Architecture of Enterprise Applications 4
Transaction Management

Haopeng Chen

RE*liable,* ***IN****telligent and* ***S****calable Systems Group (****REINS****)*
Shanghai Jiao Tong University
Shanghai, China
http://reins.se.sjtu.edu.cn/~chenhp
e-mail: chen-hp@sjtu.edu.cn

Contents and Objectives

- Contents
  - What Is a Transaction?
  - Container-Managed Transactions
  - Isolation and Database Locking
  - Updating Multiple Databases

- Objectives
  - 能够根据数据访问的具体场景设计并实现事务管理方案，包括事务边界划分和事务隔离级别设置

Transactions in Java EE Applications

- In a Java EE application, a transaction
  - is a series of actions that must all complete successfully, or else all the changes in each action are backed out. Transactions end in either a commit or a rollback.

- The Java Transaction API (JTA) allows applications
  - to access transactions in a manner that is independent of specific implementations

- The JTA defines the `UserTransaction` interface that applications use to start, commit, or roll back transactions.
  - Application components get a `UserTransaction` object through a JNDI lookup by using the name `java:comp/UserTransaction` or by requesting injection of a `UserTransaction` object.

What Is a Transaction?

- Pseudocode:

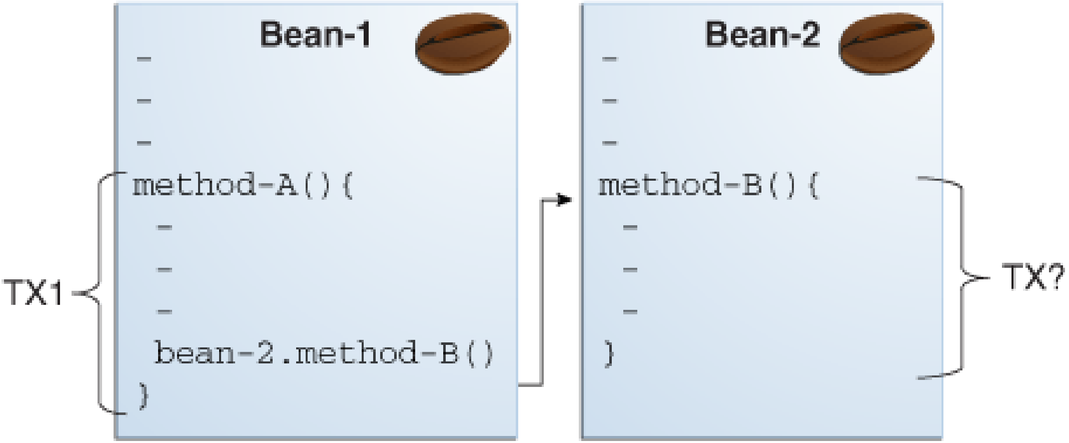
```
begin transaction
 debit checking account
 credit savings account
 update history log
commit transaction
```

- A **transaction** is often defined as an indivisible unit of work
  - Either all or none of the three steps must complete.

- A transaction can end in two ways:
  - with a commit or with a rollback.

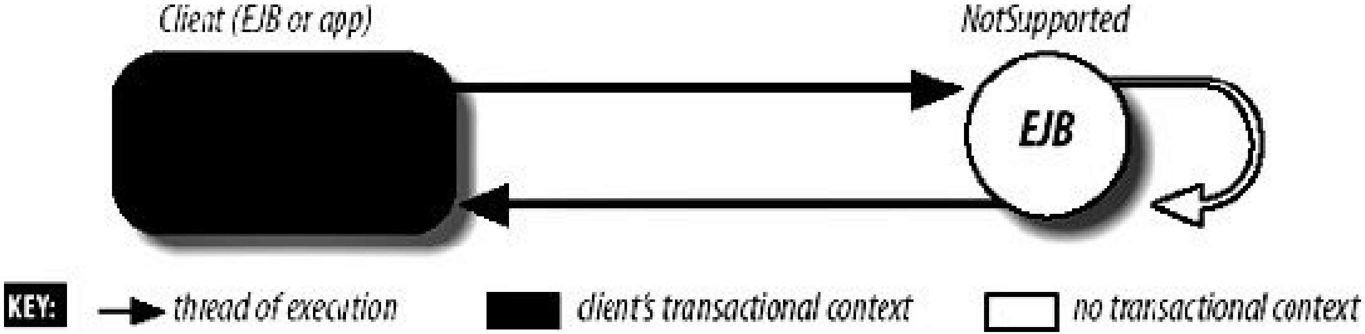
Container-Managed Transactions

- In an enterprise bean with container-managed transaction demarcation,
  - the EJB container sets the boundaries of the transactions.



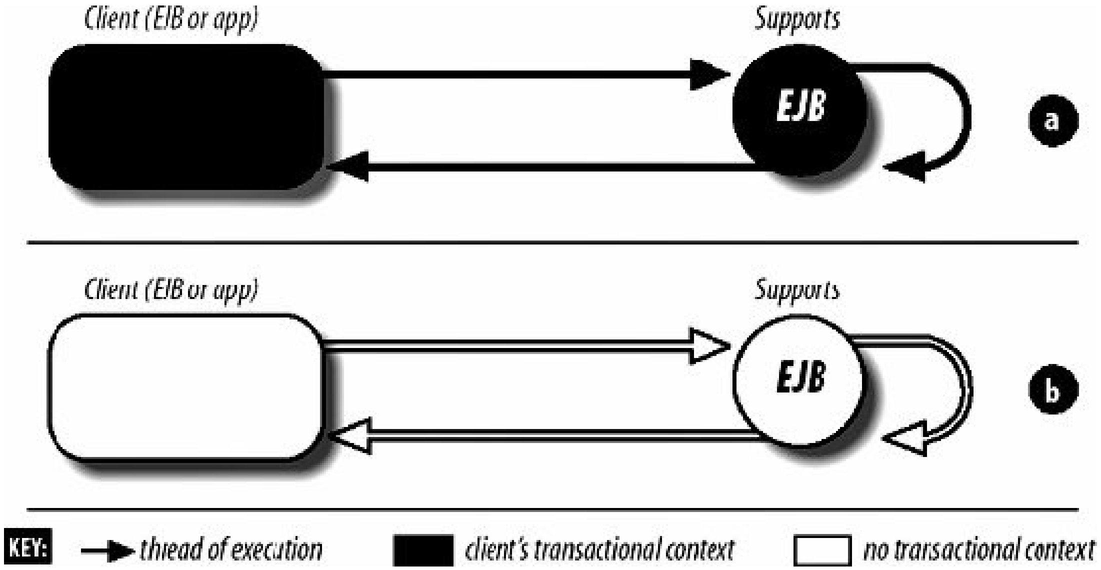
Transaction attributes defined

- NotSupported
  - Invoking a method on an app with this transaction attribute suspends the transaction until the method is completed.



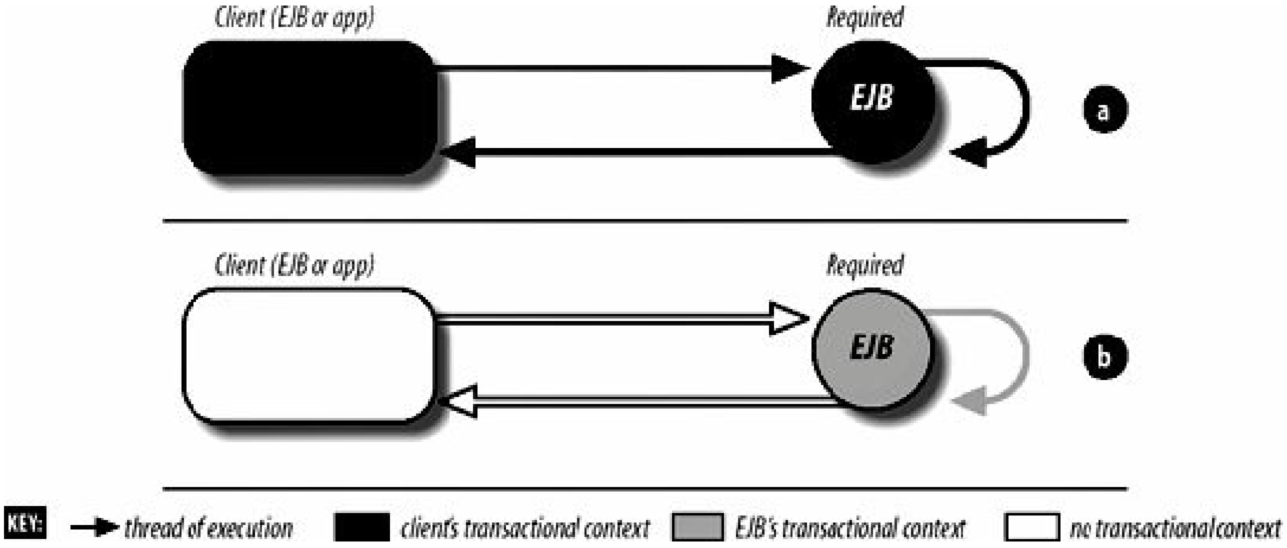
Transaction attributes defined

- Supports
  - This attribute means that the app method will be included in the transaction scope if it is invoked within a transaction.



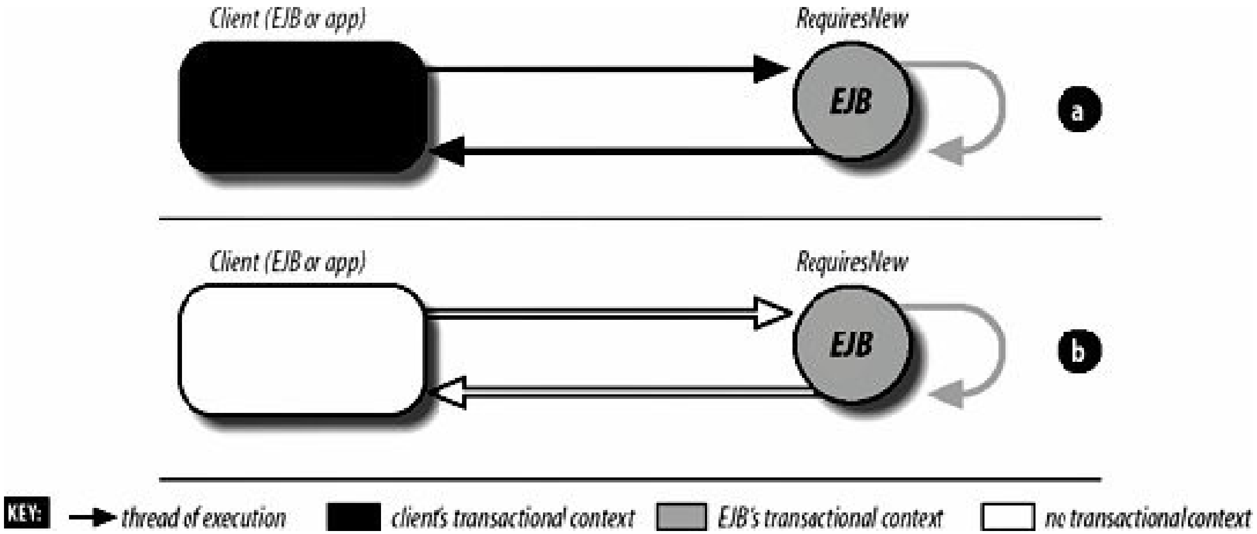
Transaction attributes defined

- Required
  - This attribute means that the app method must be invoked within the scope of a transaction.



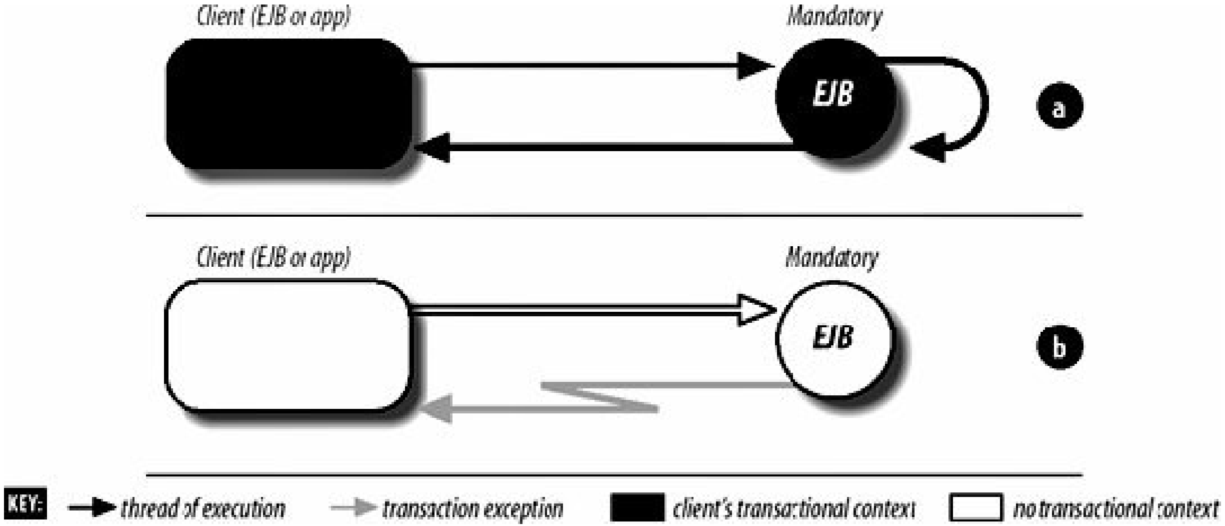
Transaction attributes defined

- RequiresNew
  - This attribute means that a new transaction is always started.



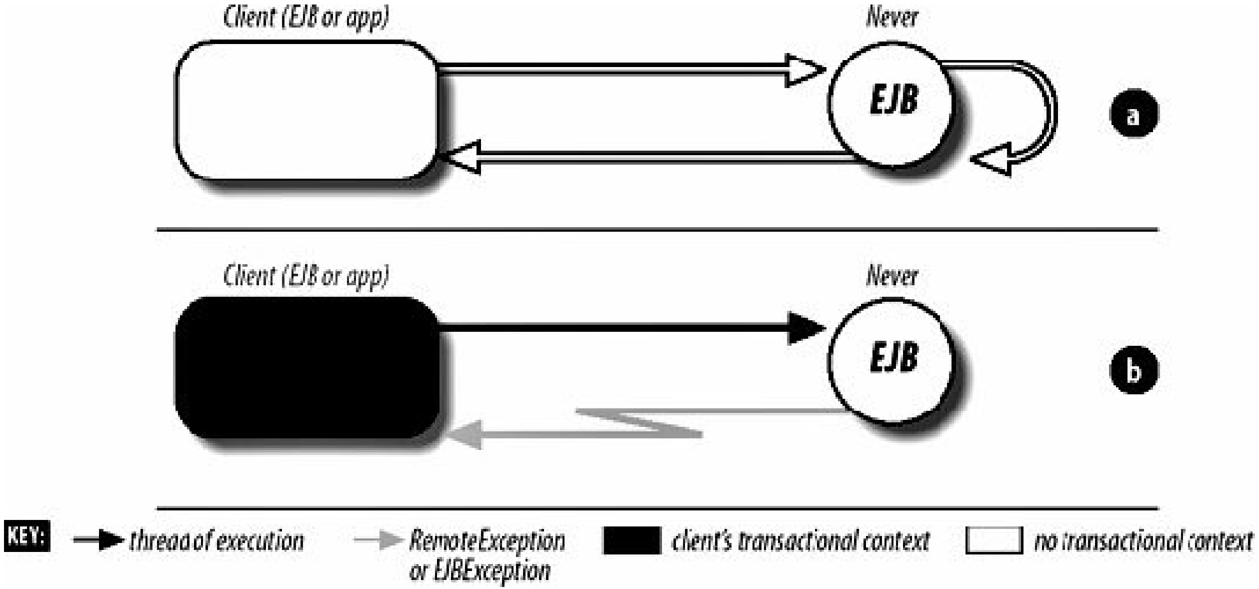
Transaction attributes defined

- Mandatory
  - This attribute means that the app method must always be made part of the transaction scope of the calling client.



Transaction attributes defined

- Never
  - This attribute means that the app method must not be invoked within the scope of a transaction.



Transaction Attributes and Scope

Transaction Attribute	Client's Transaction	Business Method's Transaction
Required	None	T2
Required	T1	T1
RequiresNew	None	T2
RequiresNew	T1	T2
Mandatory	None	Error
Mandatory	T1	T1
NotSupported	None	None
NotSupported	T1	None
Supports	None	None
Supports	T1	T1
Never	None	None
Never	T1	Error

Setting Transaction Attributes

- Transaction attributes are specified by
  - decorating the enterprise bean class or method with a `javax.ejb.TransactionAttribute` annotation
  - and setting it to one of the `javax.ejb.TransactionAttributeType` constants.

Transaction Attribute	TransactionAttributeType Constant
Required	TransactionAttributeType.REQUIRED
RequiresNew	TransactionAttributeType.REQUIRES_NEW
Mandatory	TransactionAttributeType.MANDATORY
NotSupported	TransactionAttributeType.NOT_SUPPORTED
Supports	TransactionAttributeType.SUPPORTS
Never	TransactionAttributeType.NEVER

Setting Transaction Attributes

- The following code snippet demonstrates how to use the `@TransactionAttribute` annotation:

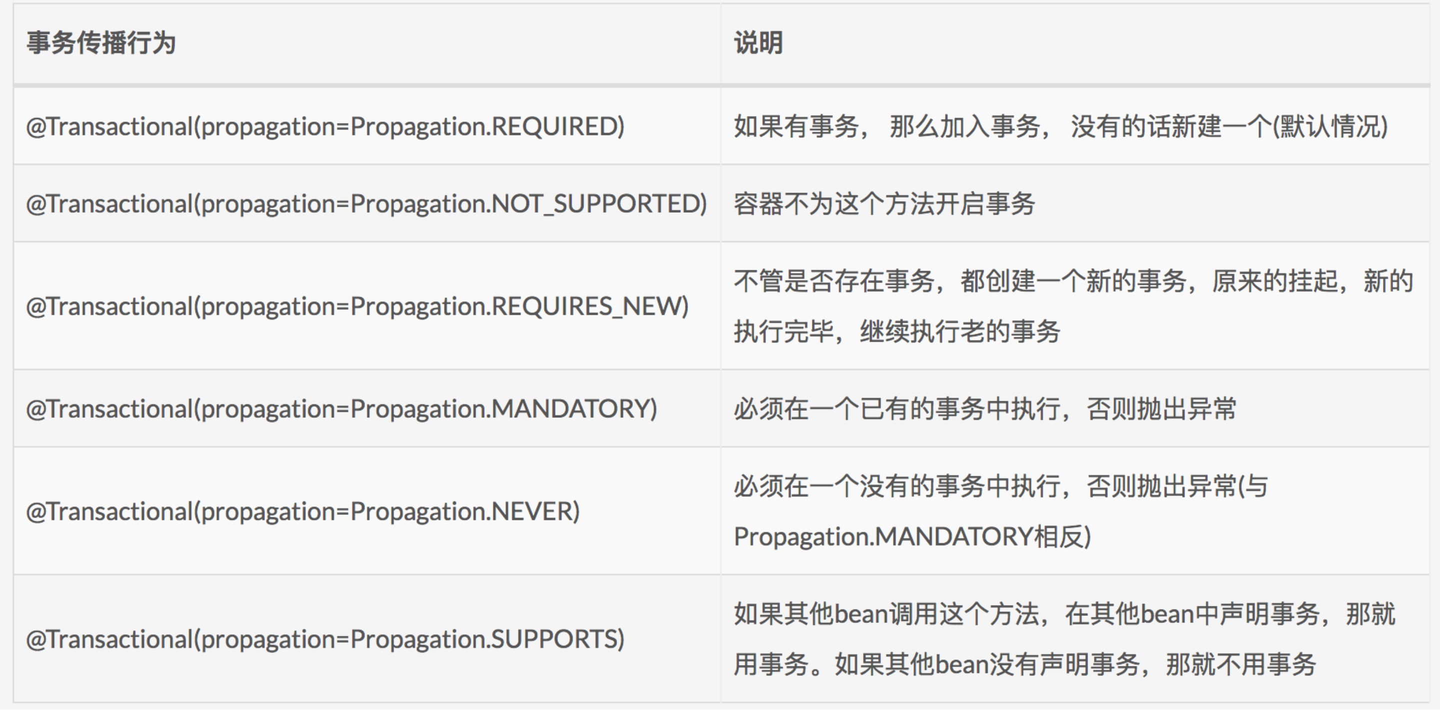
```
@TransactionAttribute(NOT_SUPPORTED)
@Stateful
public class TransactionBean implements Transaction {
 ...
 @TransactionAttribute(REQUIRES_NEW)
 public void firstMethod() {...}

 @TransactionAttribute(REQUIRED)
 public void secondMethod() {...}
 public void thirdMethod() {...}
 public void fourthMethod() {...}
}
```

Rolling Back a CMT

- There are two ways to roll back a container-managed transaction.
  - First, if a system exception is thrown, the container will automatically roll back the transaction.
  - Second, by invoking the `setRollbackOnly` method of the `EJBContext` interface, the bean method instructs the container to roll back the transaction.
  - If the bean throws an application exception, the rollback is not automatic but can be initiated by a call to `setRollbackOnly`.

Managing Transactions in Spring

- @Transactional propagation



事务传播行为	说明
@Transactional(propagation=Propagation.REQUIRED)	如果有事务， 那么加入事务， 没有的话新建一个(默认情况)
@Transactional(propagation=Propagation.NOT_SUPPORTED)	容器不为这个方法开启事务
@Transactional(propagation=Propagation.REQUIRES_NEW)	不管是否存在事务，都创建一个新的事务，原来的挂起，新的执行完毕，继续执行老的事务
@Transactional(propagation=Propagation.MANDATORY)	必须在一个已有的事务中执行，否则抛出异常
@Transactional(propagation=Propagation.NEVER)	必须在一个没有的事务中执行，否则抛出异常(与Propagation.MANDATORY相反)
@Transactional(propagation=Propagation.SUPPORTS)	如果其他bean调用这个方法，在其他bean中声明事务，那就用事务。如果其他bean没有声明事务，那就不用事务

Managing Transactions in Spring

BookingService.java

```
@Component
public class BookingService {

  private final static Logger logger = LoggerFactory.getLogger(BookingService.class);

  private final JdbcTemplate jdbcTemplate;

  public BookingService(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }

  @Transactional
  public void book(String... persons) {
    for (String person : persons) {
      logger.info("Booking " + person + " in a seat...");
      jdbcTemplate.update("insert into BOOKINGS(FIRST_NAME) values (?)", person);
    }
  }

  public List<String> findAllBookings() {
    return jdbcTemplate.query("select FIRST_NAME from BOOKINGS",
        (rs, rowNum) -> rs.getString("FIRST_NAME"));
  }

}
```

Managing Transactions in Spring

AppRunner.java

```
@Component
class AppRunner implements CommandLineRunner {

  private final static Logger logger = LoggerFactory.getLogger(AppRunner.class);

  private final BookingService bookingService;

  public AppRunner(BookingService bookingService) {
    this.bookingService = bookingService;
  }

  @Override
  public void run(String... args) throws Exception {
    bookingService.book("Alice", "Bob", "Carol");
    Assert.isTrue(bookingService.findAllBookings().size() == 3,
        "First booking should work with no problem");
    logger.info("Alice, Bob and Carol have been booked");
    try {
      bookingService.book("Chris", "Samuel");
    } catch (RuntimeException e) {
      logger.info("v--- The following exception is expect because 'Samuel' is too " +
          "big for the DB ---v");
      logger.error(e.getMessage());
    }
```

Managing Transactions in Spring

AppRunner.java

```java
    for (String person : bookingService.findAllBookings()) {
      logger.info("So far, " + person + " is booked.");
    }
    logger.info("You shouldn't see Chris or Samuel. Samuel violated DB constraints, " +
        "and Chris was rolled back in the same TX");
    Assert.isTrue(bookingService.findAllBookings().size() == 3,
        "'Samuel' should have triggered a rollback");

    try {
      bookingService.book("Buddy", null);
    } catch (RuntimeException e) {
      logger.info("v--- The following exception is expect because null is not " +
          "valid for the DB ---v");
      logger.error(e.getMessage());
    }

    for (String person : bookingService.findAllBookings()) {
      logger.info("So far, " + person + " is booked.");
    }
    logger.info("You shouldn't see Buddy or null. null violated DB constraints, and " +
        "Buddy was rolled back in the same TX");
    Assert.isTrue(bookingService.findAllBookings().size() == 3,
        "'null' should have triggered a rollback");
  }
}
```

Managing Transactions in Spring

ManagingTransactionsApplication.java

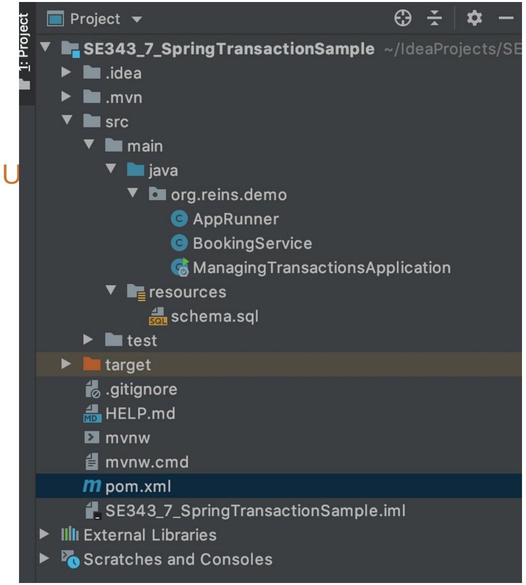
```
@SpringBootApplication
public class ManagingTransactionsApplication {
   public static void main(String[] args) {
      SpringApplication.run(ManagingTransactionsApplication.class, args);
   }
}
```

schema.sql

```
drop table BOOKINGS if exists;
create table BOOKINGS(ID serial, FIRST_NAME varchar(5) NOT NU
```

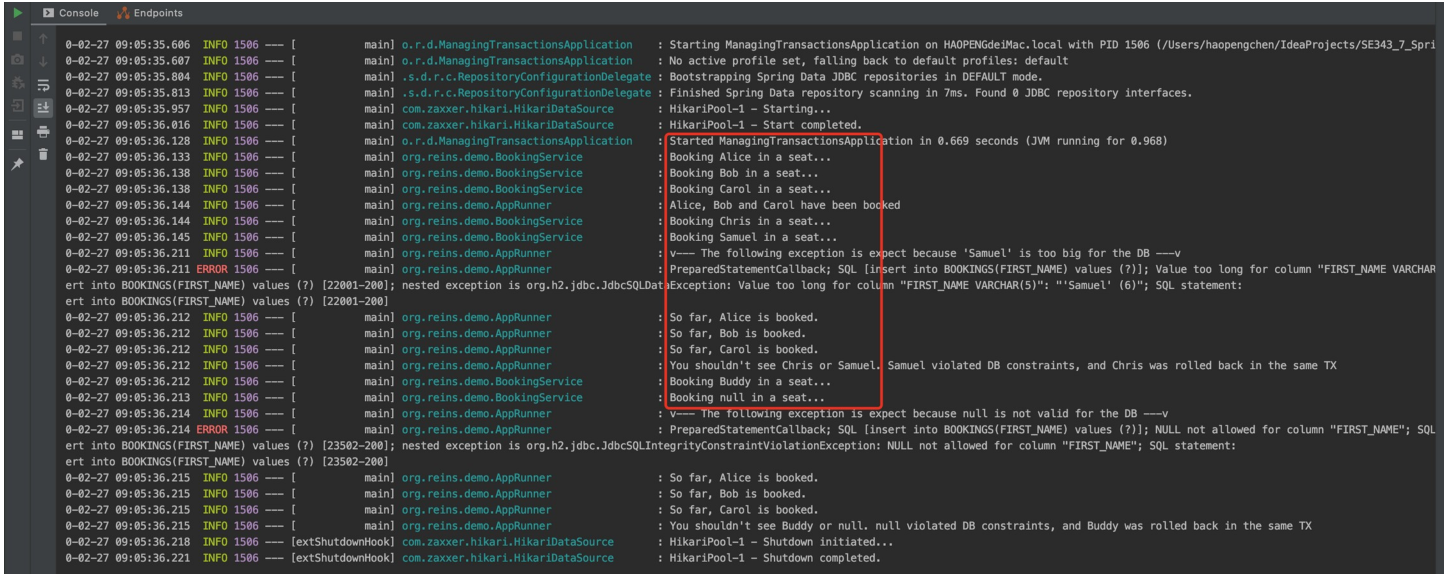
pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```



```
1: Project    Project ▾
▼ SE343_7_SpringTransactionSample ~/IdeaProjects/SE
  ▶ .idea
  ▶ .mvn
  ▼ src
    ▼ main
      ▼ java
        ▼ org.reins.demo
            AppRunner
            BookingService
            ManagingTransactionsApplication
      ▼ resources
          schema.sql
    ▶ test
  ▶ target
    .gitignore
    HELP.md
    mvnw
    mvnw.cmd
    pom.xml
    SE343_7_SpringTransactionSample.iml
▶ External Libraries
▶ Scratches and Consoles
```

Managing Transactions in Spring



```
0-02-27 09:05:35.606  INFO 1506 --- [           main] o.r.d.ManagingTransactionsApplication    : Starting ManagingTransactionsApplication on HAOPENGdeiMac.local with PID 1506 (/Users/haopengchen/IdeaProjects/SE343_7_Spri
0-02-27 09:05:35.607  INFO 1506 --- [           main] o.r.d.ManagingTransactionsApplication    : No active profile set, falling back to default profiles: default
0-02-27 09:05:35.804  INFO 1506 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JDBC repositories in DEFAULT mode.
0-02-27 09:05:35.813  INFO 1506 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 7ms. Found 0 JDBC repository interfaces.
0-02-27 09:05:35.957  INFO 1506 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
0-02-27 09:05:36.016  INFO 1506 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
0-02-27 09:05:36.128  INFO 1506 --- [           main] o.r.d.ManagingTransactionsApplication    : Started ManagingTransactionsApplication in 0.669 seconds (JVM running for 0.968)
0-02-27 09:05:36.133  INFO 1506 --- [           main] org.reins.demo.BookingService            : Booking Alice in a seat...
0-02-27 09:05:36.138  INFO 1506 --- [           main] org.reins.demo.BookingService            : Booking Bob in a seat...
0-02-27 09:05:36.138  INFO 1506 --- [           main] org.reins.demo.BookingService            : Booking Carol in a seat...
0-02-27 09:05:36.144  INFO 1506 --- [           main] org.reins.demo.AppRunner                 : Alice, Bob and Carol have been booked
0-02-27 09:05:36.144  INFO 1506 --- [           main] org.reins.demo.BookingService            : Booking Chris in a seat...
0-02-27 09:05:36.145  INFO 1506 --- [           main] org.reins.demo.BookingService            : Booking Samuel in a seat...
0-02-27 09:05:36.211  INFO 1506 --- [           main] org.reins.demo.AppRunner                 : v--- The following exception is expect because 'Samuel' is too big for the DB ---v
0-02-27 09:05:36.211 ERROR 1506 --- [           main] org.reins.demo.AppRunner                 : PreparedStatementCallback; SQL [insert into BOOKINGS(FIRST_NAME) values (?)]; Value too long for column "FIRST_NAME VARCHAR
ert into BOOKINGS(FIRST_NAME) values (?) [22001-200]; nested exception is org.h2.jdbc.JdbcSQLDataException: Value too long for column "FIRST_NAME VARCHAR(5)": "'Samuel' (6)"; SQL statement:
ert into BOOKINGS(FIRST_NAME) values (?) [22001-200]
0-02-27 09:05:36.212  INFO 1506 --- [           main] org.reins.demo.AppRunner                 : So far, Alice is booked.
0-02-27 09:05:36.212  INFO 1506 --- [           main] org.reins.demo.AppRunner                 : So far, Bob is booked.
0-02-27 09:05:36.212  INFO 1506 --- [           main] org.reins.demo.AppRunner                 : So far, Carol is booked.
0-02-27 09:05:36.212  INFO 1506 --- [           main] org.reins.demo.AppRunner                 : You shouldn't see Chris or Samuel. Samuel violated DB constraints, and Chris was rolled back in the same TX
0-02-27 09:05:36.212  INFO 1506 --- [           main] org.reins.demo.BookingService            : Booking Buddy in a seat...
0-02-27 09:05:36.213  INFO 1506 --- [           main] org.reins.demo.BookingService            : Booking null in a seat...
0-02-27 09:05:36.214  INFO 1506 --- [           main] org.reins.demo.AppRunner                 : v--- The following exception is expect because null is not valid for the DB ---v
0-02-27 09:05:36.214 ERROR 1506 --- [           main] org.reins.demo.AppRunner                 : PreparedStatementCallback; SQL [insert into BOOKINGS(FIRST_NAME) values (?)]; NULL not allowed for column "FIRST_NAME"; SQL
ert into BOOKINGS(FIRST_NAME) values (?) [23502-200]; nested exception is org.h2.jdbc.JdbcSQLIntegrityConstraintViolationException: NULL not allowed for column "FIRST_NAME"; SQL statement:
ert into BOOKINGS(FIRST_NAME) values (?) [23502-200]
0-02-27 09:05:36.215  INFO 1506 --- [           main] org.reins.demo.AppRunner                 : So far, Alice is booked.
0-02-27 09:05:36.215  INFO 1506 --- [           main] org.reins.demo.AppRunner                 : So far, Bob is booked.
0-02-27 09:05:36.215  INFO 1506 --- [           main] org.reins.demo.AppRunner                 : So far, Carol is booked.
0-02-27 09:05:36.215  INFO 1506 --- [           main] org.reins.demo.AppRunner                 : You shouldn't see Buddy or null. null violated DB constraints, and Buddy was rolled back in the same TX
0-02-27 09:05:36.218  INFO 1506 --- [extShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown initiated...
0-02-27 09:05:36.221  INFO 1506 --- [extShutdownHook] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Shutdown completed.
```

A Transaction Example

- Bank.java

```
@Entity
@Table(name = "bank")
public class Bank {
    @Id
    @Column(name = "account", nullable = false, length = 45)
    private String id;

    @Column(name = "balance")
    private Integer balance;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public Integer getBalance() {
        return balance;
    }

    public void setBalance(Integer balance) {
        this.balance = balance;
    }

}
```

- BankRepository.java

```
public interface BankRepository extends JpaRepository<Bank, String>
}
```

- BankController.java

```
@RestController
public class BankController {
    @Autowired
    private BankService bankService;

    @GetMapping(value = "/transfer")
    public String transfer() {
        System.out.println("Transferring...");
        bankService.transfer("Tom", "Jerry", 80);
        return "Done";
    }

}
```

A Transaction Example

- BankService.java

```
public interface BankService {
    public void transfer(String from, String to, int amount);
}
```

- BankServiceImpl.java

```
@Service
public class BankServiceImpl implements BankService{

    @Autowired
    private BankDao bankDao;

    @Override
    @Transactional
    public void transfer(String from, String to, int amount){
        System.out.println(from + " " + to + amount);
        bankDao.withdraw(from, amount);
//      int result = 10 / 0;
        try {
            bankDao.deposit(to, amount);
        }catch (Exception e){
            e.printStackTrace();
        }
//      int result = 10 / 0;
    };
}
```

- BankDao.java

```
public interface BankDao {
    public void withdraw(String from, int amount);
    public void deposit(String to, int amount);
}
```

- BankDaoImpl.java

```
@Repository
public class BankDaoImpl implements BankDao {
    @Autowired
    private BankRepository bankRepository;

    @Transactional
    public void withdraw(String from, int amount) {
        Bank out = bankRepository.findById(from).get();
        out.setBalance(out.getBalance() - amount);
//          int result = 10 / 0;
    };

    @Transactional (propagation = Propagation.REQUIRES_NEW)
    public void deposit(String to, int amount) {
        Bank in = bankRepository.findById(to).get();
        in.setBalance(in.getBalance() + amount);
//          int result = 10 / 0;
    };
}
```

A Transaction Example

- SQL Script

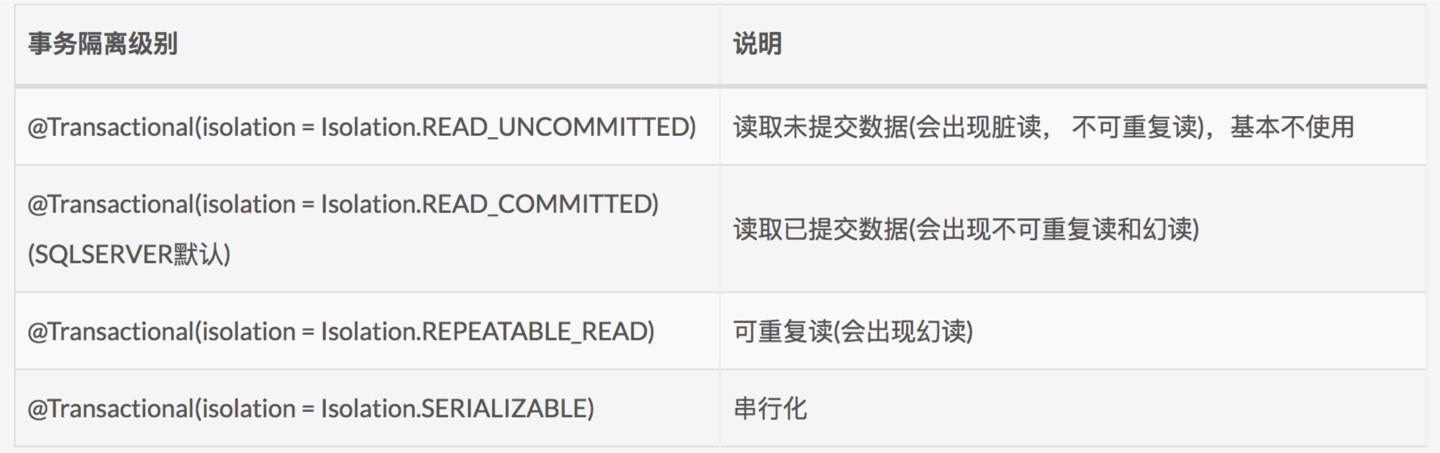
```
DROP TABLE IF EXISTS `bank`;
CREATE TABLE `bank` (
`account` varchar(45) NOT NULL,
`balance` int DEFAULT NULL,
PRIMARY KEY (`account`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

A Transaction Example

	transfer	withdraw	deposit	result
1	正常	正常	正常	Tom - 80; Jerry + 80
2	result = 10 / 0	正常	正常	整个事务回滚
3	正常	result = 10 / 0	正常	整个事务回滚
4	正常	正常	result = 10 / 0	整个事务回滚
5	正常	正常	正常 *REQUIRES_NEW*	正常
6	result = 10 / 0，在 deposit 之前	正常	正常 *REQUIRES_NEW*	整个事务回滚
7	result = 10 / 0，在 deposit 之后	正常	正常 *REQUIRES_NEW*	deposit 成功， withdraw 和 transfer 事务回滚
8	正常	result = 10 / 0	正常 *REQUIRES_NEW*	整个事务回滚
9	正常	正常	result = 10 / 0 *REQUIRES_NEW*	withdraw 和 transfer 成功， deposit 事务回滚

Managing Transactions in Spring

- @Transactional isolation



事务隔离级别	说明
@Transactional(isolation = Isolation.READ_UNCOMMITTED)	读取未提交数据(会出现脏读， 不可重复读)，基本不使用
@Transactional(isolation = Isolation.READ_COMMITTED) (SQLSERVER默认)	读取已提交数据(会出现不可重复读和幻读)
@Transactional(isolation = Isolation.REPEATABLE_READ)	可重复读(会出现幻读)
@Transactional(isolation = Isolation.SERIALIZABLE)	串行化

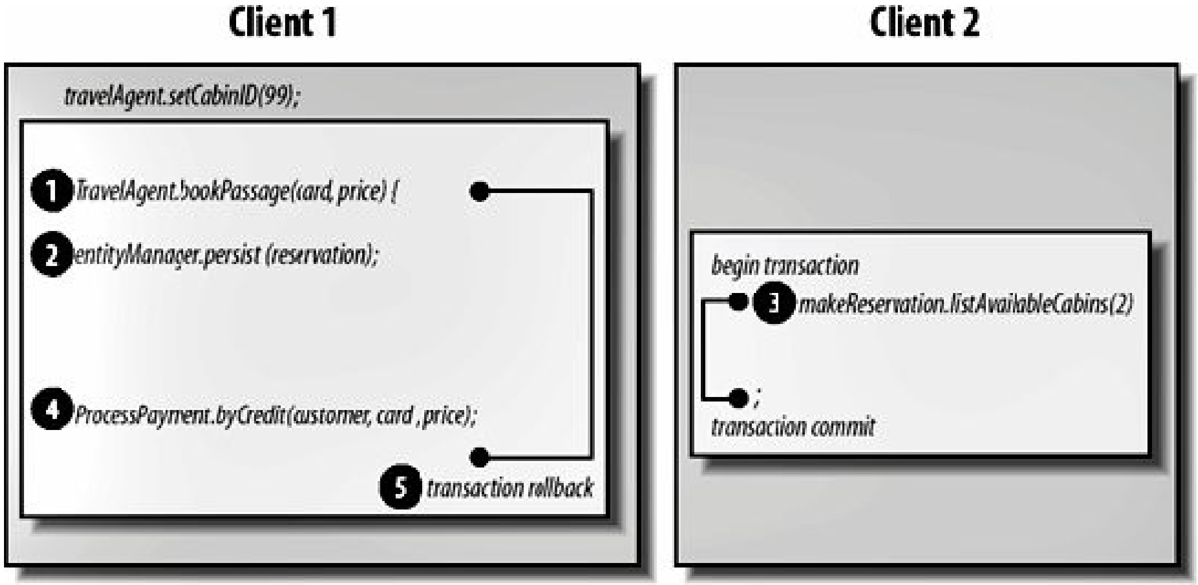
Isolation and Database Locking

- Let's think about two separate client applications accessing the same data specifically.

```
public List listAvailableCabins(int bedCount) {
        Query query = entityManager.createQuery(
                        "SELECT name FROM Cabin c
                  WHERE c.ship = :ship AND
                         c.bedCount = :beds AND NOT ANY (
                     SELECT cabin from Reservation res
                           WHERE res.cruise = :cruise");
  query.setParameter("ship", cruise.getShip( ));
  query.setParameter("beds", bedCount);
  query.setParameter("cruise", cruise);
  return query.getResultList( );
}
```

- For this example, assume that both methods have a transaction attribute of `Required` .

Dirty Reads



Client 1

travelAgent.setCabinID(99);

1 TravelAgent.bookPassage(card, price) {

2 entityManager.persist (reservation);

4 ProcessPayment.byCredit(customer, card ,price);

5 transaction rollback

Client 2

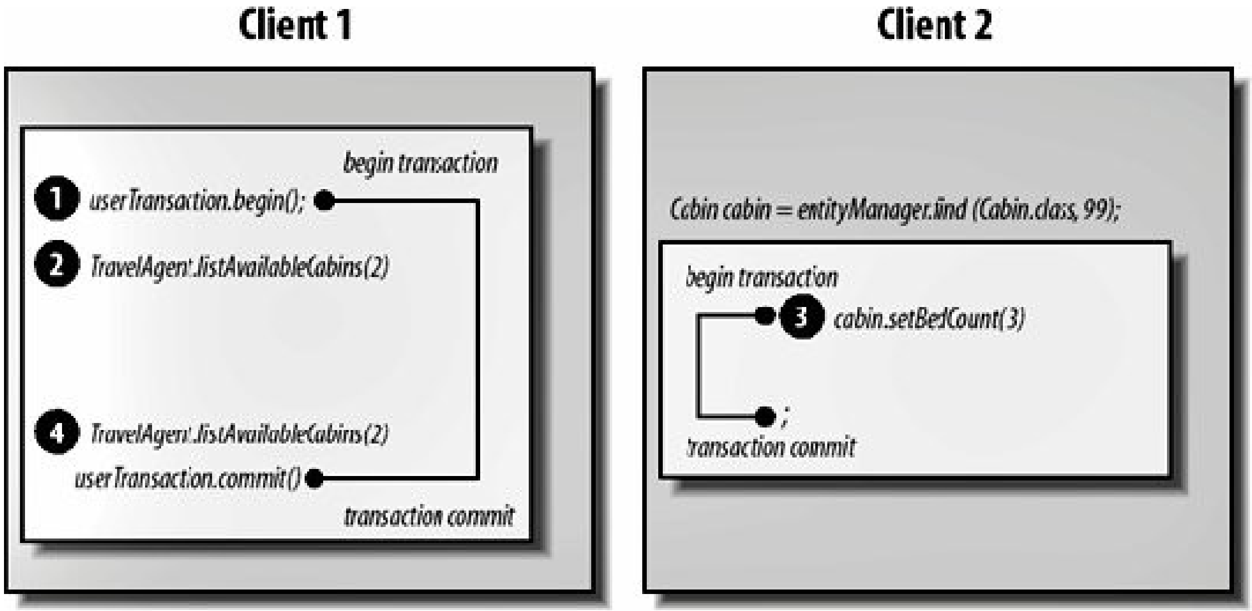
begin transaction

3 makeReservation.listAvailableCabins(2)

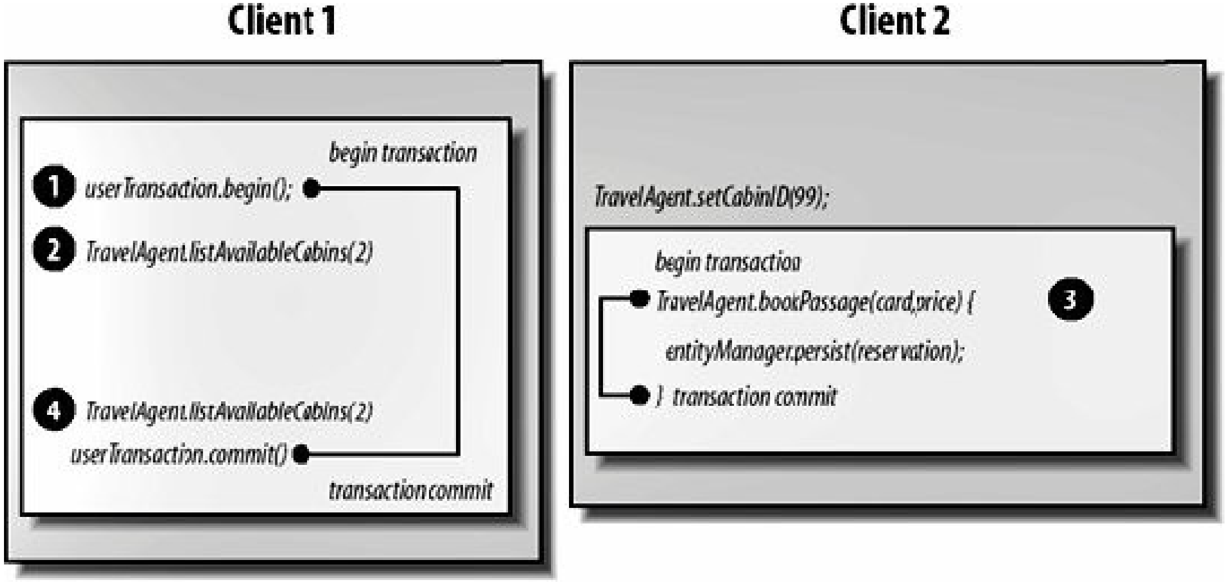
;

transaction commit

Nonrepeatable Reads


Client 1
begin transaction
1 userTransaction.begin();
2 TravelAgent.listAvailableCabins(2)
4 TravelAgent.listAvailableCabins(2)
userTransaction.commit()
transaction commit
Client 2
Cabin cabin = entityManager.find (Cabin.class, 99);
begin transaction
3 cabin.setBedCount(3)
;
transaction commit

Phantom Reads



Client 1:

```
userTransaction.begin();
TravelAgent.listAvailableCabins(2)
TravelAgent.listAvailableCabins(2)
userTransaction.commit()
```

Client 2:

```
TravelAgent.setCabinID(99);
TravelAgent.bookPassage(card,price) {
  entityManager.persist(reservation);
}
```

Database Locks

- Read locks
  - Read locks prevent other transactions from changing data read during a transaction until the transaction ends, thus preventing nonrepeatable reads.
  - Other transactions can read the data but not write to it. The current transaction is also prohibited from making changes.
- Write locks
  - Write locks are used for updates. A write lock prevents other transactions from changing the data until the current transaction is complete but allows dirty reads by other transactions and by the current transaction itself.
  - In other words, the transaction can read its own uncommitted changes.

Database Locks

- Exclusive write locks
  - Exclusive write locks are used for updates. An exclusive write lock prevents other transactions from reading or changing the data until the current transaction is complete.
  - It also prevents dirty reads by other transactions.

- Snapshots
  - A snapshot is a frozen view of the data that is taken when a transaction begins. Some databases get around locking by providing every transaction with its own snapshot.
  - Snapshots can prevent dirty reads, nonrepeatable reads, and phantom reads. They can be problematic because the data is not real-time data; it is old the instant the snapshot is taken.

Transaction Isolation Levels

- Read Uncommitted
  - The transaction can read uncommitted data (i.e., data changed by a different transaction that is still in progress).
  - Dirty reads, nonrepeatable reads, and phantom reads can occur.
  - Bean methods with this isolation level can read uncommitted changes.
- Read Committed
  - The transaction cannot read uncommitted data; data that is being changed by a different transaction cannot be read.
  - Dirty reads are prevented; nonrepeatable reads and phantom reads can occur.
  - Bean methods with this isolation level cannot read uncommitted data.

Transaction Isolation Levels

- Repeatable Read
  - The transaction cannot change data that is being read by a different transaction.
  - Dirty reads and nonrepeatable reads are prevented; phantom reads can occur.
  - Bean methods with this isolation level have the same restrictions as those in the Read Committed level and can execute only repeatable reads.

- Serializable
  - The transaction has exclusive read and update privileges; different transactions can neither read nor write to the same data.
  - Dirty reads, nonrepeatable reads, and phantom reads are prevented.
  - This isolation level is the most restrictive.

Controlling isolation levels

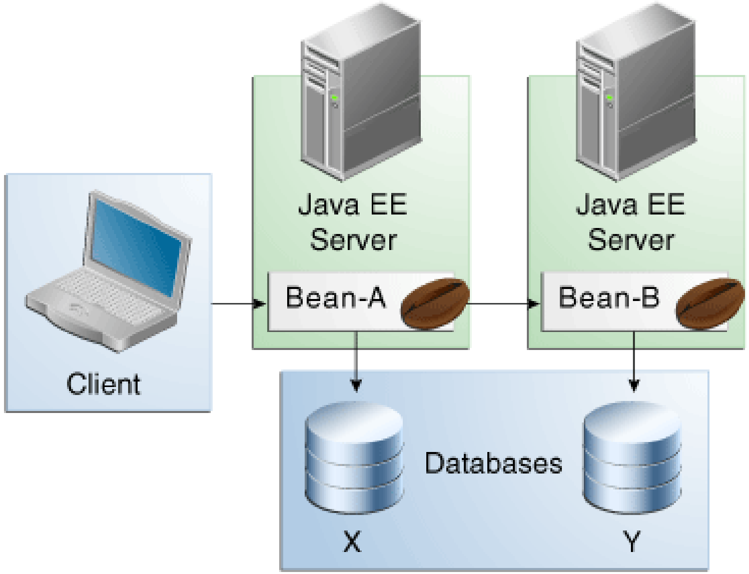
- You to specify the transaction isolation level using the database's API.
- For example:

```
   DataSource source = (javax.sql.DataSource)

jndiCntxt.lookup("java:comp/env/jdbc/titanDB");
   Connection con = source.getConnection( );
   con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

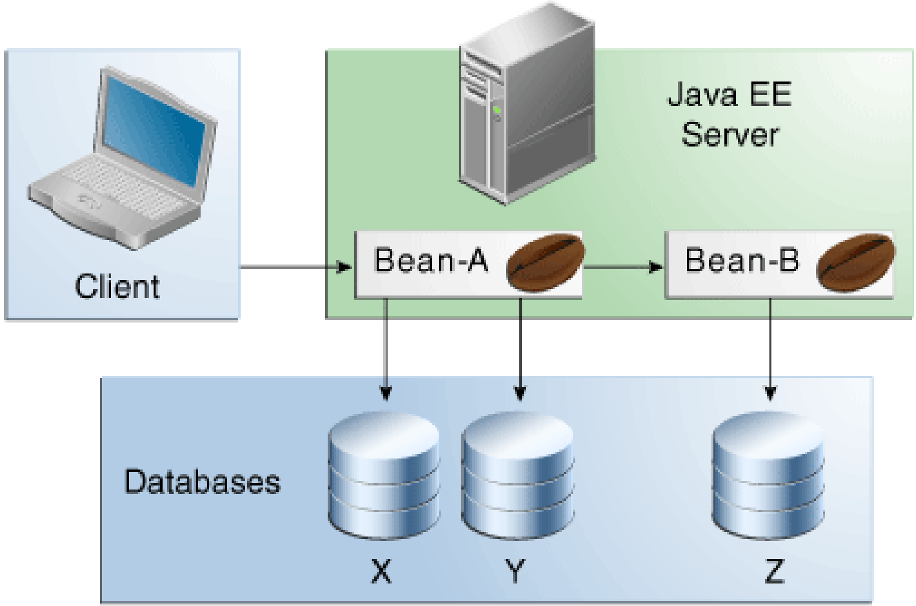
- In spring:

```
@Transactional(isolation = Isolation.READ_UNCOMMITTED)
@Transactional(isolation = Isolation.READ_COMMITTED)
@Transactional(isolation = Isolation.REPEATABLE_READ)
@Transactional(isolation = Isolation.SERIALIZABLE)
```

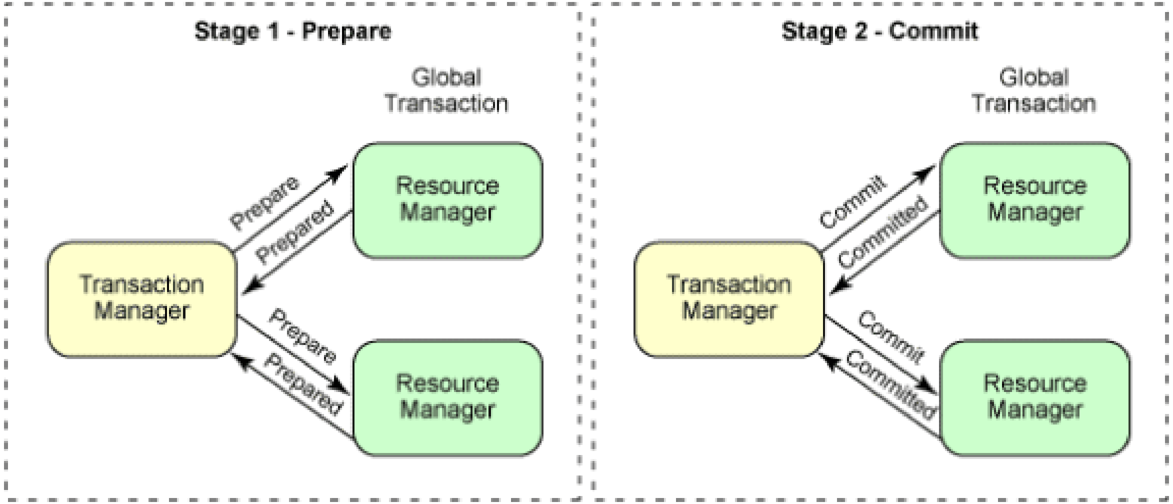
Updating Multiple Databases



Updating Multiple Databases



Two-phase commit



Concurrency

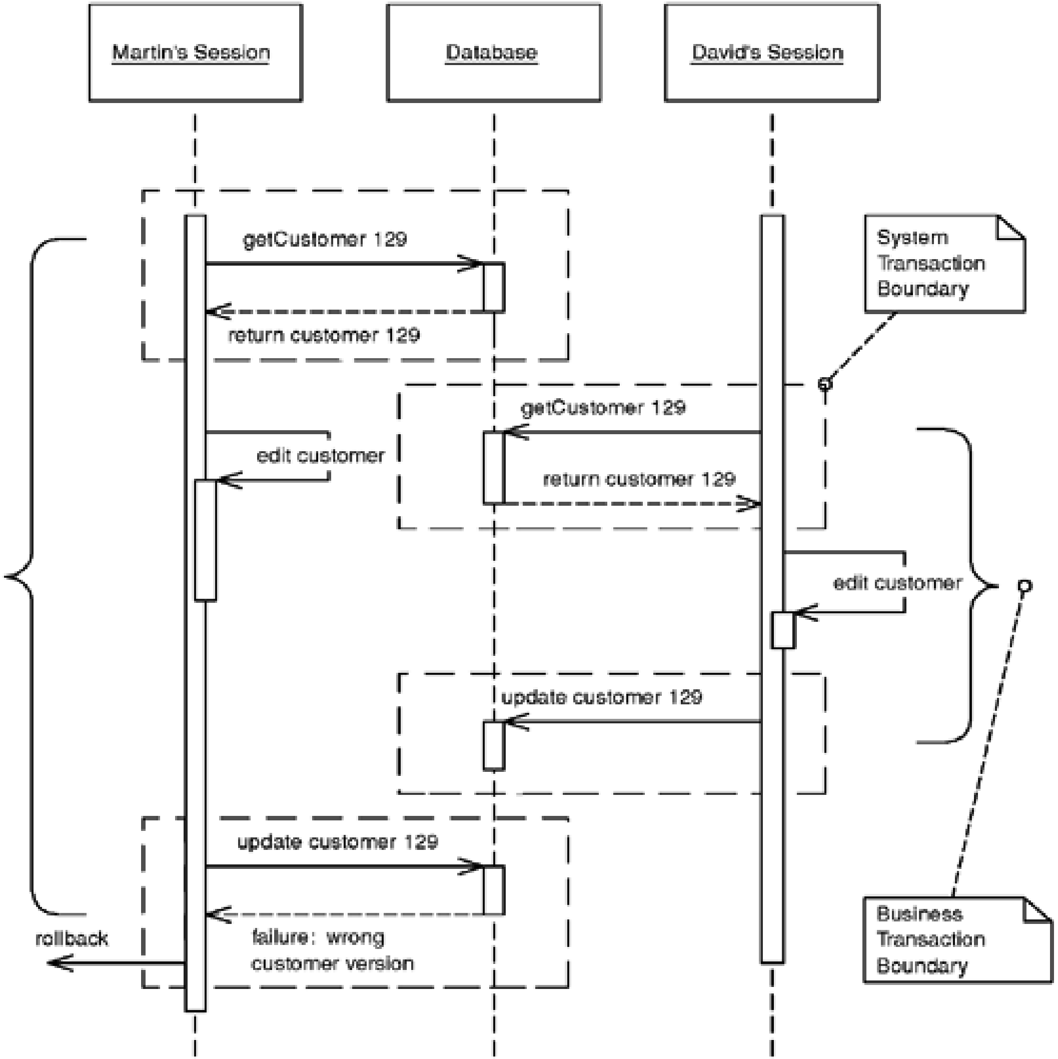
- Since we use O/R mapping, and it is an offline way to access database, we should manage the concurrent access to data.

- The core is how to prevent confliction between business transactions.

- Locking is the solution
  - What kind of lock?

Optimistic Offline Lock

- Optimistic Offline Lock solves this problem by validating that the changes about to be committed by one session don't conflict with the changes of another session.



Optimistic Offline Lock-How It Works

- An Optimistic Offline Lock is obtained by validating that, in the time since a session loaded a record, another session hasn't altered it.
  - It can be acquired at any time but is valid only during the system transaction in which it is obtained.
  - Thus, in order that a business transaction not corrupt record data it must acquire an Optimistic Offline Lock for each member of its change set during the system transaction in which it applies changes to the database.

- The most common implementation is to associate a version number with each record in your system.
  - With an RDBMS data store the verification is a matter of adding the version number to the criteria of any SQL statements used to update or delete a record.

Optimistic Offline Lock-Example

- Our data is stored in a relational database,
  - so each table must also store version and modification data.
  - Here's the schema for a customer table as well as the standard CRUD SQL necessary to support the Optimistic Offline Lock:

  - table customer:

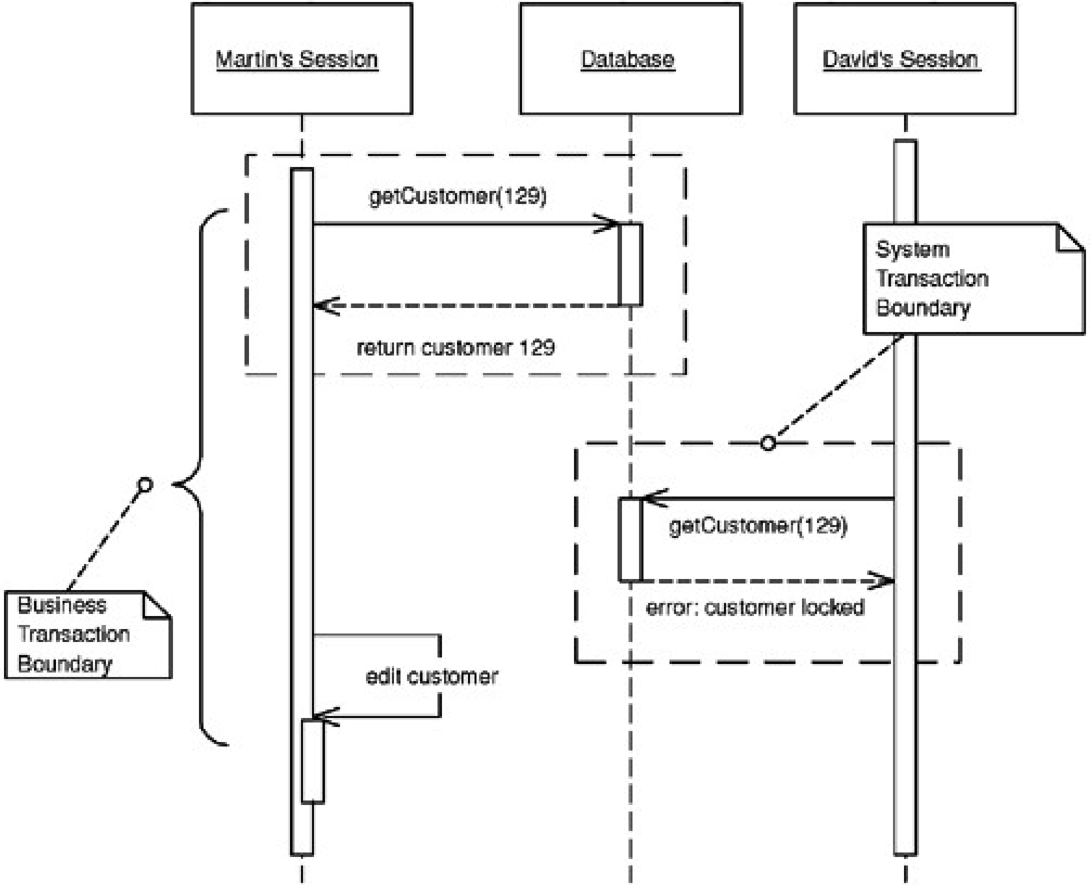
```
create table customer(id bigint primary key, name varchar, createdby varchar, created datetime,
  modifiedby varchar, modified datetime, version int)
```

  - SQL customer CRUD:

```
INSERT INTO customer VALUES (?, ?, ?, ?, ?, ?, ?)
SELECT * FROM customer WHERE id = ?
UPDATE customer SET name = ?, modifiedBy = ?, modified = ?, version = ?
    WHERE id = ? and version = ?
DELETE FROM customer WHERE id = ? and version = ?
```

Pessimistic Offline Lock

- Pessimistic Offline Lock prevents conflicts by avoiding them altogether.
  - It forces a business transaction to acquire a lock on a piece of data before it starts to use it, so that, most of the time, once you begin a business transaction you can be pretty sure you'll complete it without being bounced by concurrency control.



Pessimistic Offline Lock-How It Works

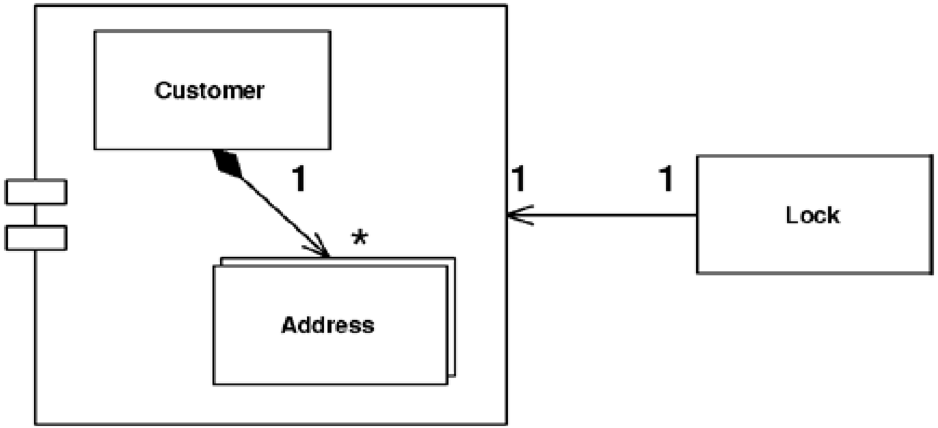
- You implement Pessimistic Offline Lock in three phases:
  - determining what type of locks you need,
  - building a lock manager,
  - and defining procedures for a business transaction to use locks.

- Lock types:
  - exclusive write lock
  - exclusive read lock
  - read/write lock
    - Read and write locks are mutually exclusive.
    - Concurrent read locks are acceptable.

- In choosing the correct lock type think about maximizing system concurrency, meeting business needs, and minimizing code complexity.
  - Also keep in mind that the locking strategy must be understood by domain modelers and analysts.

Coarse-Grained Lock

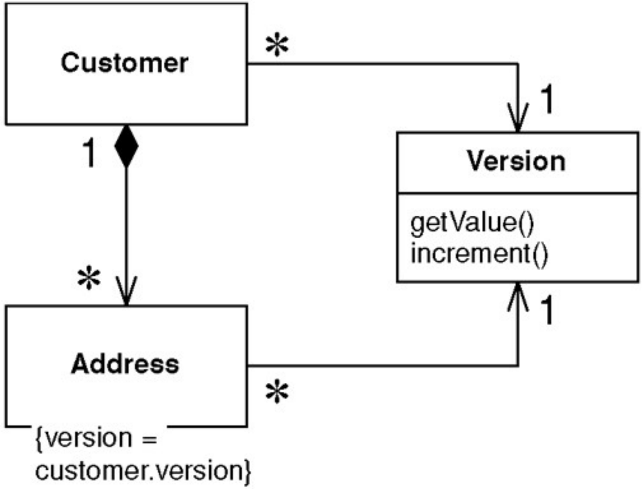
- Locks a set of related objects with a single lock.



- A Coarse-Grained Lock
  - is a single lock that covers many objects.
  - It not only simplifies the locking action itself but also frees you from having to load all the members of a group in order to lock them.

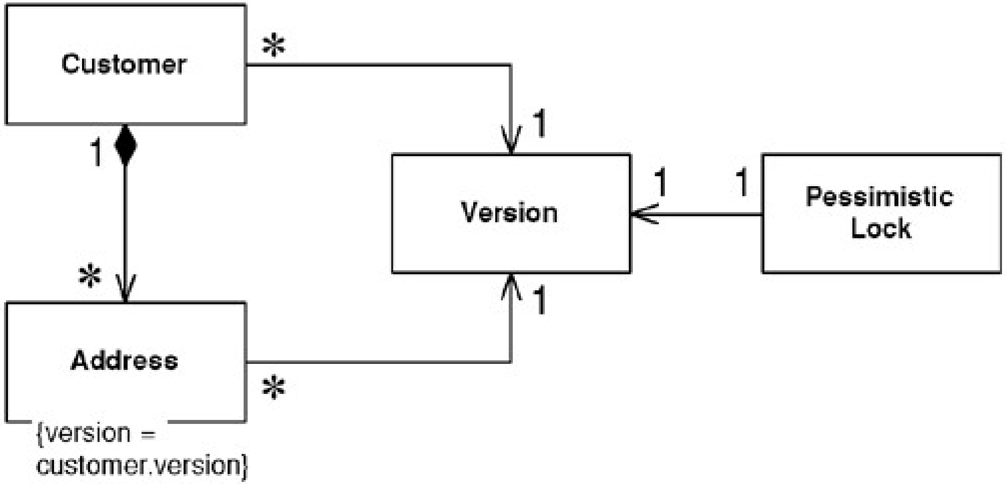
Coarse-Grained Lock-How It Works

- With Optimistic Offline Lock, having each item in a group share a version creates the single point of contention, which means sharing the same version, not an equal version.
  - Incrementing this version will lock the entire group with a shared lock.
  - Set up your model to point every member of the group at the shared version and you have certainly minimized the path to the point of contention.



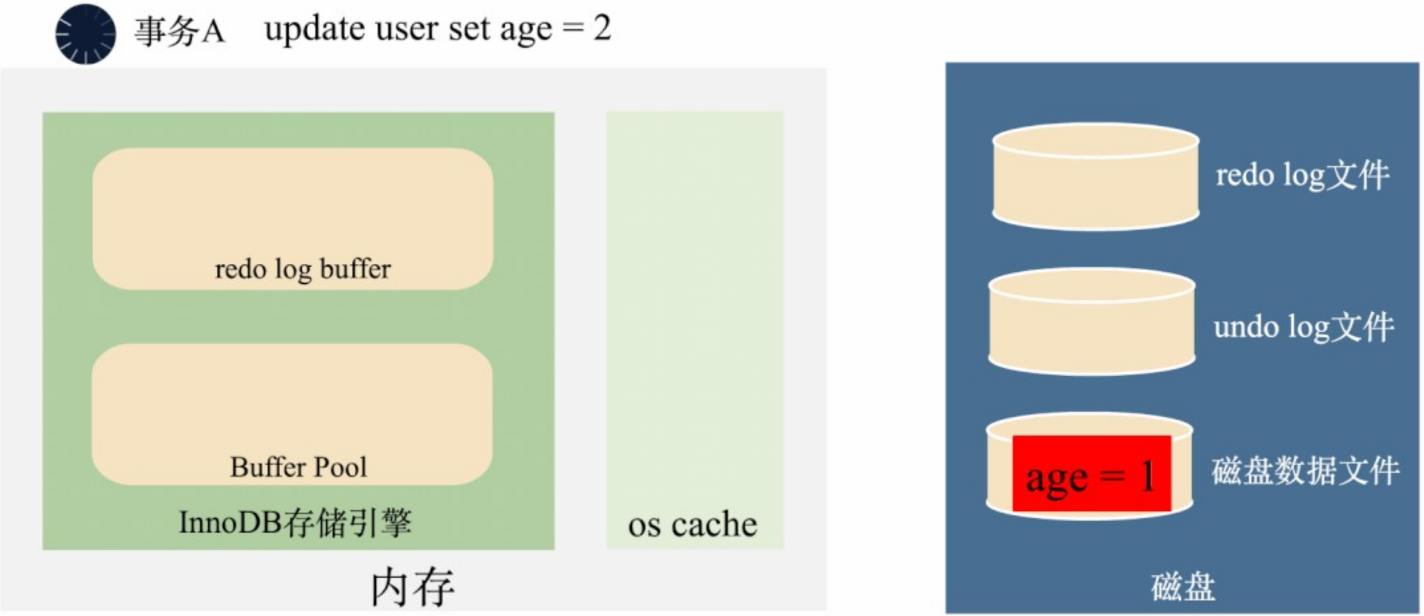
Coarse-Grained Lock-How It Works

- A shared Pessimistic Offline Lock requires that each member of the group share some sort of lockable token, on which it must then be acquired.
  - As Pessimistic Offline Lock is often used as a complement to Optimistic Offline Lock, a shared version object makes an excellent candidate for the lockable token role.



MySQL 中的事务

- 3000 帧动画图解 MySQL 为什么需要 binlog 、 redo log 和 undo log
  - https://blog.csdn.net/hollis_chuang/article/details/125383029



References

- The Java EE 8 Tutorial - Transactions
  - https://javaee.github.io/tutorial/transactions.html#BNCIH
- Managing Transactions
  - https://spring.io/guides/gs/managing-transactions/
- Patterns of Enterprise Application Architecture
  - By Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, Randy Stafford
  - Publisher : Addison Wesley
- Software Architecture in Practice, Second Edition
  - By Len Bass, Paul Clements, Rick Kazman
  - Publisher : Addison Wesley

Thank You!