transactions

First, let's cover some basics of transactional programming.

What is a Transaction?

Simply put, a transaction is a single unit of work, composed of one or more steps. These steps have a logical relationship to each other and must be considered as a whole. Put another way, each step within a transaction depends on the success or outcome of the steps that precede it.

In the real world, a transaction represents an exchange between two entities. When you go to the grocery store and buy a loaf of bread, you are conducting a transaction. The bread must be removed from the shelf, brought to the register, paid for, put in a bag, and then removed from the store.

When you order a book online, you place your order for the book, the book is removed from inventory, your credit card is processed, the book is shipped, and you receive it.

In a banking example, we would put a hold on the account so no one else could interrupt us, check to see if there is enough money in the account, debit the appropriate amount, record the account activity, and then release the account for others to use.

We need to treat these steps as one single unit of work because each step requires the successful completion of the previous step. If one of the steps fails (for example, because of insufficient funds), we must undo any changes that were made by the previous steps. Also, we must ensure that there are no interruptions between steps.

> **So, a transaction is a single unit of work that embodies the various individual steps that comprise a compose exchange.**

An application or system that uses transactions is said to be a **transactional system**. The component that manages and coordinates transactions across a system is a **transaction manager** or **transaction processing (TP) monitor.** Some examples of popular TP monitors include IBM's CICS, BEA's Tuxedo, and TOP END, which was acquired by BEA from NCR in 1998.

There are many characteristics to transactions. How they work, when they're used, what behaviors they exhibit, and so on. But before we jump into the details, let's examine some of the reasons for using transactions in the first place.

Why do we Need Transactions?

Let's consider a scenario where an airline reservation application is used to book a seat on a flight. There are several steps that have to be taken to process a reservation:

- First, an appropriate flight must be found.
- Then, an available seat on the flight must be located.
- Having found the seat, the reservation is then begun. This requires getting the name of the customer as well as their credit card information, meal preference, etc.
- Once this information has been verified, the reservation can be made.

So far so good, but what happens if, while the travel agent is processing the customer information, another travel agent comes in and books the seat for another customer?

Without any proper concurrency control, it is entirely possible that the following could happen:

- Agent 1 verifies that a seat is available and begins collecting Customer 1's information.
- Meanwhile, Agent 2 also sees that the seat is available and begins collecting Customer 2's information.
- Agent 2 finishes processing Customer 2's information first and, thus, confirms the reservation.
- Agent 1 also finishes the reservation.

The result? At the very least, two travelers are inconvenienced. At the worst, the travel agents just lost two customers and, quite possibly, the airline has lost them as well.

You can easily imagine how important it is to prevent other situations such as two people trying to draw funds out of a bank account at the same time, especially when that account only has enough money to satisfy one request. Or two processes trying to update the same set of data in a database simultaneously.

Obviously this cannot be allowed to happen. Transactional processing provides us with a way to prevent scenarios such as these. Transactions provide a structured method for controlling the execution of these processes, ensuring their success or else handling their failure.

There are other reasons for using transactions. As we have already seen, one use is when we need to group various steps together into a single unit of work. This unit must either be completed successfully or not at all.

Concurrent applications, those that support multiple users at the same time, have a number of interesting issues that must be addressed. Access to data by multiple clients must be managed and coordinated. Failure to do so can, at best, yield some unexpected results and, at worst, corrupt our data.

As more and more transactions are executed, concurrency control becomes more important and crucial to the well-being of our system. Steps must be taken to ensure that work done by one client does not affect work done by another client. Likewise, changes made by one client must be shared among all clients that rely on that data. Transactions provide a mechanism by which our applications can be aware of when these changes occur.

Transactions allow us to control how several applications access the same data. It can be determined whether a client is allowed to write or even read a given set of data. Additional granularity, or control, is supported in that you can determine how a transaction's relationship to that data is controlled.

Larger systems, especially EJB systems, also have scalability needs. It is quite common, encouraged even, for various resources be located on different servers. As a system grows and provides services for more and more users, the system will have to be maintained on more and more servers in order to handle the workload. As the application "scales-up", it becomes more difficult to manage and coordinate the concurrent data requests.

Transactions help to alleviate the complexity of coordinating data or resource access across many clients.

By their nature, transactions can effect changes on a system. These changes, once they have been completed, must be recorded in **transaction logs**. These transaction logs are used to support failure resolution. That is, in the event of an error or loss of data, the transaction logs can be used to rebuild or recreate the data changes from a previous backup. This is actually much more difficult in practice, but the theory is sound.

By now you have no doubt noticed that transactions exhibit certain properties – characteristics that are common across all transactions. These characteristics are commonly referred to as the **ACID properties**.

The ACID Properties

Transactions exhibit four main characteristics: **Atomicity**, **Consistency**, **Isolation**, and **Durability**. Compliance with these four properties is required in order for an entity to be considered a transaction. Why? We need to be assured that things will work as we expect them to.

Let's take a look at each of the four ACID properties.

Atomicity

Atomicity implies that a transaction will be treated as a single unit of work. You will recall that we said that a transaction could consist of several steps. By treating these steps as a single unit, we are able to put logical boundaries around a process. This will allow us to require that each step must complete successfully or the next step will not be allowed to proceed.

Each step relies on the successful completion of the step before it. Should one step fail, none of the remaining steps will be completed. This makes sense when you apply it to our real-world shopping scenario. When you want to buy that loaf of bread, and the store is out of bread, you stop the transaction and do not complete the purchase.

Likewise, in a banking application, if a check is being drawn on an account, and that account does not have sufficient funds, we halt the transaction. We do not continue processing as if the transaction were successful.

Equally as important is the requirement that, in the event of failure, any changes that were made along the way must be undone. This must be accomplished in a way that does not have any effect on the data or its clients.

A transaction will allow us to indicate the beginning and ending steps of this logical grouping. If all of the steps complete successfully, then any changes made along the way are made permanent, or **committed**. Should any one step fail, the changes are undone, or **rolled back**.

Consistency

Consistency guarantees that the transaction will leave the system or data in a consistent state. We say that data is in a consistent state when it adheres to the constraints, or rules, of the database. A **constraint** is a condition that must be true about the database when a transaction has been completed. Constraints are defined as part of the database schema and specify such things as primary keys, valid ranges for a numerical field, and whether a field may contain a null value, among others.

If the system was in a consistent state when the transaction began, it must be in a consistent state when the transaction ends. This is regardless of whether the transaction succeeds and is committed, or fails and is rolled back.

Consistency is determined against a set of business rules or integrity constraints. In order to ensure consistency, a dual effort is required by both the transaction manager and the developer of the application.

The transactional manager does its part by ensuring that a transaction is atomic, isolated, and durable (the latter terms we will discuss in a moment).

The application developer helps to ensure consistency by specifying primary keys, referential integrity constraints, and other declarative conditions.

When a transaction is to be committed, its consistency is validated by the DBMS against these rules. If it is determined that the results of the transaction will be consistent with the rules set forth by the system, the transaction will be committed. If the results do not satisfy the requirements, the transaction will be rolled back.

Consistency ensures that any changes that our transaction makes will not leave the system in an invalid state. Thus, our transactions will be more reliable. If our transaction is moving money from one account to another, it will be consistent if the total amount of money in the system remains the same (that is, the same amount of money that is removed from the first account is added to the second account).

Isolation

The isolation property of transactions provides us with one of the most powerful characteristics of transactional processing. Simply put, isolation guarantees that any data that a transaction accesses will not be affected by any changes made by other transactions until the first transaction completes.

This effectively allows our transaction to execute as if it were the only one in the system. Other database operations that are requested will only be permitted to proceed if they do not compromise the data that we are currently working with. This is vital to supporting concurrent access to data.

Due to the great opportunity for error and data corruption within a concurrent system, transactional processing must be made simple for a developer to utilize.

With the transaction manager acting as the coordinator of the executing transactions, the responsibility for ensuring isolation and managing concurrent execution is shifted away from the developer.

There are several different levels of isolation that can be applied to a transaction. Which one you use will be determined by a number of different factors. We'll examine these levels of isolation a little later on.

Durability

The durability property specifies that when a transaction is committed, any changes to data that it made must be recorded in permanent storage. This is typically implemented using a **transaction log**.

Transaction logs store data that a resource, such as a database, can use to either re-apply transactions that might have been lost as a result of some failure, or rollback transactions due to an error. Basically, a transaction log keeps track of every data operation that has occurred in the database so that the data can be returned to a known (uncorrupted) state. Once the system has been restored to a known state, the log can be used to reconstruct or replay the changes made since that state.

Durability is also very important when you consider that a committed data change is essentially a binding contract. The commitment of a transaction represents an agreement between the data source and the application. The transaction log is effectively the written record of that agreement. This is exceptionally useful when you remember that the changes themselves are not permanent – another transaction can subsequently change the data.

This analogy of legal compliance isn't just figurative, however. Some laws require that an audit trail of all data changes be maintained for as many as seven years. This is especially true within financial applications. Transaction logs provide the means by which to do this.

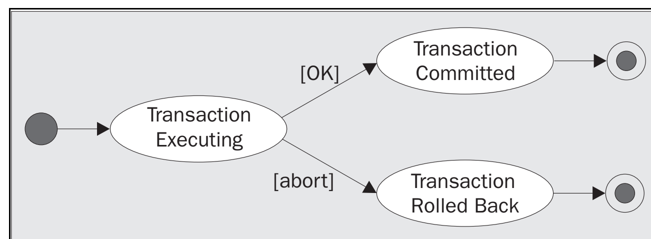
Transactional Processing

Let's take a look at how transactions work in code. You may already be familiar with transactional processing in databases. While not the only use for transactions, it certainly is the most common.

A transaction follows the following general flow. We shall examine this line-by-line:

```
Begin Transaction
    // Do what needs to be done
    if error, rollback
    else commit
End Transaction
```

The following state diagram helps to illustrate the general flow of a transaction:



First, we indicate the beginning of the transaction. This signals that whatever instructions follow should be contained within the transaction. This procedure of identifying the beginning and ending of the transaction (also referred to as **transactional boundaries**) is known as **demarcation**:

```
Begin Transaction
```

Next we put the actual instructions that we want to have controlled by the transaction. Different design scenarios will impose different rules and restrictions on what we can do. We'll examine some of those scenarios in a little bit:

```
// Do what needs to be done
```

Once we have executed our instructions, we check to see if any errors have occurred. If they have, we rollback any changes that have been made, thus undoing the transaction:

```
if error, rollback
```

Otherwise, we commit our changes, thus making them permanent:

```
else commit
```

We then end the transaction, marking the ending boundary:

```
End Transaction
```

Now, this example is intentionally simplistic, but the point here is to understand the basic mechanism. Ideally, we would do things a little more efficiently, such as checking for an error after each instruction is executed, so as not to waste time executing steps that might be unnecessary had an error occurred somewhere along the way.

In Java, of course, we can rely on exception handling to provide a more programmer-friendly means of detecting errors that might occur while a transaction is being processed.

Transactional Processing in the Real World

Before we jump into EJB transactions, let's familiarize ourselves with how transactional processing works in a more general sense.

In the examples that follow in this chapter, we will take a look at two functionally similar but structurally different solutions. The first solution will be the traditional method of database programming via JDBC. The second will focus on an EJB-based implementation. The two examples will use the ubiquitous banking example that we are already familiar with.

The following sample program will transfer funds from one bank account to another. The `transferFunds()` method will start a transaction, call methods to withdraw and deposit funds, and then end the transaction by either committing or rolling back, as necessary.

We'll create a simple account table containing two fields: an account ID and a balance. Notice that we have also placed a constraint on the table such that the balance is not allowed to drop below zero. The SQL to create the database structure for this example is as follows:

```
CREATE TABLE account (
                      AccountId int,
                      Balance    double,
                      check (Balance >= 0)
);

INSERT INTO account (AccountId, Balance) values (1, 100);
INSERT INTO account (AccountId, Balance) values (2, 0);
```

The account table should now contain the following data:

```
AccountId   Balance
---------   -------
        1       100
        2         0
```

The Bank Class

Now we'll implement the following `Bank` object. First, we'll create a general method for getting connections to a database:

```
import java.sql.*;

public class Bank {

  public Connection getConnection(String jdbcDriverName,
                                  String jdbcURL) throws Exception {

    try {
      Class.forName(jdbcDriverName);
      return DriverManager.getConnection(jdbcURL);
    } catch (ClassNotFoundException e) {
        System.out.println(e);
    }
    return null;
  }
```

Next we have a simple routine for displaying all of the accounts in the database and their balances:

```
  public void printBalances(Connection conn) {

    ResultSet rs = null;
    Statement stmt = null;

    try {
      stmt = conn.createStatement();
      rs = stmt.executeQuery("SELECT * from Account");

      while(rs.next())
        System.out.println("Account " + rs.getInt(1) +
                           " has a balance of " + rs.getDouble(2));

    } catch(Exception e) {
        e.printStackTrace();
    } finally {
```

Let's clean up our JDBC objects:

```
      if(rs != null) {
        try {
          rs.close ();
        } catch(Exception ex) {
```

```
          System.err.println(ex);
        }
      }

      if(stmt != null){
        try {
          stmt.close ();
        } catch(Exception ex) {
          System.err.println(ex);
        }
      }
    }
  }
```

The following method, `transferFunds()`, is called when funds need to be transferred:

```
  public void transferFunds(int fromAccount, int toAccount, double amount,
                            Connection conn) {

    Statement stmt = null;

    try {
```

The following lines denote the beginning of the transaction, which continues until the `conn.commit()` method is reached. In JDBC, all database statements are contained within a transaction by default. Setting the connection's `autoCommit` property to `false` allows us to manually control the transaction. In the code below, we call two methods that update the database – `withdraw()` and `deposit()`. Since we want these two statements to be contained within the same transaction, we have to set the `autoCommit` property to `false` and manage the transactions ourselves:

```
      conn.setAutoCommit(false);  // Beginning of transaction
      withdraw(fromAccount, amount, conn);
      deposit(toAccount, amount, conn);
      conn.commit();
```

During the execution of our routine, if an exception is thrown, we will need to undo any database updates that we have executed. The `rollback()` method, found within our `catch` block, does just that:

```
    } catch(Exception e) {
      try {
        System.out.println("An error occurred!");
        System.out.println(e);
        conn.rollback();
      } catch(Exception ex) {
          System.out.println(ex);
      }
    }
  }
```

The deposit() and withdraw() methods take three arguments each: the account number, the dollar amount to process, and a reference to the database connection. The two methods share the same database connection, thus ensuring that the transaction will be propagated, or associated, with each method. Both methods throw a SQLException so that the calling method can rollback the transaction. We could optionally handle the exception within the method and then re-throw the same method or throw a different, more business-specific exception if we wanted. However, this is sufficient for now:

```
    public void deposit(int account, double amount,
                        Connection conn) throws SQLException {

      String sql = "UPDATE Account SET Balance = Balance + " + amount +
                   " WHERE AccountId = " + account;
      Statement stmt = conn.createStatement();
      stmt.executeUpdate(sql);
      System.out.println("Deposited " + amount + " to account " + account);

    }
```

```
    public void withdraw(int account, double amount, Connection conn) throws
      SQLException {

      String sql = "UPDATE Account SET Balance = Balance - " + amount +
                   " WHERE AccountId = " + account;
      Statement stmt = conn.createStatement();
      stmt.executeUpdate(sql);
      System.out.println("Withdrew " + amount + " from account " + account);

    }
```

Notice that neither the deposit nor the withdraw methods are aware that they are part of a transaction. If we were to invoke either of these methods on their own, say if we were making a bank deposit, we wouldn't have to create a transaction for the database operation. This is because, by default, the database connection will automatically create a transaction for it. Only when we need to programmatically control the transaction ourselves do we need to disable the autoCommit feature.

The releaseConnection() method simply frees the database connection. You'll notice that we do not invoke conn.setAutoCommit(true). There is no need: autoCommit will be set to true automatically the next time that a new connection is retrieved from the DriverManager:

```
    public void releaseConnection(Connection conn) {

      if(conn != null){
        try {
          conn.close();
        } catch(Exception e) {
        }
      }
    }

  }
```

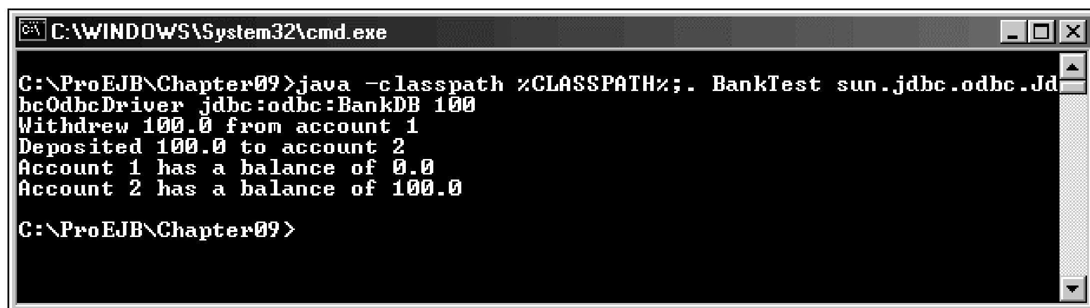
The BankTest Class

Next we have a simple test program. After getting a database connection to work with, we invoke the `transferFunds()` method. Depending on the amount specified from the command line, money will be transferred from Account 1 to Account 2. We then print the balances for all of the accounts in the database. When all is done, regardless of success or failure, we release the database connection.

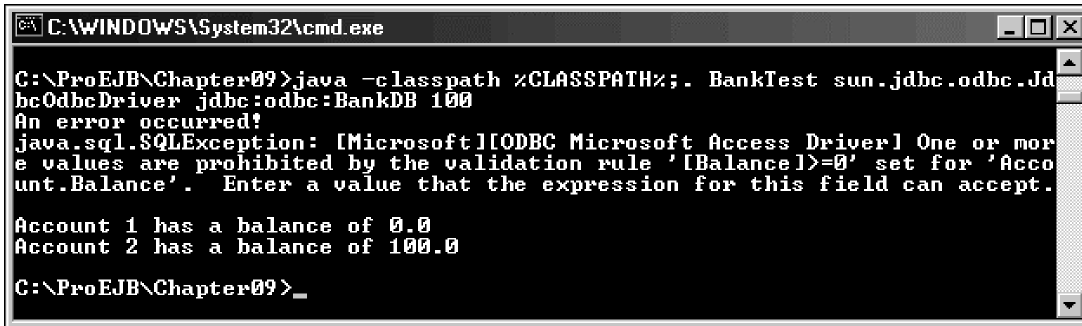
The program takes three parameters: the name of the database driver, the database's URL, and the amount to transfer. For example, if we were using a Microsoft Access database, the driver name would be "`sun.jdbc.odbc.JdbcOdbcDriver`" and the database URL might be "`jdbc:odbc:BankDB`," depending on what we've called the ODBC datasource:

```
import java.sql.*;

public class BankTest {

  public static void main(String args[]) {

    if(args.length < 3) {
      System.exit(1);
    }

    Connection conn = null;
    Bank bank = new Bank();

    try {
      conn = bank.getConnection(args[0], args[1]);
      bank.transferFunds(1, 2, Double.parseDouble(args[2]), conn);
      bank.printBalances(conn);
    } catch(Exception e) {
        e.printStackTrace();
    } finally {
        bank.releaseConnection(conn);
    }
  }

}
```

The first time that we run the program, all will be fine. One hundred dollars is withdrawn from Account 1 and deposited into Account 2. The balances are then displayed on the screen. The results should look like this:



However, if we run the program again, the system will attempt to overdraw the first account thus producing a negative balance. Since this violates the constraints of our Account table `(balance >= 0)`, a `SQLException` will be thrown. This, in turn, will result in the transaction being rolled back. The results will look like this:



Note that the reference to Microsoft Access occurs because we are running this against an MS-Access database via ODBC. You would get a different error message if you used another database.

As you can see, using transactions in standard Java is quite straightforward and easy. You simply demarcate the transaction's boundaries, execute your database code, and either commit or rollback the transaction.

The database drivers handle the actual complexity of watching the operational progress of the transaction and notifying you of any errors that might occur. Of course, you could also check for other conditions that might not indicate a SQL error (failed validation perhaps) but you would most likely want to do this prior to executing your code if possible.

We'll revisit the example above and build upon it as we learn more about some of the finer control options that are available to us. But first, let's take a look at a special type of transaction called a **compensating transaction**.

Compensating Transactions

Sometimes we need to undo a transaction after it has been committed. This is usually not due to a database error, but to the fact that the user just changed their mind. **Compensating transactions** are a strategy for using transactional processing whereby committed transactions can be undone after they have been committed. They contain the necessary logic to reverse the changes made by the original transaction.

Consider the case where a bank teller makes a deposit to an account and, after the transaction has completed, realizes that the wrong account was credited. Here we have an instance where the transaction has committed but needs to be reversed nevertheless. The bank's accounting procedures won't allow us to simply delete the money from the account – we have to account for every transaction that is executed. A deletion would raise quite a few eyebrows. However, we can't simply withdraw the money back out of the account as this might upset the customer when they receive their statement at the end of the month. A compensating transaction can be used to rollback the changes that we have made.

Another use for compensating transactions comes into play when we need to wire funds between different accounts at different banks. Here we have transactions that might not get resolved quickly. In fact, wire transfers can take hours or even a few days to complete – these are referred to as **long-lived transactions**. Regardless, they still need to be protected by a transaction. However, we cannot simply tie up our resources while we wait for the transaction to complete. What to do?

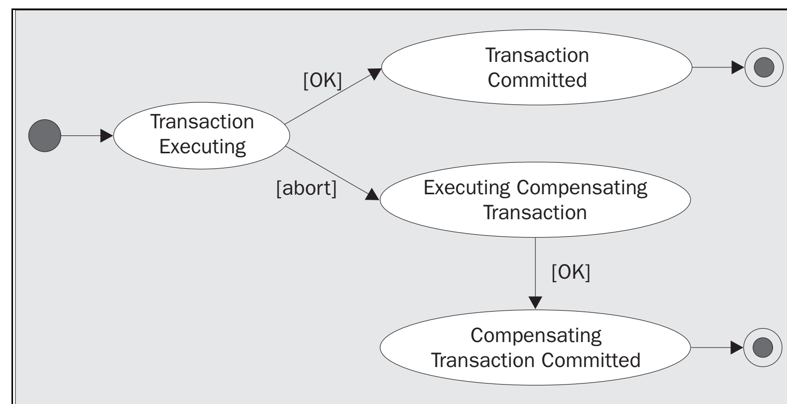
We can break the transaction up into a series of smaller transactions. Each transaction must be completed in series. The transactions themselves might look like the following:

- Withdraw funds from source account
- Wire funds to federal exchange (an intermediary between the two banks)
- Wire funds from exchange to destination bank
- Deposit money into target account

Here we have four distinctly different transactions, with the second and third capable of taking a considerable amount of time. Each transaction is executed in series and can be committed. However, should one of the transactions need to roll back, we'll need a way to roll back the previously committed transactions.

By associating a compensating transaction with each of the individual transactions, we have all the information necessary to roll back each of the committed transactions, if necessary. In J2EE, we should use compensating transactions when a particular resource (such as a flat text file) does not support JTA-managed transactions.

The following state diagram illustrates the transaction life cycle using compensating transactions:



Now that we have a good understanding of how transactions work, let's take a look at some of the different ways that transactions can be used.

Transactional Models

There are many different types of and uses for transactions. How a transaction is used and how it is structured typically falls in to one of many transactional models. We'll take a look at four here, namely:

- Flat transactions
- Nested transactions
- Chained transactions
- Saga transactions

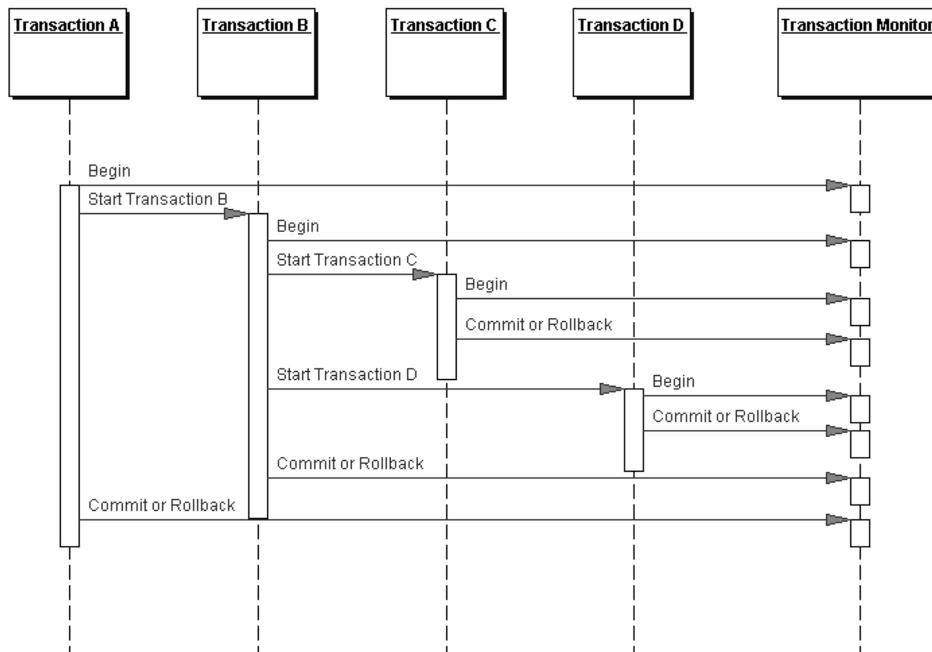
We'll start with flat transactions.

Flat Transactions

A single or atomic unit of work, composed of one or more steps, the flat transaction is the simplest of transactions. Should one of the steps fail, the entire transaction is rolled back. Flat transactions are the de facto standard for database operations, as well as Enterprise JavaBeans in general.

Nested Transactions

The nested transaction model has atomic units embedded in other transactions. This has the effect of a transaction "tree" – a root transaction that contains several branches of other transactions. The nested transaction differs from the flat transaction, in that should one of the sub-transactions roll back, it will not affect the parent transactions. In other words, the failure of a transaction is limited to just that transaction. A sub-transaction can be either a flat transaction or another set of nested transactions. The following sequence diagram illustrates using nested transactions. After Transaction A begins, it starts Transaction B. Transaction B, in turn, invokes first Transaction C, then Transaction D.



Another example of nested transactions might be an airline reservation system. Let's say that we want to book a flight from New York to San Francisco. Our first transaction (T) entails our attempt to book a direct flight. No direct flights are available, so our transaction fails. This is a flat transaction.

We then start a new transaction (U) and try to find a flight that is available between New York and Chicago. We want to hold this reservation, so we keep the transaction open. We then begin a new transaction (V), with our current one, that will attempt to book a flight from Chicago to San Francisco. There are no direct flights between the two cities, so our transaction fails and rolls back. However, since the transaction is localized at Chicago, we will not lose the New York booking. Remember, nested transactions do not roll back the transaction in which they are contained.

Another transaction (W) is started, this time attempting to book a flight between Chicago and Dallas. We are successful in finding a reservation, so we hold transaction W open.

Finally, we initiate a new transaction (X) in hopes of finding a flight from Dallas to San Francisco. If we are successful, we commit transaction X. Transaction W checks the status of transaction E and, seeing that it was successful, W commits. This continues with W's parent, transaction U, and then T. With all of the transactions committed (T, U, W, & X), we have booked our flight.

If we had decided that four flights were too many and decided to scrap the whole affair and take the train instead, X would roll back, W would see that X failed and, as part of a business rule to not accept this many layovers, would roll back, and up and up until the entire tree of transactions (T, U, W, & E) had been aborted and the entire transaction cancelled.

Chained Transactions

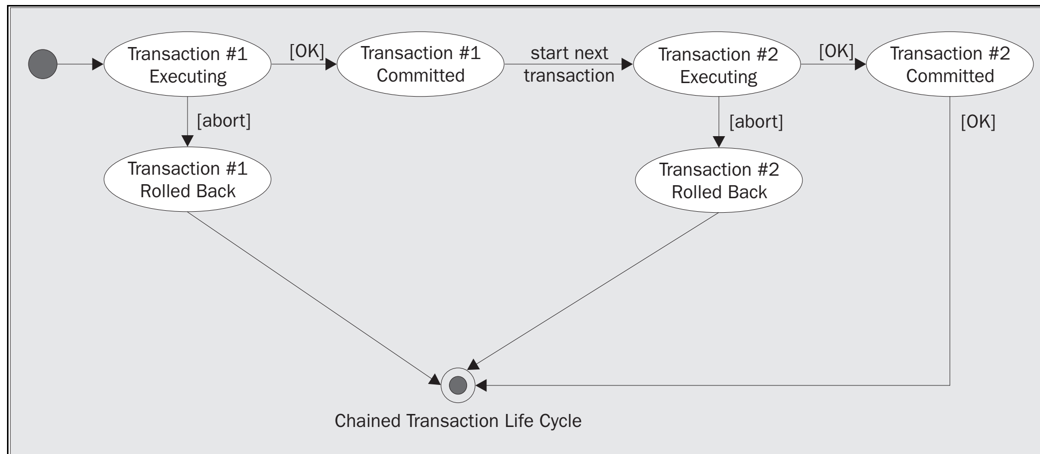
Chained transactions, sometimes referred to as **serial transactions**, are a set of contiguous transactions related together. Each transaction relies on the results and resources of the previous transaction. Typically, the developer identifies the boundaries of each transaction and then submits all of the transactions as a group. The transaction manager then executes each transaction in series, one after the other.

With chained transactions, when a transaction commits, its resources (say, cursors) are retained and immediately made available to the next transaction in the chain – effectively combining the `commit()` and the next `beginTransaction()` into a single atomic step. The result is that transactions outside of the chain cannot see or alter the data being affected by the transactions in the chain. This is different from committing a transaction and then starting a new one in two separate steps, as that would result in the resources used by the first transaction being released.

Another advantage of chained transactions is that resources (such as locks), used by one transaction and that are not required by subsequent transactions, can be released. This is more efficient than keeping the resources for the full length of the chain.

If one of the transactions should fail, only the currently executing transaction (the one that failed) will be rolled back – the rest of the previously committed transactions will not.

The following state diagram will give you an idea of the general progression through a chain of transactions:



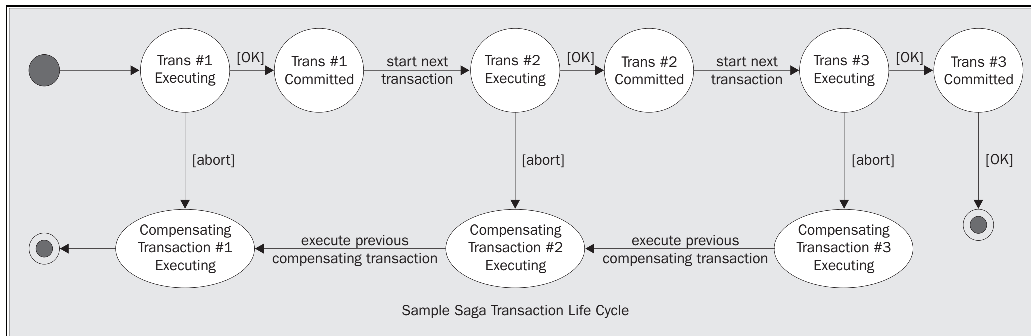
Chained Transaction Life Cycle

The downside to chained transactions is that they can potentially lock up a lot of valuable database resources, thus slowing the system down. Careful thought should be given when using these transactions.

Sagas

Sagas are long-lived transactions that model workflow processes – similar to chained transactions, in that they are composed of multiple transactions, each of which has a corresponding compensating transaction. In the event that one of the transactions fails, the compensating transaction for each transaction that successfully ran is invoked. This is done automatically by the transaction manager. Therefore, before execution of the saga, all of the associated transactions and their relationships must be identified to the transaction manager.

The state diagram below illustrates a sample saga that consists of three transactions:



Sample Saga Transaction Life Cycle

Other Transaction Models

There are many other transaction models and strategies, but these are the most common. However, the J2EE specification only requires that flat transactions be supported. The reason this limitation was placed was to make EJB available to a wider selection of database systems, as not all of them support nested transactions. Many transactional processing systems also provide support for nested transactions. If this functionality is required, you will need to pay close attention to the features offered by a given server. You should also remember that any time you use a non-standard feature, you limit portability.

> **Note that nested transactions are *not* supported by EJB. In fact, the EJB specification requires that a `javax.transaction.NotSupportedException` be thrown if a bean starts a transaction while an existing transaction is already in progress.**

Transactions and EJB

As we already know, Enterprise JavaBeans provide us with a wealth of built-in functionality that we would otherwise have to develop ourselves, and transactional processing is no exception. However, like most things, we have to be willing to accept certain limitations in what we are able to do.

The J2EE's implementation of transactional processing is made available through the **Java Transaction API**, or **JTA**. JTA is an interface that provides access to an underlying transaction manager. Support for transactional processing is a standard requirement of all EJB servers, although the level of support for transactions varies. We'll examine some of those differences as we advance through this chapter. As we will shortly see, EJB offers a wide array of options when using transactions.

Demarcation

The most fundamental decision that you will need to make is whether to manage the transaction yourself or to let the EJB container manage it for you. If you are using an entity bean, the decision is an easy one – as of the EJB 1.1 specification, you are only allowed to use container-managed transactions (this is not to be confused with container-managed persistence, which is responsible for data retrieval and storage). However, for session beans, you can allow the container to manage the transactions or take on the task yourself. Either way, you will have a great deal of flexibility available to you.

The process of determining where a transaction begins and ends is called **demarcation**. If you decide to let the container manage your transactions, you are using **declarative demarcation**. They are called declarative transactions because you will declare to the container how a transaction should behave. If you opt to manage the transactions on your own, you are using **programmatic demarcation**. Programmatic demarcation is the "traditional" method of transactional programming.

Why would you choose one over the other? Declarative demarcation is simpler to use, relieving you of having to incorporate transactional processing into your code. Of course, you are not absolved of having to use transactions completely. You will still have to configure the characteristics of the transaction. However, you specify the transactional properties in the deployment descriptor.

Declarative demarcation differs from programmatic in where the boundaries are placed. With programmatic demarcation, you determine exactly where the transaction will begin and end. With declarative demarcation, the container is not able to determine the boundaries within your code. Rather than try to guess at what your intentions are by examining the instructions, it simply places the entire method into the transaction.

This should immediately suggest to you a potential source for problems. If your method invokes a long-running process (like a large loop), you will be blocking other transactions from getting access to your data. Even if the loop might seem small to you, you must remember that small delays grow to large bottlenecks very quickly in a concurrent system. Therefore, the smallest gains can have performance improvements of an order of magnitude. If your session bean's method has a slow process within it, you should consider isolating the database calls yourself and using programmatic demarcation.

Since the majority of the transactional processing that you will most likely be doing in EJB will be declarative, we'll focus on how that works first. Afterwards, we'll return to programmatic demarcation for some more advanced techniques.

Transaction Attributes

With declarative demarcation, you specify to the container which methods to subject to transactional processing. In addition, you also specify certain behavioral properties that control how and when the transaction will be created, depending on the needs of our component in relationship to the **transactional context**. Transactional context refers to the relationship between the transactional operations conducted on a given set of resources, and the clients invoking those resources. A transactional context embodies the transaction from the point of inception by the client, through to its completion. Any operation that occurs on a resource capable of participating in a transaction is said to occur within the context of that transaction, and will be subject to the rules of that context.

Enterprise JavaBeans support seven types of transactional attributes:

- `Required`
- `RequiresNew`
- `Mandatory`
- `Supports`
- `NotSupported`
- `Never`
- `"Bean-managed"`

Required

The `Required` transactional attribute indicates that the specified method must always be run within a transaction. If the method is called within the context of an existing transaction, that same transaction will be used, or propagated. If no transactional context exists, the container will create a new one. The new transaction will begin when the method begins and will commit when the method ends. If a rollback is required, the container will undo the transaction and throw a `RollbackException`. This will allow the calling method an opportunity to respond to the failure.

`Required` should be used whenever your method will be changing data, so as to ensure that the data operation will occur within and be protected by a transaction.

RequiresNew

The `RequiresNew` transaction attribute indicates that the specified method must always be executed within its own transaction. This would be used when a transaction is required, but it is not desirable for a local rollback to affect transactions outside of the method – the failure is localized. Another use is when the transaction must commit its results regardless of the outcome of the outer transactions – logging, for example.

The container will create a new transactional context for the method before it is invoked. If the method is called within the context of an existing transaction, that first transaction will be suspended and a new transaction will begin. Once the method completes, the container will commit the new transaction and then reinstate the pre-existing transaction.

Mandatory

The Mandatory transaction attribute indicates that your method can only be invoked within the context of a pre-existing transaction. Therefore, it is mandatory that the method's client must already have started a transaction. The transaction context will be propagated to your method. If a component tries to invoke the method and does not have a transactional context already associated with it, the container will throw a TransactionRequiredException or a TransactionRequiredLocalException.

We should use the Mandatory attribute when our method needs to verify that the component was invoked within the context of a client-managed transaction.

Supports

The Supports transaction attribute tells the container that the associated method will use a transaction if one is already available. However, if one is not previously available, that is, if a transactional context does not already exist, then the method can be invoked without a transaction.

We would use the Supports attribute when we do not want to incur the processing overhead of suspending and resuming a pre-existing transaction and are confident that our method will either not cause an exception or will not cause an exception that would signal a failure within the context of a transaction. Furthermore, we must be certain that our transaction will not violate any data constraints that might otherwise cause a transaction to fail.

> **It is imperative to remember that, most likely, we will not know what component is invoking our bean's method and what the intention of that component is. We must always be careful to consider that our bean may be used in ways that we might not have originally intended.**

Given this, it is better to support transactions and to let the client decide how failure within your bean should affect them.

NotSupported

The NotSupported transaction attribute tells the container that the method should not be run within a transaction. If the method is called within the context of an existing transaction, the transaction will be suspended before the method is invoked.

If the business method invokes other EJB methods, the container will not pass a transactional context with the invocation. In other words, if a bean's method is indicated as not supporting transactions, and the method is invoked from within a transaction, the transactional context will not "pass through" the method. Should an exception occur, it would not affect the suspended transaction of the caller. However, once the method has completed executing, the container will resume the original transaction. The NotSupported attribute should also be used when an enterprise bean needs to interact with a resource manager that does not support transactions. In this particular situation, it is recommended that the NotSupported attribute be used in all of the bean's methods.

Never

The Never transaction attribute indicates that the method should never be called within the context of another transaction. The Never attribute should be used when the component needs to verify that the method was not invoked within a client-managed transaction and that the container will not attempt to provide a transaction for it.

Never should also be used when a given method is not capable of participating in a transaction. The container will execute the method without starting a transaction for it. If a component invokes the method and a transaction context does exist, the container will throw a RemoteException if the client is remote or EJBException if the client is local. This differs from NotSupported, in that NotSupported will simply suspend a transaction if one already exists, resuming upon completion of the method. Never will throw an exception if a transaction exists.

"Bean-managed"

The absence of a transactional attribute in the deployment descriptor indicates that the bean will manage its own transactions. This can only be done by session beans or message-driven beans. Furthermore, a component cannot have some of its methods managed by the container, and others managed by itself.

> *Note: In EJB 1.0, there was an attribute to explicitly indicate that a transaction was bean-managed. TX_BEAN_MANAGED was a static value of the ControlDescriptor object – part of the DeploymentDescriptor object structure. In EJB 1.1, the DeploymentDescriptor was deprecated in favor of the XML-formatted deployment descriptor. The absence of a transaction entry for a given method in the deployment descriptor implies that the method's transaction is bean-managed.*

The following table summarizes the transactional attributes and the effect that they have on a component's method's transaction context, in relationship to a calling client's transactional context:

Transaction Attribute	Client's Transaction	Component Method's Transaction	Comment
Required	None	T2	If no transaction exists, the server creates one
	T1	T1	If a transaction exists, the server uses it
RequiresNew	None	T2	If no transaction exists, the server creates one
	T1	T2	If a transaction exists, the server creates a new one
Mandatory	None	Error	If no transaction exists, an exception is thrown
	T1	T1	If a transaction exists, the server uses it
Not Supported	None	None	The server does not provide transactional support

Transaction Attribute	Client's Transaction	Component Method's Transaction	Comment
	T1	None	The server does not provide transactional support
`Supports`	None	None	If no transaction exists, the server does not provide support
	T1	T1	If a transaction exists, the server uses it
`Never`	None	None	The server does not provide transactional support
	T1	Error	If a transaction exists, an exception is thrown

Transactional Attribute Usage

The differences between the various transactional attributes are sometimes slight and it can be confusing trying to determine when to use which one. Which attribute is selected will undeniably be the result of which ACID properties are required and the extent to which they are needed.

For example, `RequiresNew` provides a greater level of atomicity and isolation, albeit at the cost of performance. This is due to the overhead of creating a new transaction and managing the existing one.

Another factor that will affect which attribute is used will be the level of transactional support that is provided by the transactional processor in question. For example, if the processor does not support nested transactions, neither will the application.

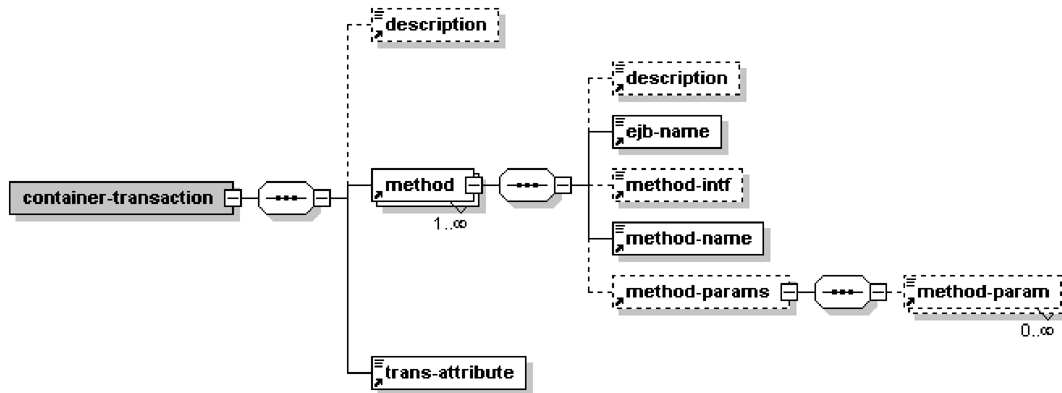
As a general guideline, consider the following:

- Use the `Required` attribute when the code needs to change the value of some data.
- Use the `Supports` attribute when the code needs to read data from a data source. This will allow the caller of the component to determine whether a transaction should be used. This is appropriate because the enterprise bean really has no way of knowing what the intention of the calling method is.
- If we are communicating with resources that do not support transactional processing, then we should use the `NotSupported` attribute.
- When using message-driven beans, the transaction attributes must be specified for the bean's `onMessage()` method.Message-driven beans only support the `Required` and `NotSupported` transaction attributes. They do no support `RequiresNew` or `Supports` as they are never invoked within the context of a pre-existing transaction. Also, they do not support `Mandatory` and `Never`, as message-driven beans are only ever invoked by the container. Therefore, it would not make sense to support client-side transactions.

Declaration of Transactional Attributes

Associating an attribute with a transaction is typically done in one of two ways: either through the EJB container's deployment tool, or by directly including it in the deployment descriptor. Of course, regardless of which method you choose, the result will be the same: the deployment descriptor contains the declaration of all transaction attributes.

Here is the relevant section of the DTD for setting transactional attributes, which comes under the <assembly-descriptor> element:



There are actually three ways to declare a transactional attribute for a given method. However, they all follow the same general format:

```
<ejb-jar>
  ...
  <assembly-descriptor>

    <container-transaction>
      <method>
        <ejb-name>someEJBean</ejb-bean>
        <method-name>someMethod</method-name>
      </method>
      <trans-attribute>transactionalAttribute</trans-attribute>
    </container-transaction>

   </assembly-descriptor>
   ...
</ejb-jar>
```

The deployment descriptor entry starts with <container-transaction> and contains two elements: <method> and <trans-attribute>.

The <method> element contains two additional sub-elements: <ejb-name> and <method-name>. The <ejb-name> sub-element identifies the enterprise bean that the method is part of, while <method-name> identifies the name of the method to apply the transactional attribute to.

The <trans-attribute> element indicates the transactional attribute to apply to the named method, using one of the keywords discussed above: Required, RequiresNew, NotSupported, Supports, Mandatory, Never, or BeanManaged.

Let's take a look at a typical deployment descriptor entry. Here we have an enterprise bean named Customer, with a method getCustomerName(). We have applied the Supports transactional attribute to the method:

```
<container-transaction>
  <method>
    <ejb-name>Customer</ejb-name>
    <method-name>getCustomerName</method-name>
  </method>
  <trans-attribute>Supports</trans-attribute>
</container-transaction>
```

If we have an overloaded method, we have to include the parameter declarations within the deployment descriptor as well.

The following entry applies the Requires transactional attribute to the Customer bean's getProfile() method. The method accepts three parameters, an int, a String and an array of FilterType objects. Thus, the methods signature might look like this:

```
Profile getProfile(int type, String groupId, FilterType filter[]);
```

In the deployment descriptor, notice that the <method-param> entry for the array of FilterType contains an empty bracket, indicating that it is an array:

```
<container-transaction>
  <method>
    <ejb-name>Customer</ejb-name>
    <method-name>getProfile</method-name>
    <method-params>
      <method-param>int</method-param>
      <method-param>java.lang.String</method-param>
      <method-param>FilterType[]</method-param>
    </method-params>
  </method>
  <trans-attribute>Requires</trans-attribute>
</container-transaction>
```

If the getProfile() method were to also have an overridden method that accepted no parameters, the deployment descriptor entry would have an empty <method-params> section, like so:

```
<container-transaction>
  <method>
    <ejb-name>Customer</ejb-name>
    <method-name>getProfile</method-name>
    <method-params></method-params>
  </method>
  <trans-attribute>Requires</trans-attribute>
</container-transaction>
```

Note that for each method with the same name that our EJB has, if we supply a transaction information entry for one, we must supply an entry in the deployment descriptor for each.

Finally, if all of the methods in the enterprise bean are to utilize the same transactional attribute, a shortcut is provided. Simply use an asterisk, *, for the <method-name> value:

```
<container-transaction>
  <method>
    <ejb-name>Customer</ejb-name>
    <method-name>*</method-name>
  </method>
  <trans-attribute>Supports</trans-attribute>
</container-transaction>
```

Now every method within the `Customer` enterprise bean will be associated with the `Supports` attribute.

If the bean's transactions are to be managed by the bean itself, no deployment descriptor entry is required. Remember that only session beans can manage their own transactions. Also, if one method in the bean is bean-managed, they must all be bean-managed. You cannot have some methods managed by the bean and others managed by the container. This would potentially cause problems if one method called another. However, it is perfectly fine for a bean-managed method to call a container-managed method in another bean.

Unspecified Transaction Context

When a method does not have a transactional context associated with it, such as with `NotSupported` or `Never`, that transaction context is said to be **unspecified**. An unspecified transaction context can also result when a method `Supports` transactions, but is invoked without a transactional context.

When handling a method that runs with an unspecified transaction context, we must carefully plan out any data operations that might be performed. However, as it is always a bad idea to manipulate data outside the scope of a transaction, this should generally be avoided. The Enterprise JavaBeans specification does not mandate how a container should handle the execution of a method with an unspecified transaction context. It is important that we review our EJB server's documentation for specifics. However, a container might utilize the following general approaches:

- The container may execute the method, and access any resource managers referenced within, without a transaction context.
- The container may execute each call within the method to a resource manager as an individual transaction.
- The container may combine all calls to one or more resource managers within a single transaction.
- If the bean instance invokes methods on other bean instances, and those instances also have an unspecified transaction context, the container may combine all calls to all resource managers in all of the methods and execute them within a single transaction.

These methods are not absolutes, but instead are possible approaches to managing an unspecified transaction context.

We have to be careful when writing methods that may run in an unspecified transaction context. Since the EJB specification does not define how the container should handle this situation, we cannot rely on any specific behavior. Therefore, we should avoid writing our methods to rely on any set behavior. It is also important to remember that, should a failure occur during an unspecified transaction, any resource managers accessed from the method could be left in an unknown state, or corrupted.

Implementing Bean-Managed Transaction Demarcation

Sometimes it is desirable for a bean to manage its own transaction. Some reasons for managing your own transactions include:

- Maintaining transactional state across multiple methods in a stateful session bean
- Providing transactional behavior for a resource that doesn't provide transactional processing (wire services, e-mail, flat files)
- Providing support for compensating transactions

The Java Transaction API provides an interface for explicitly controlling a transaction – the UserTransaction interface:

```
public interface javax.transaction.UserTransaction {

  public void begin() throws NotSupportedException, SystemException;

  public void commit() throws RollbackException,
                              HeuristicMixedException,
                              HeuristicRollbackException,
                              SecurityException,
                              IllegalStateException,
                              SystemException;

  public void rollback() throws IllegalStateException,
                                SecurityException,
                                SystemException;

  public int getStatus() throws SystemException;

  public void setRollbackOnly() throws IllegalStateException,
                                       SystemException;

  public void setTransactionTimeout (int seconds) throws SystemException;
}
```

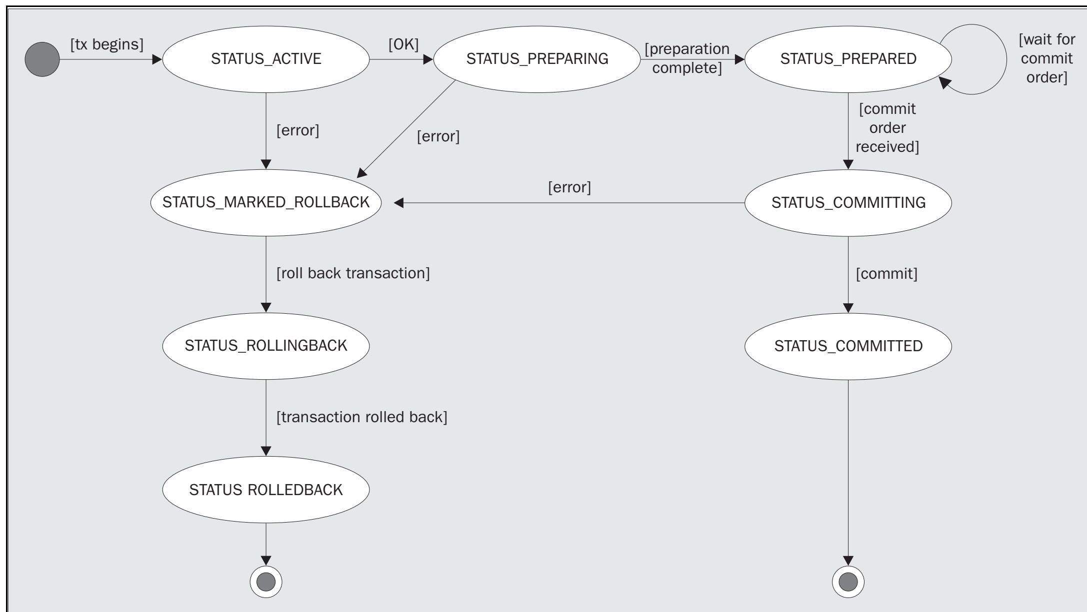
The use of the UserTransaction interface is very straightforward, as detailed below:

1. Begin the transaction by invoking begin()
2. Execute any business logic to be contained within the transaction
3. Complete the transaction by either invoking commit() or rollback()

The setRollbackOnly() method will mark a transaction for rollback without actually rolling the transaction back. This is desirable when other methods within the transaction need to run regardless of whether the transaction will commit or rollback. When the setRollbackOnly() method is used, the transaction's status should be checked before the end of the transaction and handled accordingly. This is done by invoking the getStatus() method which will return the status of the current transaction. getStatus() returns one of the following values (embodied in the javax.transaction.Status interface):

- `STATUS_ACTIVE`
  A transaction is associated with the component and is in the active state.
- `STATUS_COMMITTED`
  A transaction is associated with the component and has been committed.
- `STATUS_COMMITTING`
  A transaction is associated with the component and is in the process of committing.
- `STATUS_MARKED_ROLLBACK`
  A transaction is associated with the component and has been marked for rollback.
- `STATUS_NO_TRANSACTION`
  No transaction is associated with the component.
- `STATUS_PREPARED`
  A transaction is associated with the component and has been prepared for committing in a two-phase commit (discussed later). It is waiting for a `commit` instruction from the transaction manager.
- `STATUS_PREPARING`
  A transaction is associated with the component and is in the process of preparing for a two-phase commit, but has not yet completed the preparation.
- `STATUS_ROLLEDBACK`
  A transaction is associated with the component and has been rolled back.
- `STATUS_ROLLING_BACK`
  A transaction is associated with the component and is in the process of rolling back.
- `STATUS_UNKNOWN`
  A transaction is associated with the component but its status cannot be determined.

The following state diagram shows the relationship of the statuses to the transaction lifecycle:



There are two ways to obtain a reference to the UserTransaction interface. We can get an instance via a JNDI lookup to java:comp/UserTransaction or via the EJBContext.getUserTransaction() method.

The setTransactionTimeout() method sets the timeout value of the transaction that is associated with the current thread. If a transaction does not complete within the specified length of time, a SystemException will be thrown. Generally, the timeout value for a transaction is defined through the application server's configuration. As such, it is better to set the timeout value at the server level, than at the method level.

Restrictions on the UserTransaction Interface

There are certain restrictions to using the UserTransaction interface:

- First, keep in mind that entity beans cannot manage their own transactions and, therefore, cannot access UserTransaction. Only session beans and message-driven beans can.
- A bean instance that starts a transaction must complete the transaction before starting a new one. Nested transactions are not allowed in EJB.
- Code within a transaction must not attempt to use control methods specific to a resource manager. For example, calling commit()on a JDBC connection is not permitted when that connection is obtained via the container's resource factory.
- A stateless session bean must close a transaction in the same invocation in which it was started. As the bean does not maintain state, it will not be able to maintain transactional state, either. This is appropriate behavior for a stateless session bean. If the method exits without finishing the transaction, the following will happen:
  - The server will log an application error
  - The transaction will be rolled back
  - The session bean instance will be discarded
  - A RemoteException or EJBException will be thrown
- A stateful session bean may start a transaction in one method without finishing the transaction before the method invocation ends. That is, the method invocation does not need to call commit() or rollback() before exiting. The container will remember the transactional context across multiple invocations. Thus, subsequent method calls will be invoked within the same transactional context.

  Furthermore, a database connection need not be kept open during the entire transaction. In fact, a method may open and close a connection several times during multiple invocations within the same transaction context. The connections will all be managed within the same transaction provided the transaction is started before all of the database operations and committed afterwards.
- A message-driven bean must close the transaction before the onMessage() method returns.
- A J2EE server is not required to provide access to its transaction manager to an application client. By application client we mean any client that is not contained within the J2EE server itself. Furthermore, if a JDBC transaction on the client invokes an enterprise bean, the context of that client's transaction is not required to be propagated to the EJB server.

What about JTS?

The **Java Transaction Service (JTS)** should not be confused with the Java Transaction API (JTA). JTA serves as an interface between our code and a system's transaction manager. JTS, on the other hand, provides an interface to the CORBA Object Transaction Service (OTS). JTS provides interoperability between transaction managers to support distributed transactions. An EJB server is not even required to provide a client access to its JTS implementation. In practice, JTA is used by application developers and JTS is used by the transaction manager vendors.

Distributed Transactions

So far, we have looked at transactions that are executed against a single resource (one database). This has left us with a number of questions:

- What happens if our transaction needs to execute against multiple resources such as two database servers?
- What if we need to mix resource types within a transaction? For example, a database and a message queue.
- What if we need to have multiple enterprise beans within the same transactional context?
- What issues need to be taken into consideration when working with transactions that are distributed throughout the enterprise?
- How are these **distributed transactions** managed in the real world?

Fortunately for us Java developers, we needn't be too concerned with the physical complexity of distributed transactions. The Java Transaction API (JTA) that we have already learned will suffice quite nicely.

Consider a session bean, `EmployeeManager`, that provides a method `createNewEmployee()`. This method performs several tasks all within a single transaction:

- Create a new record in the human resources employee database
- Create a new record in the accounting system's payroll database
- Send a message to the technical support group requesting the creation of a network ID and e-mail account for the new employee

We are able to include in our transaction not only the two databases, but a message server as well. If any of the tasks were to fail, we would want the entire operation to be rolled back. In order to support transactions across multiple resources, all you need to do is enclose the calls to those resources within a transaction. As long as each resource manager provides support for distributed transactions via JTA, you can be assured of the ACID benefits of transactional processing.

But what exactly are distributed transactions and, especially, how do they work?

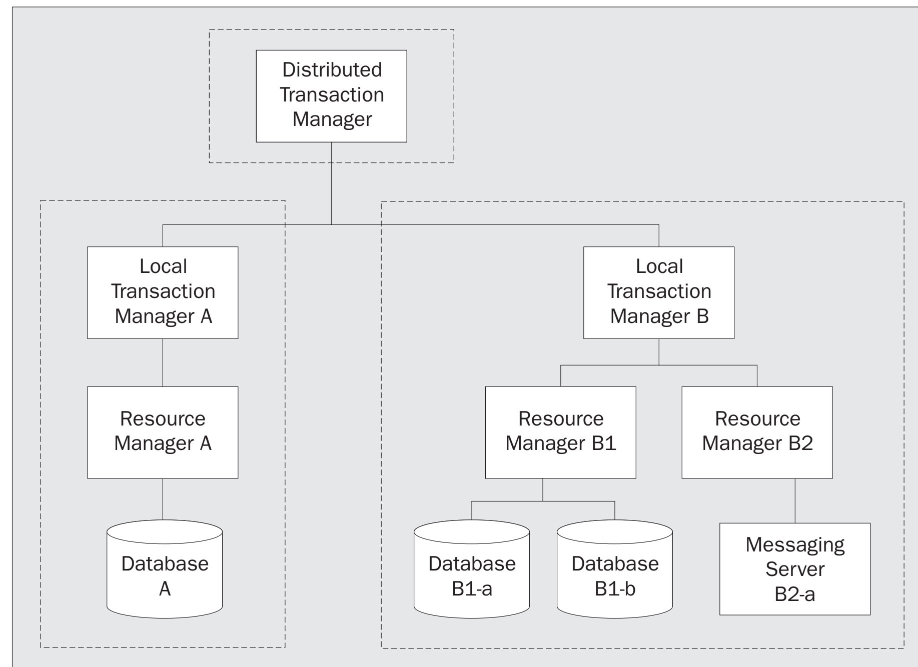
What is a Distributed Transaction?

A distributed transaction is a transaction whose context spans more than one resource and/or whose context is propagated or shared by more than one component. Distributed transactions support scenarios such as:

- A component needs to communicate with multiple resources within the same atomic operation. A bank account session bean might debit funds from an account in one database and credit the funds in another database.
- Multiple components need to operate within the same atomic operation. Our banking system could debit the balance in one account entity bean, credit the funds to another entity bean, and create an audit log entry by calling a third session bean.

A distributed transaction requires the cooperation of several different transaction managers. A master transaction manager known as a **distributed transaction manager** coordinates the other transaction managers. It is the responsibility of the distributed transaction manager to control the propagation, demarcation, and resolution of a single transaction across several participating *local* transaction managers. A local transaction manager is a transaction manager that participates, or is "enlisted", in a distributed transaction.

The following diagram illustrates one possible configuration:



Four resources exist on two different systems. The resources are three databases (named Database A, Database B1-a, and Database B1-b) and a messaging server (Messaging Server B2-a). Database A is managed by Resource Manager A. Typically, the two together compose a database management system (DBMS). Here, they are shown separately to distinguish between the physical data and the component responsible for managing that data. Resource Manager A's transactional context is managed by Local Transaction Manager A. For almost all databases, the transaction manager is coupled with the resource manager.

This topology also applies to Database B1-a and Database B1-b. The messaging server also has an associated resource manager, B2. For the purpose of our example, the database and the messaging server are managed by the same transaction monitor (perhaps they are part of a vendor's integrated solution). However, more often than not, these resource managers will have separate transaction managers.

The Distributed Transaction Manager at the top is responsible for coordinating the efforts of the underlying local transaction managers when the resources associated with the local managers are involved within a distributed transaction.

The Two-Phase Commit Protocol

The **two-phase commit protocol** is a method of communication for the coordination of transactions across multiple servers and/or resources. The Open Group (X/Open, http://www.opengroup.org/) manages a standardized version of the two-phase commit protocol. However, not all vendors support the standardized protocol, preferring to implement their own. EJB supports the X/Open standard via the Java Transaction API. For consistency and compatibility across different platforms, you should make sure that your application server supports the X/Open specification.

When a transaction requires the services of multiple resources, the transactional processing of these resources will be managed using the two-phase commit protocol. Although we have primarily discussed databases, know that any resource manager that supports the two-phase commit protocol can participate in a distributed transaction.

As its name suggests, the two-phase commit protocol consists of two phases: the "prepare" phase and the "commit" phase.

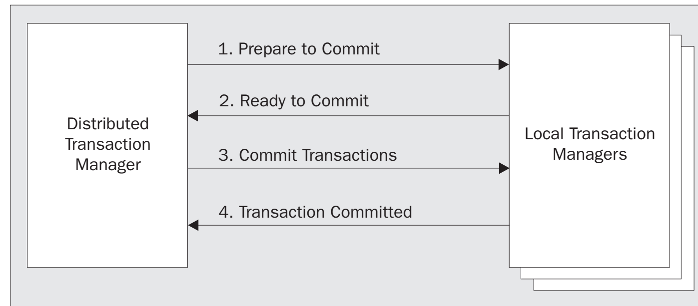
In the prepare phase, the following happens:

- The distributed transaction manager tells the various local transaction managers to prepare for the requested data operations.
- Each local transaction manager writes the details of the data operation to a transaction log. In the event that a failure occurs and the data operation was not submitted successfully to the resource manager, the transaction manager has a 'local' copy from which it can try to recreate this transaction.
- The local transaction managers will create a local transaction and notify their respective resource managers of the operations.
- Once the data operation has been executed, the resource manager will notify its transaction manager that it is ready to commit or that it needs to rollback.
- The resource manager will then wait for further instructions from the transaction manager.
- The local transaction manager will then notify the distributed transaction manager of the success or failure of their transaction.

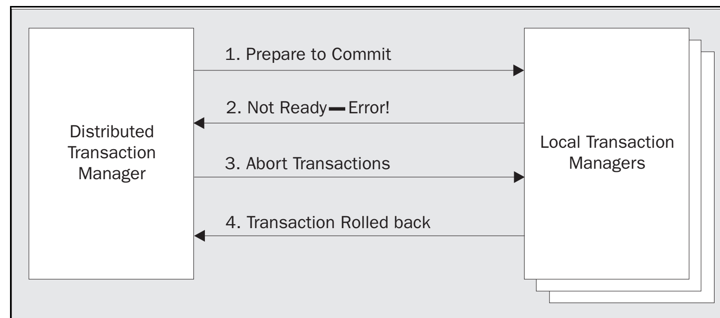
In the commit phase:

- The distributed transaction manager notifies the enlisted transaction managers to commit or rollback. This decision is based on the results from the various local transaction managers in the prepare phase.
- The local transaction managers notify their resource managers to commit or rollback their changes.
- The resource managers comply and report the outcome back to their local transaction manager.
- Finally, the local transaction manager reports the results to the distributed transaction manager, which likewise, returns the result to the calling application.

The following diagram summarizes this process:



Should a local transaction manager determine that it cannot commit its transaction, the distributed transaction manager will notify the other local transaction managers to abort – as illustrated below:



While it appears to the programmer that utilizing a distributed transaction is rather straightforward (and indeed it is), the actual implementation is extremely complex and difficult. So much so that some EJB vendors simply do not support distributed transactions. When we look at some of the problems inherent in implementing a distributed transactional system, it's easy enough to understand why:

- Failure recovery cannot be relied on, if it exists at all. A transaction can fail due to connection problems, server crashes, transaction timeouts, and deadlocks or other situations that might cause a transaction to get "stuck".
- What if our distributed transaction has three enlisted resource managers and the third resource manager fails during the commit phase? The other two resource managers have already committed their local transactions. Once data is committed, it usually cannot be rolled back. As a result, our data is now in an inconsistent, or corrupt, state. Worse still, other transactions may have executed against the now-committed data, compounding the problem and making recovery horrifically difficult if not outright impossible.
- If a distributed transaction does fail, how does our application determine where the failure occurred? How can we implement compensating transactions if we don't know where to apply them?
- While many transaction managers have logs that can be consulted to assist in recovery, this is often a manual process. In fact, you'd be surprised by how many organizations do not have a process in place to support transactional failures. The most common resolution is to restore from an earlier backup. This works provided, of course, that the organization is diligent with maintaining backups. The organization also has to be willing to accept the loss of work completed prior to the failure – most are not.

- The performance of distributed transactions is very slow compared to traditional, local transactions. Distributed transactions require a large amount of resources potentially spanning several servers. The two-phase commit protocol requires more communication than local transactions and this communication is conducted over a network, which can present all sorts of connectivity problems.
- Distributed transactions are typically longer-lived than local transactions, affecting overall system response time. This is due to the increased number of enlisted transaction managers and resource managers involved.
- Distributed transactions typically use the strictest form of isolation due to their higher-risk nature. This increases data contention and presents additional bottlenecks. Add to these performance issues the various schemes for ensuring stability, caching, logging, and recovery support, and the performance implications become substantial.
- By their very nature, the design of distributed transactional systems is much more complex. Special care must be taken to ensure that one transaction does not rely on the data affected, and thus locked, by another transaction.

The use of distributed transactional processing is more complex than the code to implement it would imply. The increased scalability and capability comes at the expense of performance and the added cost of pre-planning. There is greater risk for failure and more effort required for recovery tactics. Some systems do provide support for recovery methods. However, effective recovery relies not just on the automated capabilities of the application server, but on corporate policy, as well. Without a proper contingency plan and standardized, agreed-upon, rules for data reconciliation, data recovery success will be minimal at best.

Putting It All Together: An Example

Let's take a look at a full example of a stateful session bean that manages its own transactions. What follows is a bean used for interacting with a ledger – a part of our banking system. The `LedgerBean` will allow us to enter account activity for checking accounts and savings accounts. It will also allow us to send messages to a bank supervisor.

All of this will be conducted within a single transaction. Also, several account activities and messages can be created all within the same transaction. If at any point an error should occur, the entire transaction will be rolled back. For the sake of simplicity, the banking activity is a simple string.

This example will illustrate a number of different things:

- How to manage our own transactions
- How to maintain a transaction across method invocations
- How to include different databases within the same transaction
- How to include other resources, such as a messaging server, in a transaction

First, we need two databases, one called `CheckingDB` and another called `SavingsDB`. For the purposes of this example, we will assume that `CheckingDB` is an Oracle database and `SavingsDB` is in Cloudscape.

Next, within each database, we create the following table:

```
CREATE TABLE Ledger(  Activity VARCHAR(100)  );
```

The `LedgerBean` class contains five remote methods that are called by the client:

- `openLedger()`
  This method starts the transaction.
- `addCheckingAccountActivity()`
  Allows us to enter activity in to the checking account ledger. Checking accounts are maintained in the `CheckingDB` database.
- `AddSavingsAccountActivity()`
  Allows us to enter activity in to the savings account ledger. Savings accounts are maintained in the `SavingsDB` database.
- `notifySupervisor()`
  Sends a copy of the activity being reported to a message queue. At some other point (and in another application) a bank supervisor will read the message from the queue. The message queue accepts messages from client programs and submits them to a workflow queue for Supervisors, named `SupervisorQueue`. For more information on messaging and the Java Messaging Service (JMS), please refer to *Professional JMS Programming*, from Wrox Press, ISBN 1-861004-93-1, or the JMS specification document available at http://java.sun.com/jms/.
- `closeLedger()`
  This method completes the transaction.

The LedgerBean Class

At various points, the bean may throw an application exception. We can assume that these exceptions simply extend `java.lang.Exception` (see below):

```
package bank.ledger;

import javax.ejb.*;
import javax.transaction.*;
import javax.sql.*;
import java.sql.*;
import javax.jms.*;
import javax.naming.*;

public class LedgerBean implements SessionBean {

  SessionContext sessionCtx = null;
```

In order for us to be able to get a reference to the `UserTransaction` interface, we must access it through the `SessionContext` object. We can get a reference to the `SessionContext` object when the container calls the `setSessionContext()` method:

```
  public void setSessionContext(SessionContext ctx) {
    sessionCtx = ctx;
  }
  public void ejbCreate() {};
  public void ejbActivate() {};
  public void ejbPassivate() {};
  public void ejbRemove() {};
```

The openLedger() method prepares the ledger for use by starting a new transaction:

```
public void openLedger() throws LedgerNotAvailableException {

  try {
    // Get a UserTransaction instance
    UserTransaction userTrx = sessionCtx.getUserTransaction();

    // Start the transaction
    userTrx.begin();
  } catch(Exception ex) {
    System.err.println(ex);
    throw new LedgerNotAvailableException();
  }
}
```

The addCheckingAccountActivity() method will add an account activity record to the Ledger in the Checking account database. We look up a context for the CheckingDB database to give us a data source, establish a connection, insert the activity log, and close up the connection. On an exception being caught, we attempt to rollback the transaction, print the exception to the standard error log, and throw a LedgerFailureException; if the rollback fails, we log the error, and throw a LedgerNotAvailableExcpetion. We'll see these simple exceptions after we've defined the bean in full. This method can be called several times within the same transaction:

```
public void addCheckingAccountActivity(String activity)
                                      throws LedgerNotAvailableException,
                                             LedgerFailureException  {

  try {
    InitialContext initialCtx = new InitialContext();

    // Get a handle to the datasource and a connection for use
    DataSource ds =
           (DataSource)initialCtx.lookup("java:comp/env/jdbc/CheckingDB");
    java.sql.Connection conn = ds.getConnection();
    Statement stmt = conn.createStatement();
    stmt.execute("INSERT INTO Ledger (activity) VALUES('" + activity +
                                                         "')");
    conn.close();
  } catch(Exception ex) {
    try {
      UserTransaction ut = sessionCtx.getUserTransaction();
      ut.rollback();
    } catch(Exception ex2) {
      System.err.println(ex2);
      throw new LedgerNotAvailableException();
    }
    System.err.println(ex);
    throw new LedgerFailureException();
  }
}
```

The addSavingsAccountActivity() method adds an account activity record to the Ledger in the Savings account database; this method works just like the previous method, only it is targeted at the SavingsDB database. This method can also be called several times within the same transaction:

```
    public void addSavingsAccountActivity(String activity)
                                      throws LedgerNotAvailableException,
                                             LedgerFailureException {

      try {
        InitialContext initialCtx = new InitialContext();

        // Get a handle to the datasource and a connection for use
        DataSource ds =
        (DataSource)initialCtx.lookup("java:comp/env/jdbc/SavingsDB");
        java.sql.Connection conn = ds.getConnection();

        Statement stmt = conn.createStatement();
        stmt.execute("INSERT INTO Ledger (activity) VALUES('" + activity +
                                                           "')");
        conn.close();
      } catch(Exception ex) {
        try {
          UserTransaction ut = sessionCtx.getUserTransaction();
          ut.rollback();
        } catch(Exception ex2) {
          System.err.println(ex2);
          throw new LedgerNotAvailableException();
        }
        System.err.println(ex);
        throw new LedgerFailureException();
      }
    }
```

The `notifySupervisor()` method will send a copy of the activity to a banking supervisor. First we create a `QueueConnectionFactory`, which we use to create a `QueueConnection`. Using this connection, we then create a `QueueSession`, get a context for the message queue we want to send to, create a sender, create a `TextMessage`, set its text, and send it. On an exception being caught, we'll do much the same as with the previous methods – attempt to roll back the transaction, and print the exception to the standard error log. This is another method that can be called several times within the same transaction:

```
    public void notifySupervisor(String activity)
                                 throws LedgerNotAvailableException,
                                        NotificationFailureException {

      QueueConnectionFactory factory = null;
      QueueConnection connection = null;
      Queue queue = null;
      QueueSession session = null;
      QueueSender sender = null;
      TextMessage message = null;

      try {
        InitialContext initialCtx = new InitialContext();

        // Prepare the messaging objects
        factory = (QueueConnectionFactory)initialCtx.lookup
                                 ("java:comp/env/jms/QueueConnectionFactory");
        connection = factory.createQueueConnection();
        session = connection.createQueueSession(true,
                                                Session.AUTO_ACKNOWLEDGE);
        queue = (Queue)initialCtx.lookup("java:comp/env/jms/SupervisorQueue");
        sender = session.createSender(queue);
```

```
        message = session.createTextMessage();
        message.setText(activity);

        sender.send(message);
      } catch(Exception ex) {
        try {
          UserTransaction ut = sessionCtx.getUserTransaction();
          ut.rollback();
        } catch(Exception ex2) {
          System.err.println(ex2);
          throw new LedgerNotAvailableException();
        }
        System.err.println(ex);
        throw new NotificationFailureException();
      }
    }
```

The `closeLedger()` method will close the ledger and commit the transaction:

```
    public void closeLedger() throws LedgerNotAvailableException,
                                     LedgerFailureException {

      try {
        // Get a UserTransaction instance
        UserTransaction userTrx = sessionCtx.getUserTransaction();

        // Start the transaction
        userTrx.commit();
      } catch(Exception ex) {
        try {
          UserTransaction ut = sessionCtx.getUserTransaction();
          ut.rollback();
        } catch(Exception ex2){
          System.err.println(ex2);
          throw new LedgerNotAvailableException();
        }
        System.err.println(ex);
        throw new LedgerFailureException();
      }
    }

  }
```

The LedgerHome Interface

```
  package bank.ledger;

  import java.rmi.RemoteException;
  import javax.ejb.*;

  public interface LedgerHome extends EJBHome {
    Ledger create() throws RemoteException, CreateException;
  }
```

The Ledger Interface

```
package bank.ledger;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Ledger extends EJBObject {

  public void openLedger()
          throws LedgerNotAvailableException, RemoteException;

  public void addCheckingAccountActivity(String activity)
          throws LedgerFailureException, LedgerNotAvailableException,
                 RemoteException;

  public void addSavingsAccountActivity(String activity)
          throws LedgerFailureException, LedgerNotAvailableException,
                 RemoteException;

  public void notifySupervisor(String activity)
          throws NotificationFailureException, LedgerNotAvailableException,
                 RemoteException;

  public void closeLedger()
          throws LedgerFailureException, LedgerNotAvailableException,
                 RemoteException;
}
```

The LedgerClient Class

Here is a simple client that will invoke our session bean. The client program will start by obtaining a reference to the Ledger session bean. Once it has a valid remote reference, it will then deposit $100 in the checking account, deposit $200 in the savings account, and withdraw $500 from the checking account. Lastly, it will send a message to a supervisor indicating that the $500 withdrawal was made:

```
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import bank.ledger.LedgerHome;
import bank.ledger.Ledger;

public class LedgerClient {

  public static void main(String args[]) {

    try {
      InitialContext ctx = new InitialContext();
      Object obj = ctx.lookup("Ledger");
      LedgerHome ledgerHome = (LedgerHome) PortableRemoteObject.narrow(obj,
                                                    LedgerHome.class);
      Ledger ledger = ledgerHome.create();

      ledger.openLedger();
      ledger.addCheckingAccountActivity("$100 deposited to checking " +
                                        "account 12345");
      ledger.addSavingsAccountActivity("$200 deposited to checking " +
                                       "account 12345");
      ledger.addCheckingAccountActivity("$500 withdrawn from checking " +
                                        "account 12345");
```

```
        ledger.notifySupervisor("$500 withdrawn from checking account 12345");
        ledger.closeLedger();

      } catch(Exception e) {
        System.out.println(e.toString());
      }
    }
  }
```

The Exceptions

These exceptions simply extend `java.lang.Exception`:

```
package bank.ledger;

public class LedgerFailureException extends Exception {}
```

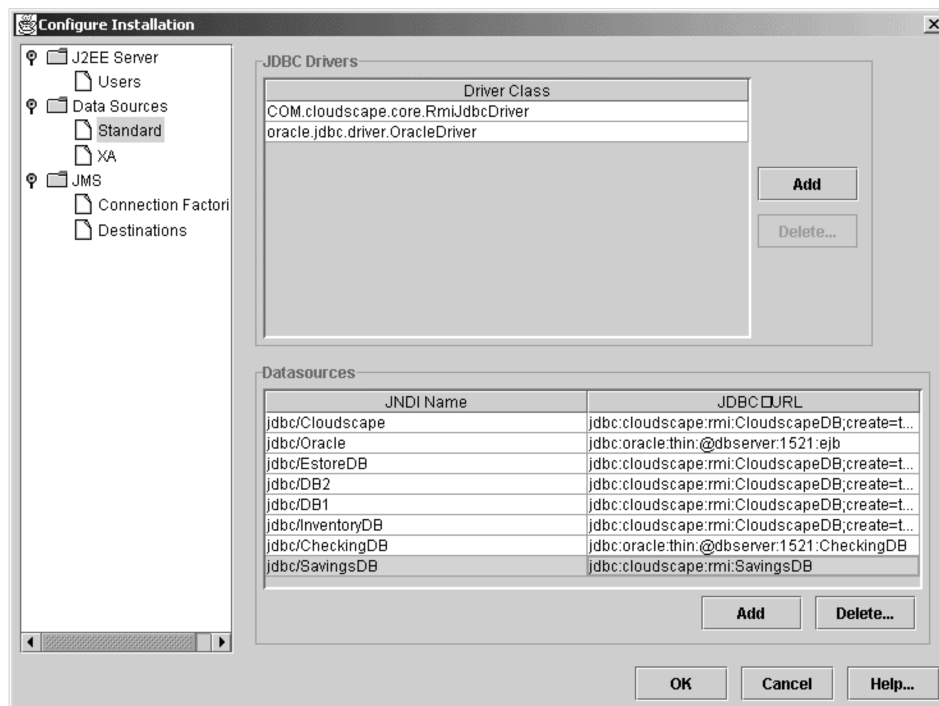
```
package bank.ledger;

public class LedgerNotAvailableException extends Exception {}
```

```
package bank.ledger;

public class NotificationFailureException extends Exception {}
```

Running the Example

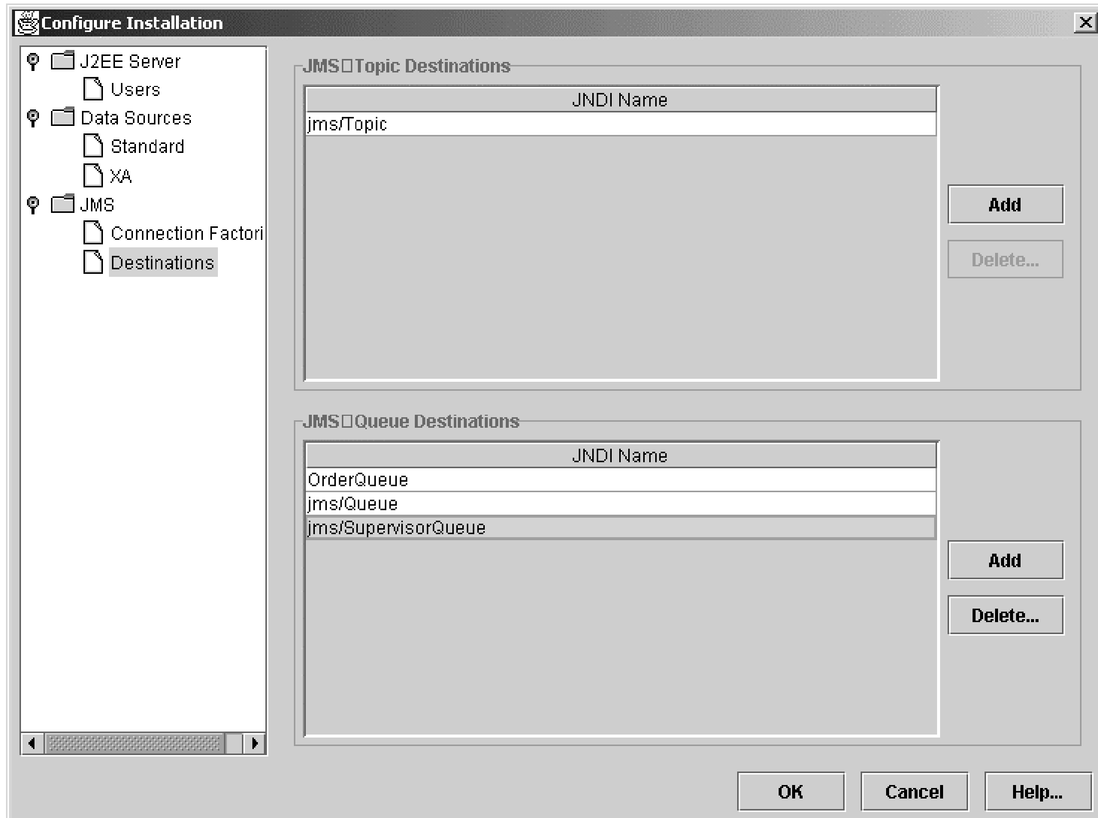
We will run this example on the J2EE 1.3 Reference Implementation. Before we actually run the code, we need to configure some data sources for our bean to use.

Click on the Tools | Server Configuration... | Data Sources | Standard:

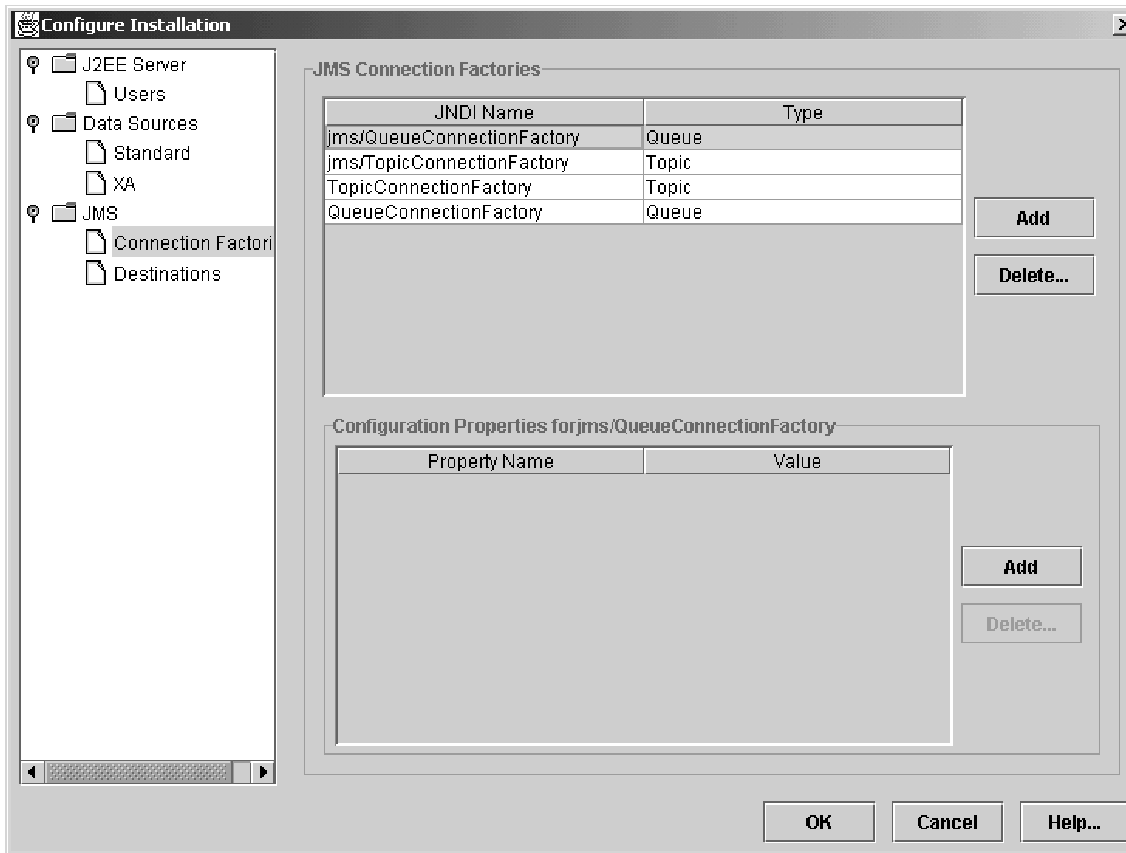


Enter the driver class for your database. In our case, we will be using Oracle, so we specify the `oracle.jdbc.driver.OracleDriver`. Next, we need to specify the JNDI names of our two database tables that are looked up in the `LedgerBean` class: `jdbc/CheckingDB` and `jdbc/SavingsDB`, and then provide them with a suitable URL for the database.

We also need to set up a destination JMS queue, which we can do on the same dialog. Select the Destinations page from the JMS folder on the left:



To create a JMS queue destination, all we need do is click the Add button, and then supply a name – in this case, `SupervisorQueue`. Notice that we won't worry about getting the message from the queue in this example. We'll also create a connection factory for our queue, on the tab above Destinations. Click the Add button next to the list of Queue Connection Factories to add our new entry, which we'll call `jms/QueueConnectionFactory`:



Now create a JAR file called `Ledger.jar` with the following structure (the `ejb-jar.xml` file for this example will be shown below):

```
bank/
     ledger/
            Ledger.class
            LedgerHome.class
            LedgerBean.class
            LedgerFailureException.class
            LedgerNotAvailableException.class
            NotificationFailureException.class
META-INF/
         ejb-jar.xml
```

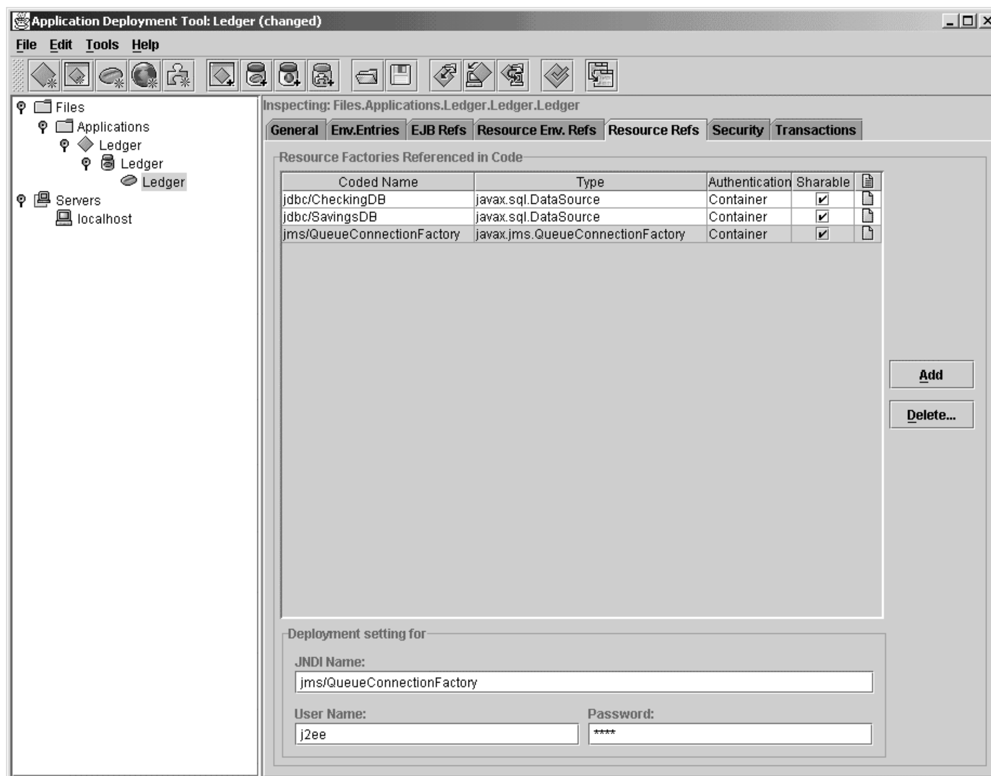
The Deployment Descriptor

```
<?xml version="1.0" encoding="Cp1252"?>

<!DOCTYPE ejb-jar PUBLIC
          '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN'
          'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
```

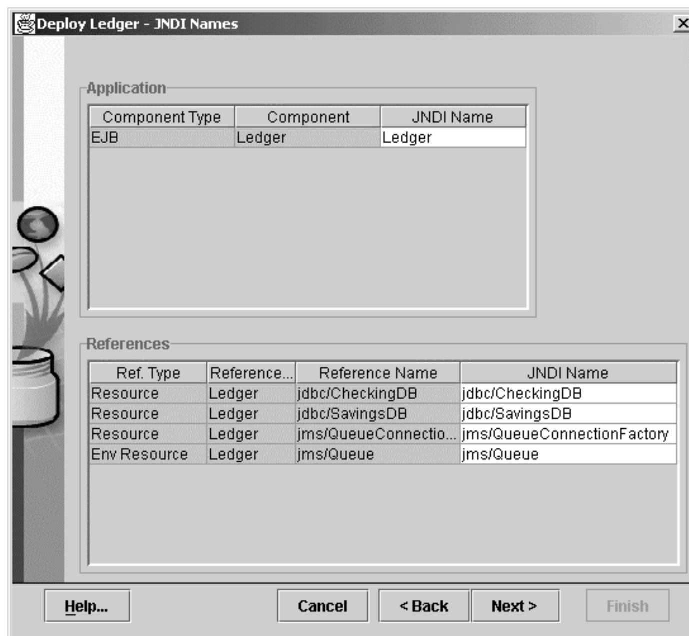
```
<ejb-jar>
  <display-name>Ledger</display-name>
  <enterprise-beans>
    <session>
      <display-name>Ledger</display-name>
      <ejb-name>Ledger</ejb-name>
      <home>bank.ledger.LedgerHome</home>
      <remote>bank.ledger.Ledger</remote>
      <ejb-class>bank.ledger.LedgerBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Bean</transaction-type>
      <resource-ref>
        <res-ref-name>jdbc/CheckingDB</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
        <res-sharing-scope>Shareable</res-sharing-scope>
      </resource-ref>
      <resource-ref>
        <res-ref-name>jdbc/SavingsDB</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
        <res-sharing-scope>Shareable</res-sharing-scope>
      </resource-ref>
      <resource-ref>
        <res-ref-name>jms/QueueConnectionFactory</res-ref-name>
        <res-type>javax.jms.QueueConnectionFactory</res-type>
        <res-auth>Container</res-auth>
        <res-sharing-scope>Shareable</res-sharing-scope>
      </resource-ref>
      <resource-env-ref>
        <resource-env-ref-name>jms/Queue</esource-env-ref-name>
        <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
      </resource-env-ref>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Deployment

Create a new application in the deployment tool called Ledger and import `Ledger.jar` by selecting File | Add to Application | EJB JAR... and specifying the correct path. Next, select the JNDI Names tab of the Ledger application. Give our EJB the JNDI name `Ledger`. Now, expand the `Ledger` bean and select the Resource Ref's tab. We need to provide some additional deployment information for the resource factories such as security credentials:



The final stage is deployment. Select Deploy... from the Tools menu, remembering to specify the directory the client will be run from as the place for the client JAR to be saved to. Here are the JNDI names we used:



Lastly, we have to run the example:

```
java -classpath %J2EE_HOME%\lib\j2ee.jar;LedgerClient.jar LedgerClient
```

Verify that actions have been inserted into the appropriate tables by using a tool such as SQL+ to read the row entries.

Transactions and Entity Beans

When using an entity bean, special care should be taken with container-managed persistence. The `ejbLoad()` and `ejbStore()` methods are called with the same transactional context as the business method that triggered them. Since `ejbLoad()` is invoked as the result of another of the bean's methods being called, this is quite understandable.

The `ejbStore()` method is always invoked when a bean's data is to be persisted. As the data is always persisted when a commit is executed, the `ejbStore()` method must execute within the same transactional context. For `ejbLoad()` and `ejbStore()`, the transactional attribute is, essentially, `Supports`.

This raises an interesting question. What happens if the originating or calling method does *not* have a transactional context? That is, what if there is no transaction in use when `ejbLoad()` or `ejbStore()` is called? Since their transactional attribute is `Supports`, they will not create a new transaction.

The EJB specification does not require that `ejbLoad()` and `ejbStore()` operate within a transaction. Instead, `ejbLoad()` will execute in the same transaction context as the business method which causes it to be invoked. The `ejbStore()` method will execute in the same transaction context that `ejbLoad()` or `ejbCreate()` was executed in – whichever was called prior to `ejbStore()`. In fact, the specification simply states when they should be invoked. The `ejbLoad()` method will be called at some point between when an entity bean is associated with a context ID and a business method is called. The `ejbStore()` method will be called at some point between when the business method is called and the object is disassociated from the context ID.

This poses a problem in that data could be persisted without a transaction. Worse than that, since the entity bean is, for all intents and purposes, a caching mechanism between the container and the data source, if the data is not maintained within a transaction, it will be very easy for that data's state to become invalid.

This is especially problematic when you consider that it is entirely possible for the following scenario to occur:

- A client updates a bean's data
- Another process updates the data in the data store
- The bean persists the data in the database

Changes made to the data by the external process are lost. Worse, if that process is operating within a transaction, the entity bean's persistence operation could fail. However, the client will already have been told that the update completed successfully. Furthermore, the entity bean has no way of notifying the client that the persistence operation failed.

Another scenario: the database server crashes or becomes unavailable while the bean is in the middle of updating. Without a transaction, information about the update will not be stored in a transaction log. Thus, the server will not be able to recover the update at a later point. Granted, most database servers have their own internal transaction managers, but that will be of little use to a bean that has multiple update processes in the same atomic operation.

The easiest way to avoid this scenario is to require your business methods to operate within a transactional context. Again, this is all provided that you are using container-managed persistence.

Why not create our own transaction in `ejbStore()`? In other words, why don't we just assign the `Required` transaction attribute to the method and not worry about it? The answer is simple enough: We cannot `Require` a transaction in `ejbStore()` because the `ejbStore()` method is always invoked by the container when the bean is passivated. If a bean is passivated while it is in the middle of a transaction, the act of creating a transaction in `ejbStore()` would result in the data being committed. If the bean's data is in an inconsistent state, this could cause a lot of problems.

If we are using bean-managed persistence, we should not rely on `ejbLoad()` and `ejbStore()` to manage our bean's state. This can be achieved by directly interacting with the database from within a given method and not relying on the container to notify your bean when to manage its state. Effectively, the `ejbLoad()` and `ejbStore()` methods are empty. We are, of course, gaining safety at the cost of performance, but this is always going to be a tradeoff. We must always be careful to choose wisely.

Transactional Context Propagation

When a client invokes a bean's method, the client's transactional context is propagated to the bean. However, before the bean's method is invoked, the client's transaction is suspended. If the bean has container-managed transaction demarcation, the container will handle the method's transaction in accordance with the transactional attribute. If the bean has bean-managed transactional demarcation, it will be up to the bean provider to determine if, and how, to incorporate the client's transactional context.

Transactional context propagation for bean-managed transaction demarcation can be summarized as follows:

- If the client calling the enterprise bean instance does not have a transactional context associated with it, and the bean instance does not have a transactional context associated with it, the container will invoke the bean method with an unspecified transaction context.
- If the calling client is already associated with a transaction, and the bean instance is not associated with a transaction, the container will suspend the client's transaction and invoke the bean method with an unspecified transaction context.
- If the calling client is not associated with a transaction, and the bean instance is already associated with a transaction (one is already in progress), the container will invoke the bean method with the transaction that is already associated with the bean instance. Note that this cannot happen with a stateless session bean, as they do not maintain transactional state across method invocations.
- If the client is already associated with a transaction, and the bean instance is also associated with a different transaction, the container will suspend the client's transaction and invoke the bean method with the transaction that is already associated with the bean instance. Once the bean method is completed, the container will resume the client's transaction.

The general rule here is that the bean's transaction context will take priority over the client's transaction context.

Transaction Interaction

For container-managed transactions, Enterprise JavaBeans provide a set of methods that allow some interaction with the current transactional context. These are:

- `setRollbackOnly()`
- `getRollbackOnly()`

The `setRollbackOnly()` method marks the current transaction for rollback. It does not actually roll the transaction back, as this would be in violation of the constraints imposed by container-managed transaction demarcation. Instead, it indicates that the transaction should be rolled back. The container, at a later point, will roll back the transaction. When the transaction is actually rolled back is at the discretion of the transaction. However, it is guaranteed that the transaction will not commit.

There are some restrictions to the use of the `setRollbackOnly()` method:

- First, it can only be called by container-managed transaction beans.
- Second, it can only be called by methods whose transactional attributes are `Required`, `RequiresNew`, or `Mandatory`. If `setRollbackOnly()` is invoked from a method that does not have one of these transactional attributes, the `java.lang.IllegalStateException` will be thrown.

The `getRollbackOnly()` method will return the current state of the transaction, embodied by the `javax.transaction.Status` interface. This provides us with the opportunity to determine, at an arbitrary point, whether the transaction should be rolled back due to some error condition or if it should be allowed to continue. By having the ability to preempt the transaction, we can prevent the invocation of resource calls that would otherwise have to be rolled back at the end of the transaction.

Transaction Events

There are three basic events associated with transaction demarcation:

- The transaction has started
- The transaction is about to commit
- The transaction has ended

When using container-managed transaction demarcation with session beans, EJB provides an optional callback mechanism by which to be notified of these transaction events, namely the `SessionSynchronization` interface.

The `SessionSynchronization` interface is defined as follows:

```
public interface javax.ejb.SessionSynchronization {

  public void afterBegin() throws EJBException, RemoteException;

  public void afterCompletion(boolean committed) throws EJBException,
                                                        RemoteException;

  public void beforeCompletion() throws EJBException, RemoteException;
}
```

The `afterBegin()` method is invoked by the container when a new transaction has started. This method will be called prior to any of the business methods that will be invoked within the context of the transaction. This gives the session bean an opportunity to prepare data for use, such as reading it in from a database or some other source, or formatting it.

The `beforeCompletion()` method is called by the container just prior to when a transaction is to be committed. This provides the session bean with an opportunity to conduct any last minute data preparation, such as validation or formatting. This also gives the session bean one last chance to rollback the transaction via the `setRollbackOnly()` method, or by throwing an exception.

The `afterCompletion()` method is called by the container once the transaction has committed. A `Boolean` value is passed to the method indicating whether the transaction successfully committed, `true`, or was rolled back, `false`. When this method is invoked, the transaction has completed. Therefore there is no transaction context available.

Transaction Isolation

The ACID property **isolation** requires that a transaction must be able to operate without regard for and without being affected by other active transactions in the system. In a concurrent system, several transactions may be executing at once – often on the same data. It may be necessary to protect one transaction from the efforts of another. Transaction isolation is achieved via locking and the serialization of data requests.

Locking controls access to a given set of data. The two primary types of locks are read locks and write locks. Read locks are **non-exclusive** locks – they will allow multiple transactions to read data simultaneously. Write locks are **exclusive** locks – they will only allow a single transaction to update a set of data.

Serialization guarantees that concurrently executing transactions will behave as if they were executing sequentially, not concurrently. Of course, the transactions will be executing concurrently, but they will appear to be executing in series. The result of serialization is the appearance that multiple transactions are working with data one at a time, in order.

Isolation levels specify concurrency control at a high-level. The types of locks and serialization used, as well as the extent to which they are applied, determines the level of isolation that a transaction will execute under. The actual implementation is up to the resource manager and/or transaction manager. Isolation levels vary from very relaxed to very strict. As might be expected, the stricter the level of concurrency control, the greater the impact on performance. As with so many other issues in designing concurrent systems, special care must be taken when determining which isolation level to use.

J2EE provides support for four types of isolation levels, as defined in the `java.sql.Connection` interface (we will see how to set these later):

- `TRANSACTION_READ_UNCOMMITTED`
- `TRANSACTION_READ_COMMITTED`
- `TRANSACTION_REPEATABLE_READ`
- `TRANSACTION_SERIALIZABLE`

In transactional processing, there are three major types of concurrency issues that the isolation levels attempt to address:

- Dirty reads
- Unrepeatable reads
- Phantom reads

The following discusses the various problems as well as the appropriate isolation level to use to resolve these problems.

Dirty Reads

A dirty read occurs when a transaction reads data that has been written by another transaction but has not been yet been committed. This happens when there is a complete lack of synchronization on the data. We return to our banking system for an example of a dirty read:

- A client's bank account has $1000.
- Transaction 1 deposits $500 into the account, but does not yet commit the operation.
- Transaction 2 is posting a check for $1500, reads the account and sees the balance is $1500. It then processes the check against the account and commits.
- Transaction 1, which is still active, rolls back its operation. The balance is restored to the $1000 that it was at before transaction one started.
- The $1500 check has still been cleared and the bank just lost the money!

A dirty read can occur if we use the lowest level of transactional isolation, `TRANSACTION_READ_UNCOMMITTED`. The only time that this level of isolation should be used in a system is when the transaction will be the only one accessing the data. Similarly, we should also use it when the data is, and always will be, read-only.

An example of this would be a static lookup table. Even in this case, we might want to implement some sort of read/write control mechanism in the rare event that the data might need to be updated. `TRANSACTION_READ_UNCOMMITTED` is used by the reading transactions, but the reads are blocked when a writing transaction is active.

The use of `TRANSACTION_READ_COMMITTED` avoids the dirty read problem. It requires that a transaction can only read data that has been committed. It cannot read data that is in the scope of another active transaction. This is the most common level of isolation and, in fact, is the default isolation method for most databases servers. However, since we will be using Enterprise JavaBeans which are concurrent by nature, we try to avoid using `TRANSACTION_READ_UNCOMMITTED` whenever possible.

Unrepeatable Reads

An unrepeatable read occurs when a transaction reads data from a database, but gets a different result if it tries to read the same data again within the same transaction. This typically happens when another transaction writes over some of the data that was read in by the first transaction.

Consider an order entry system where a customer's invoice is being reviewed:

- Clerk 1 is reading an invoice.
- Clerk 2 makes changes to the invoice's line items updating the unit price.
- Clerk 1 fulfills the invoice at the original price.
- The company has charged the client the wrong amount for the order!

The second clerk should not be allowed to modify the order while the first clerk is working with it.

If this behavior needs to be prevented, the `TRANSACTION_REPEATABLE_READ` isolation level should be used. `TRANSACTION_REPEATABLE_READ` provides more reliable transactions where data must be read and subsequently re-read. This isolation level works by locking the data so that other transactions cannot make changes to it. Once the transaction is completed, the other transactions will then be permitted to continue.

Phantom Reads

A phantom read occurs when a transaction executes multiple reads against a set of data and, in between two of the read operations, another transaction slips in and inserts additional data. This differs from the unrepeatable read problem in that here data is being inserted into our data set, rather than that data merely being updated.

Returning again to our example of an order-entry system, let's say we were fulfilling an order:

- The shipping department reviews the order, packages the items, and sends them on their way.
- Meanwhile, here comes Clerk 2 again who adds an additional item to the order.
- If the shipping department were to review the order again, they would see that another item has magically appeared. The client does not get their correct order and is very upset!

It is desirable that Clerk 2 should not be allowed to add items to this invoice while another person is working on it.

Using the transaction isolation level `TRANSACTION_SERIALIZABLE` prevents this from occurring. This level of isolation provides the strictest form of transaction management available.

Specifying Isolation Level

In container-managed transaction demarcation, the isolation level is determined by the container. Since EJB 1.1, there is no support for specifying the isolation level in the deployment descriptor. While some containers provide the ability to specify the isolation level at the resource-manager level, you should not rely on this. It is generally best to leave this decision to the container.

In bean-managed transaction demarcation, the isolation level is specified by directly interacting with the resource manager's API. In the case where a database is being used, this would be via the `java.sql.Connection.setTranactionIsolation()` method.

Be forewarned that support for transaction isolation by a resource manager is not a requirement. Therefore, caution should be used when specifying a transaction's isolation level. In fact, many resource managers will prevent you from doing this (database managers especially).

When determining the proper isolation level to use, it is important to remember that stricter levels of control will have a greater impact on performance. It is for this reason that each transaction will have to be considered on an individual basis. One isolation level will not be applicable across all transactions.

An isolation level is associated with a resource manager. As such, if a bean interacts with multiple resource managers (say, two database servers), a separate isolation level can be specified for each resource manager. However, most resource managers require that all access to that resource manager within a given transaction must be executed at the same isolation level.

It is very important that you do not attempt to switch isolation levels mid-way through a transaction, as this could cause the transaction to commit prematurely, among other erratic behavior. This is because you would be changing the rules governing locking and serialization in the middle of the transaction. This gets even more complex when you have multiple beans accessing the same resource manager within the same transaction. Certainly, a great deal of caution, coordination, and planning is required.

Alternatives to Isolation Control

Strict transaction isolation comes at a price – the greater the level of control, the greater the performance cost. When strict isolation is required, the negative performance impact could require us to seek alternative control methods. Two such methods are **optimistic locking** and **pessimistic locking**.

Optimistic locking allows many clients to access the same data concurrently. Optimistic locking is called so because we are "optimistic" that the data will not change while we are using it. When one of the clients needs to update the data, it submits the changes to a controller (say, an entity bean) for consideration. The controller compares the data in the database (or similar) to the data that was originally provided to the client. If they match, that is if no changes have occurred between the time that the client read the data and when the client requested to change the data, then the update is allowed to proceed. If, on the other hand, the data has changed since it was initially provided to the client, the client is notified that it must refresh its copy of the data and re-submit the update request. As this can result in a duplication of effort, optimistic locking is best suited for situations where most of the clients will be reading, not updating.

Pessimistic locking takes the view that there is a high probability that someone will want to change data while we are working with it, thus the data must be protected. Pessimistic locking is usually achieved through the use of transactions. However, where transactions are not available or could be very long-lived, a semaphore can be used in its place. In this case, the master data record contains a flag that indicates whether the data is currently in use by a client for either viewing or editing.

Pessimistic locking is achieved by utilizing read locks every time data is read, and write locks every time the data is to be updated. These locks are held for the duration of the transaction in which the data is being used. Clients may obtain a read lock on the data provided that no other transactions have write locks. Write locks are used to update the data and other clients are still allowed to read the data provided that they do not require a read lock. However, a write lock will not allow another transaction to read its changing data until those changes have been committed.

Behavior of Exceptions in Container-Managed Transactions

The behavior of exceptions when using container-managed transactions depends on the type of transaction being used, who initiated the transaction, and the type of exception occurring. Let's briefly review how exceptions work in EJB.

Recall that there are two categories of exceptions in EJB:

- Application exceptions
- System exceptions

Application exceptions are basically any exceptions that are declared in the `throws` clause of the methods of a bean's home or remote interfaces. Put another way, application exceptions are those exceptions that will be propagated to a bean's client.

The purpose of an application exception is to indicate an abnormal application-level condition. For example, an invalid account balance, an invalid account number, or attempting to access prohibited data could all result in an application exception.

Typically, an application exception is defined by the bean developer and represents a business-logic exception more than a system exception. `InsufficientFundsException`, `AccountNotFoundException`, and `AccessDeniedException` are all examples of application exceptions. In addition, `javax.ejb.CreateException`, `RemoveException`, and `FinderException` are also considered application exceptions, despite the fact that they do not represent business logic exceptions. However, it is desirable that these exceptions be propagated back to the client.

System exceptions are any non-application exceptions. When a system exception occurs, it is due to a system-level error, not a business-level error. System exceptions are not returned to the client. Instead, they are caught at the server level and dealt with there. If an exception is to be returned to the client, either an `EJBException` will be thrown or an application exception will be created and thrown instead.

There are no hard and fast rules regarding the behavior of system exceptions, but the following guidelines usually apply:

- If a `RuntimeException` or `Error` occurs, it will be propagated to the container
- If the exception is not a `RuntimeException`, it will be wrapped in an `EJBException` and returned to the bean
- Any other type of unexpected error will result in an `EJBException`

Exception Scenarios

When an exception occurs within the bounds of a transaction, the results vary depending on the type of error and where the transaction originated. The following sections detail the different possible exception scenarios.

Scenario One

The transaction originated on the client and the transactional attribute for the bean's method is either `Required`, `Mandatory`, or `Supports`.

When an application exception occurs within this context, the container will still attempt to commit the transaction. Since an application exception does not necessarily indicate an error with the data operation itself, the container cannot assume on its own that the transaction should be rolled back. Furthermore, as the container did not start the transaction itself, it cannot safely guess the use of the transaction.

If we raise an application exception and know that the transaction should be rolled back, we should invoke the `setRollbackOnly()` method on the `EJBContext` object. This should be done within our bean's method where the application exception would be raised.

It is at the discretion of the client whether or not to continue a transaction when an application exception occurs. A good example of this would be an `InsufficientFundsException` being thrown when posting a check within our banking system. The client may wish to try posting the check to another account, such as a savings account or overdraft account. When a client does receive an application exception, however, they should check the `getRollbackOnly()` or `getStatus()` methods.

When a system exception or error occurs within this context, the container will:

- Log the exception or error for later review by the system administrator
- Mark the transaction for rollback, via the `setRollbackOnly()` method
- Discard the instance of the bean
- Throw a `TransactionRolledBackException` or `TransactionRolledBackLocalException` back to the client

Scenario Two

The transaction is started by the container, with the transactional attribute `Required` or `RequiresNew`.

When an application exception is thrown, if the bean's method where the exception was thrown marked the transaction for roll back via the `setRollbackOnly()` method, the container will rollback the transaction. Otherwise the container will attempt to commit the transaction. The container will then re-throw the application exception back to the client.

When a system exception occurs, the container will:

- Log the exception or error for later review by the system administrator
- Mark the transaction for rollback, via the `setRollbackOnly()` method
- Discard the instance of the bean
- Throw a `RemoteException` or `EJBException` back to the client

This is similar to the behavior of a transaction started by the client, except that a `RemoteException` or `EJBException`, depending on client location, is thrown. Since the transactional context is limited to the bean's method, it would not make sense to return a `TransactionRolledBackException` or `TransactionRolledBackLocalException`, as above.

Scenario Three

The bean method is invoked with an unspecified transaction context, such as when the transaction attributes `NotSupported` or `Never` are used, or when the bean's method is marked as `Supports` but is invoked by an unspecified transaction context.

The container will not rollback the transaction if an application error is thrown, simply re-throwing the application exception that occurred. Remember that a transaction cannot be marked for rollback within an unspecified transaction context. The `setRollbackOnly()` method is only supported for methods with the transaction attribute `Required`, `RequiresNew`, or `Manadatory`. Attempting to set a transaction for rollback within an unspecified transaction context will result in a `java.lang.IllegalStateException`.

When a system error occurs, the container will log the exception or error, discard the instance of the bean, and throw a `RemoteException` or `EJBException`.

Scenario Four

The method is part of a session bean or message-driven bean and the transaction is bean-managed.

When an application exception is thrown, the client will receive the application exception.

When a system error occurs, the container will:

- Log the exception or error for later review by the system administrator
- Mark the transaction for rollback, via the `setRollbackOnly()` method
- Discard the instance of the bean
- Throw a `RemoteException` back to the client

When dealing with container-managed transactions, if a transaction was rolled back due to the `setRollbackOnly` flag being set, the container will roll the transaction back, but will not throw a `RemoteException`. Instead, the container will either return the normal result of the method or, if one was thrown, an application exception.

In container-managed transactions, if an exception occurs, the container will release any connections to any managed resources. A managed resource is any resource (such as a database) that the bean's instance might have obtained via the container's resource factory. However, the container cannot release any *unmanaged* resources (such as a network socket) that the bean instance might have obtained through the standard JDK APIs. Unmanaged connections will be released during regular garbage collection. If you are using unmanaged connections, you should be careful to release them in the event of an exception.

When a system exception occurs, the transaction may not necessarily be marked for rollback. For example, a communication problem prevents the remote bean from even begin called. In this case, the server won't have the opportunity to mark the transaction for rollback because it never even got the transaction in the first place.

Summary

Transactional processing is critical in any application where data is being accessed by more than one source at a time. The field of transactional processing is quite complex and we have really only scratched the surface here. Through the course of this chapter, we have seen:

- While the Enterprise JavaBeans framework strives to free us of a lot of the concerns involved with developing enterprise-class applications, we are still required to understand how components interact with each other and the data that they share.
- Different levels of isolation and concurrency control are available to our components. Each component will require individual consideration to determine the best balance of control versus balance. The stricter the control, the greater the performance cost.
- We can rely on the EJB container to manage our transaction for us, or we can manage them ourselves. If we choose to let the container manage our transactions, we must indicate, in the deployment descriptor, how those transactions will behave. Also, container-managed transactions are conducted at the method level. If we desire a finer level of control, we will have to manage the transactions ourselves.
- Finally, EJB provides developers with a considerable amount of control over the transactional processing of their components. Although entity beans can only be controlled by the container, we can still largely effect the transactions on those beans by specifying their transaction attributes.

Understanding how transactional processing works is crucial to the success of any enterprise-level system and we must ensure that we have a solid understanding of the concepts. For a more thorough exploration of transactional processing, *Transaction Processing: Concepts and Techniques* (ISBN 1-55860-190-2) by Jim Gray and Andreas Reuter is considered to be the definitive source.

In the next chapter, we will look at some of the key issues in security for EJBs.