

CS544

LESSON 7

TRANSACTIONS

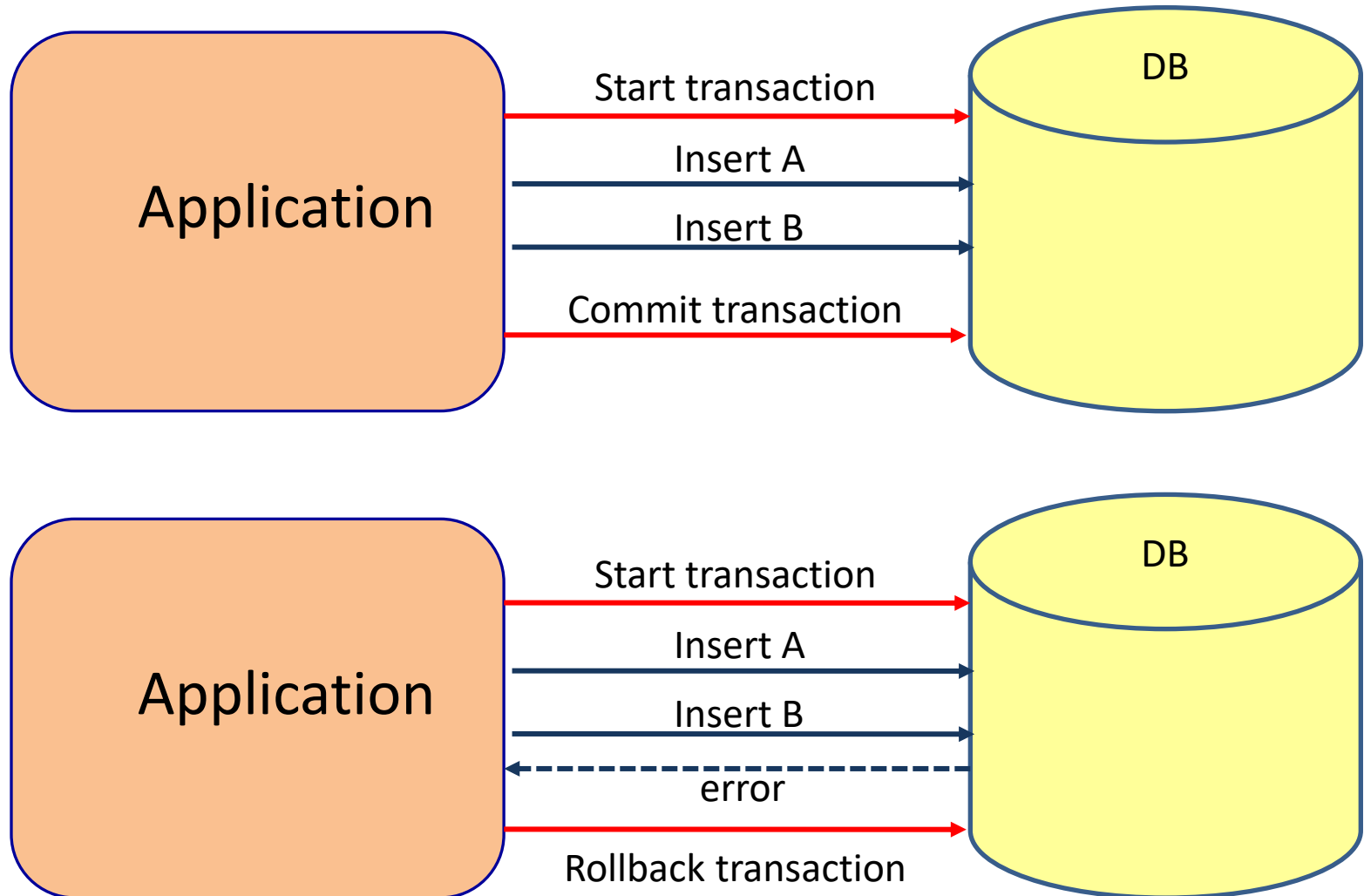
BASICS OF TRANSACTIONS

What is a transaction?

- A unit of actions with the following ACID characteristics:
 - **ATOMICITY**: All changes occur together or no change occurs
 - All-or-nothing
 - **CONSISTENCY**: The transaction transforms the system from one consistent state to another consistent state
 - Transaction must be correct according the application rules
 - **ISOLATION**: Data used in one transaction cannot be used in other transactions until the transaction is committed.
 - **DURABILITY**: Once a transaction is committed, its effects are guaranteed to be persistent



How do transactions work?



Local or global transaction

Transaction propagation

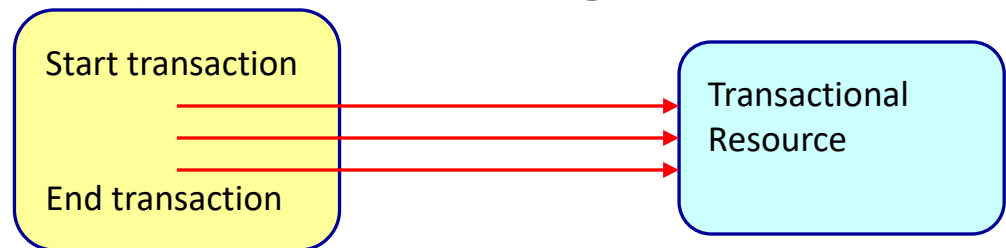
Isolation level

GLOBAL OR LOCAL TRANSACTION

Local or global transactions

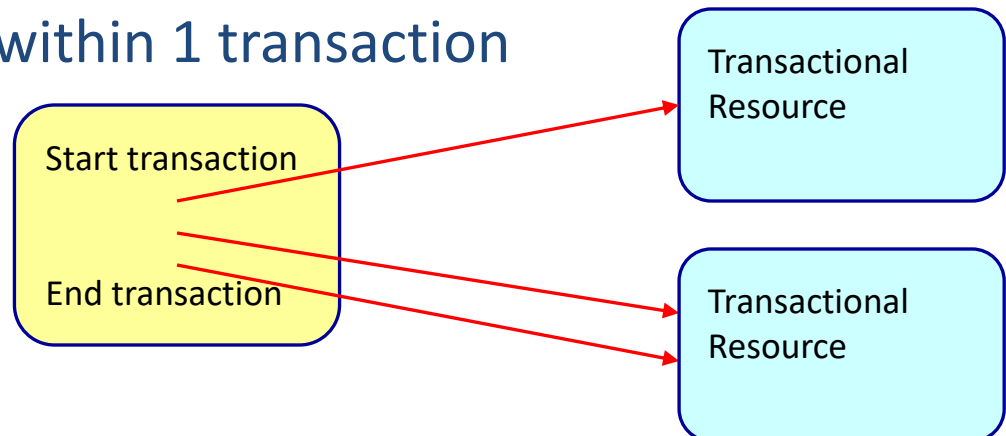
- Local transactions

- 1 transactional resource (database, message bus)



- Global transactions

- More than 1 transactional resource (database, message bus) used within 1 transaction

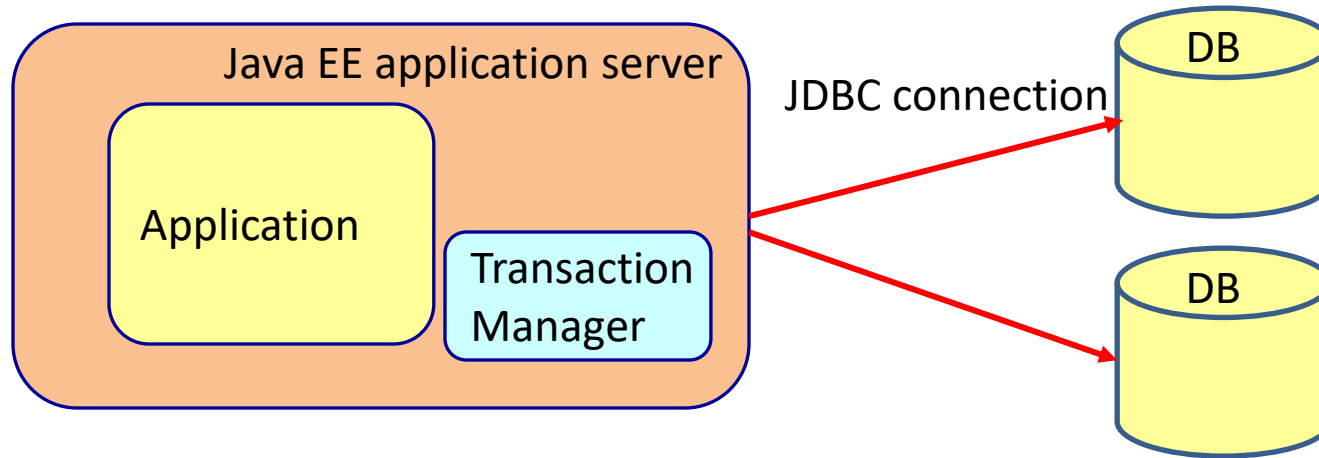


Local transaction



- The transaction is managed by the database
- Simple
- Fast

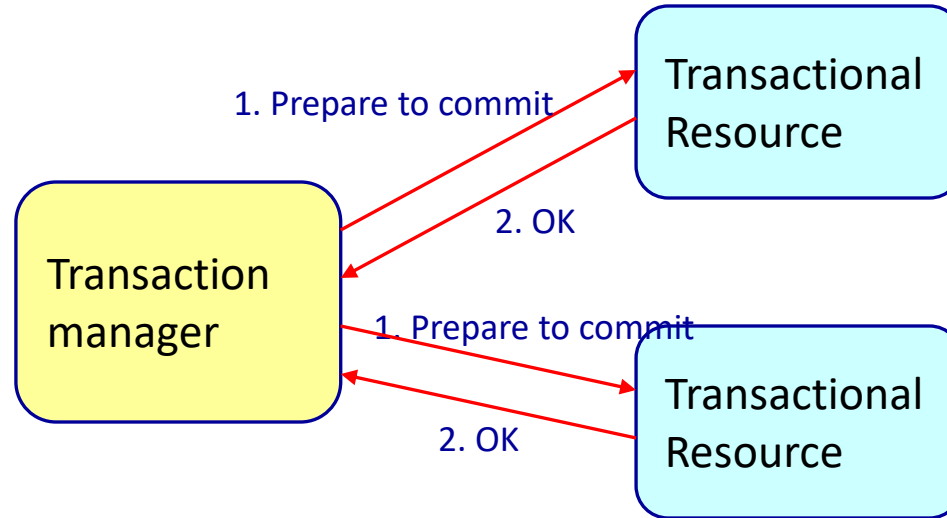
Global transaction



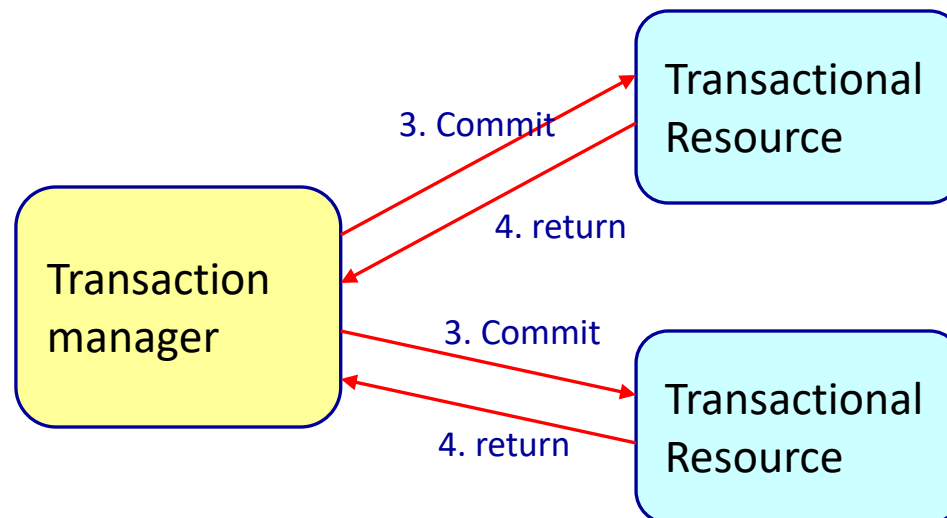
- The transaction is managed by the transaction manager in the Java EE application server
- Also called XA transactions
- Only needed when 2 transactional resources are used within one transaction
- 2 Phase commit

2 phase commit

■ Phase 1

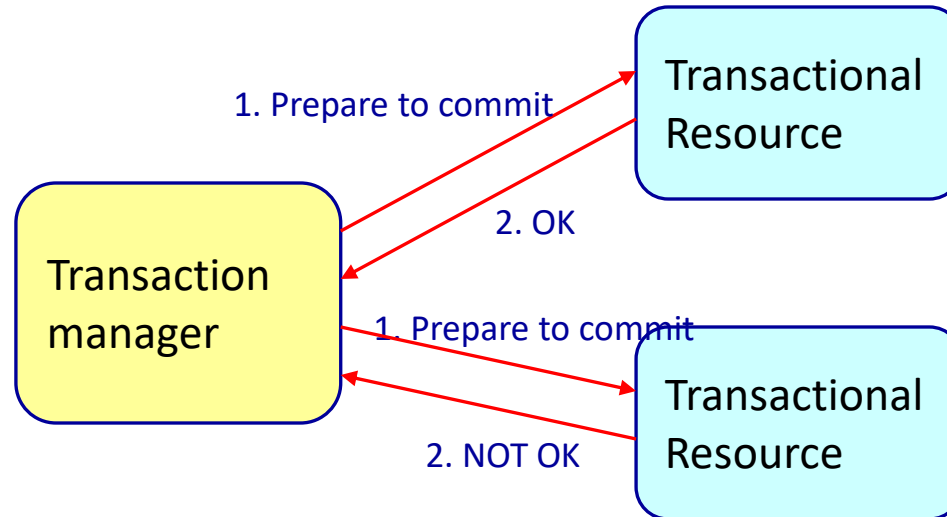


■ Phase 2

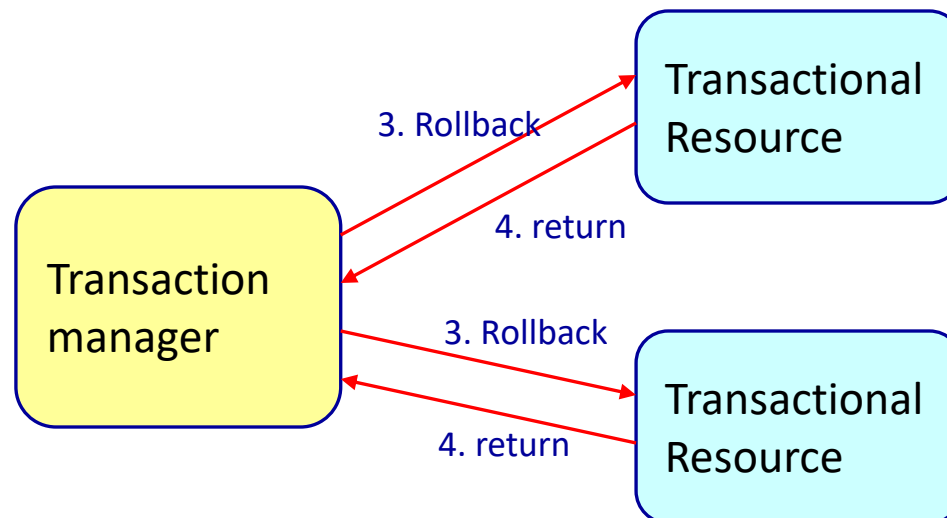


2 phase commit

■ Phase 1



■ Phase 2



Characteristics of XA transactions

- 2 phase commit does not guarantee that nothing can go wrong anymore
- 2 phase commit is slow
 - Often runs over remote connections
- Transactional resources become dependent on each other
 - You have to keep the locks until ALL resources are finished

Main point

- Always try to use local transactions. Only use global transactions when there is no other choice.

Science of Consciousness: In higher states of consciousness one always chooses the path of least resistance.

Local or global transaction

Transaction propagation

Isolation level

TRANSACTION PROPAGATION

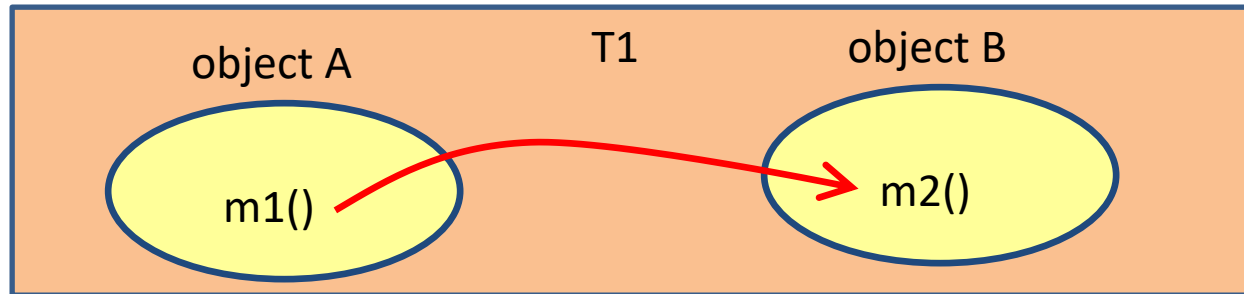
Transaction propagation

- REQUIRED
- REQUIRES_NEW
- MANDATORY
- SUPPORTS
- NEVER
- NOT_SUPPORTED

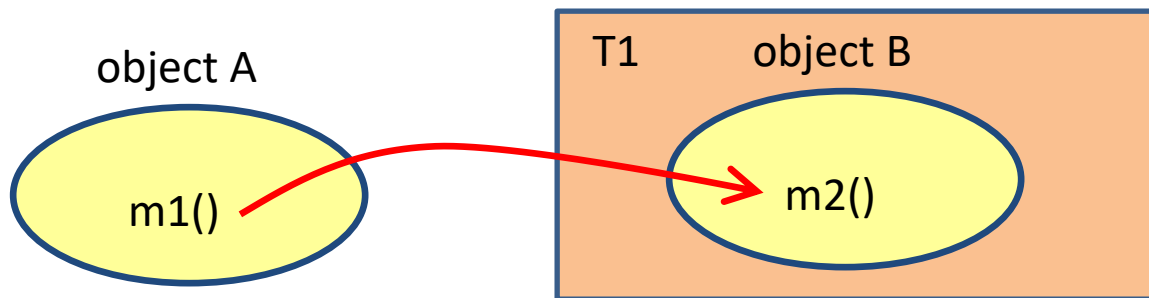
Default, mostly used

Transaction propagation: REQUIRED

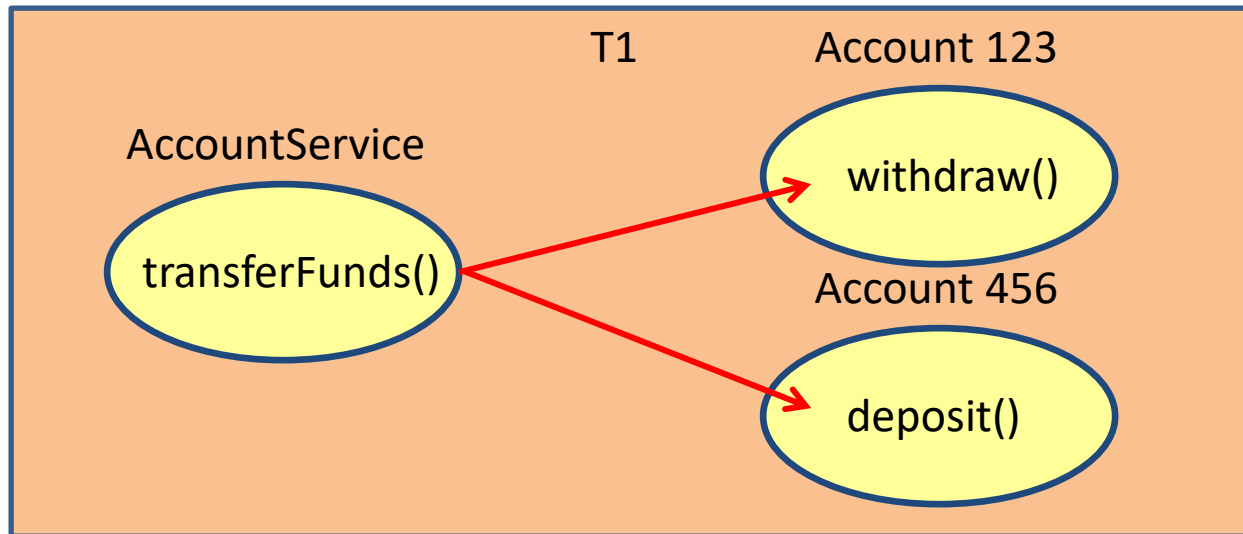
- If the calling method `m1()` runs in a transaction `T1`, then method `m2()` joins the same transaction `T1`



- If the calling method `m1()` does not run in a transaction, then method `m2()` runs in a newly created transaction `T1`

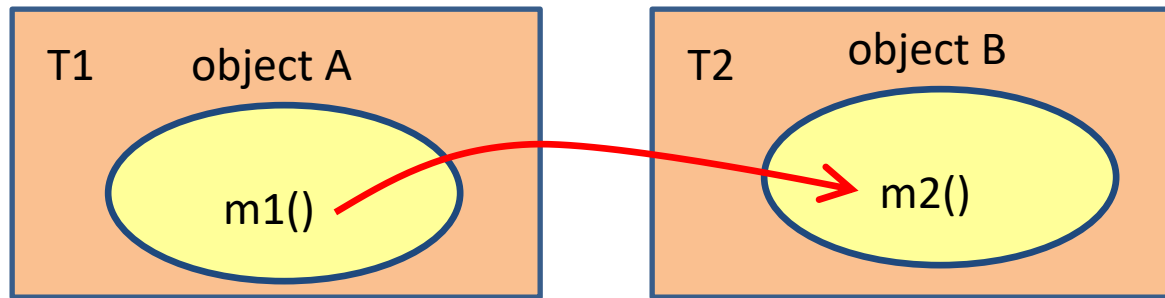


Example of transaction propagation

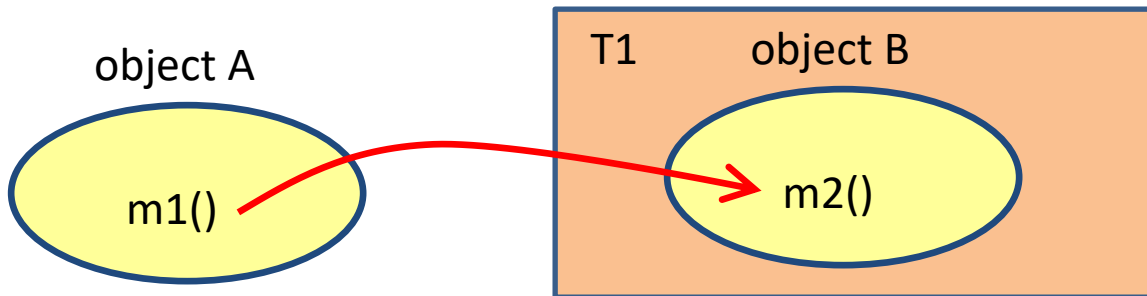


Transaction propagation: REQUIRES_NEW

- If the calling method `m1()` runs in a transaction `T1`, then method `m2()` runs in a new created transaction `T2`

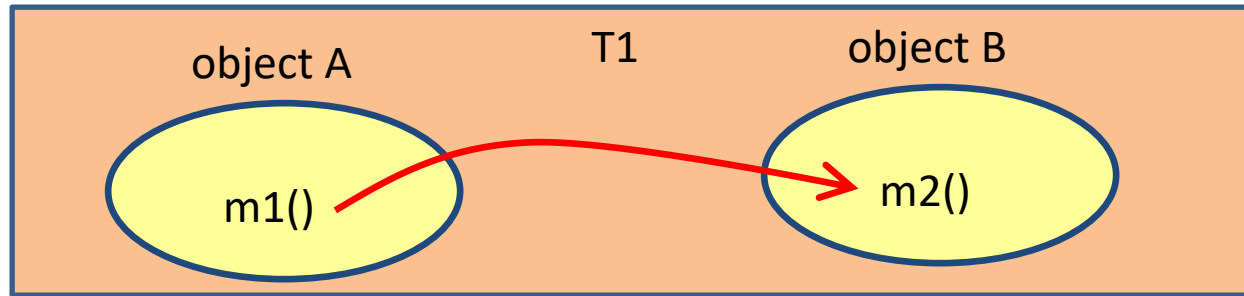


- If the calling method `m1()` does not run in a transaction, then method `m2()` runs in a newly created transaction `T1`

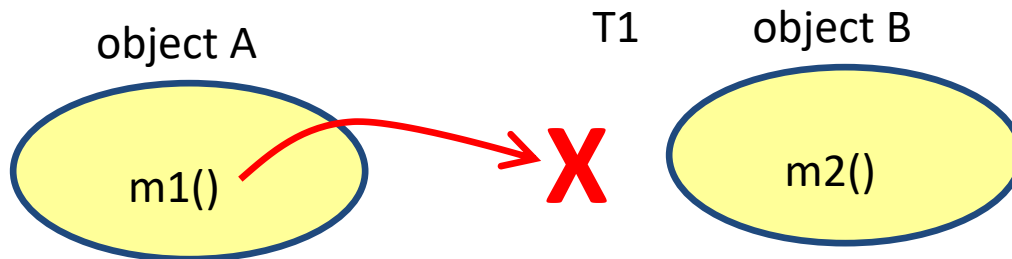


Transaction propagation: MANDATORY

- If the calling method `m1()` runs in a transaction `T1`, then method `m2()` joins the same transaction `T1`

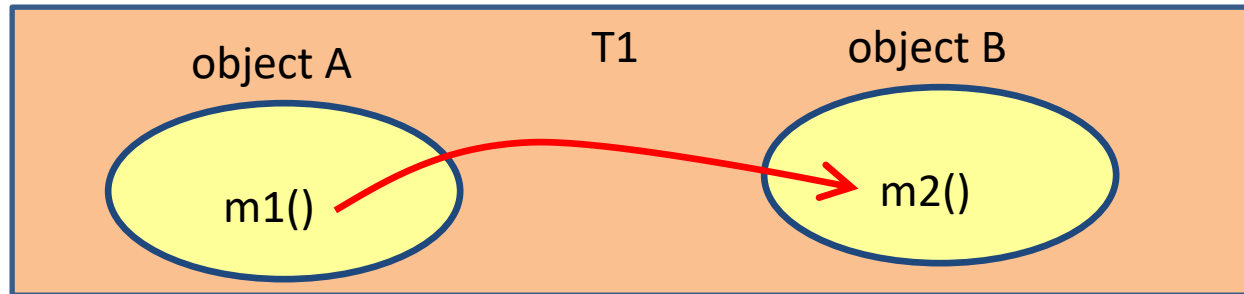


- If the calling method `m1()` does not run in a transaction, an exception is thrown

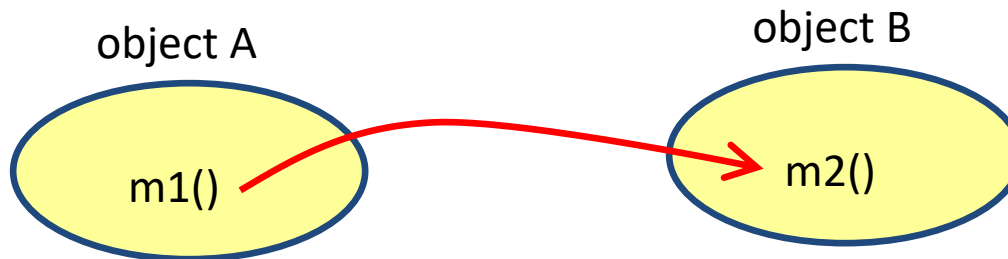


Transaction propagation: SUPPORTS

- If the calling method `m1()` runs in a transaction `T1`, then method `m2()` joins the same transaction `T1`

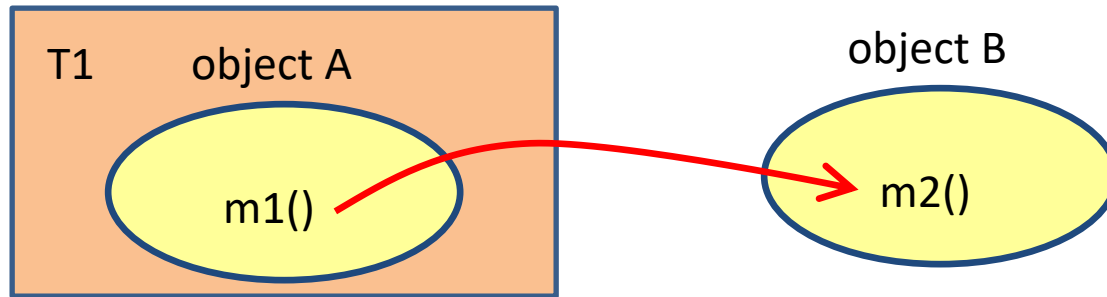


- If the calling method `m1()` does not run in a transaction, then method `m2()` also does not run within a transaction

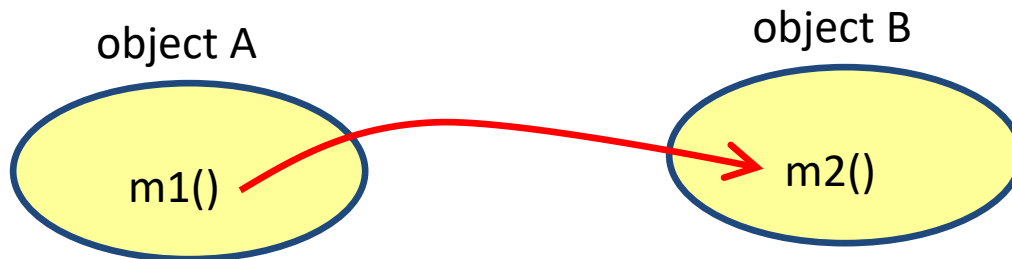


Transaction propagation: NOT SUPPORTED

- If the calling method `m1()` runs in a transaction `T1`, then method `m2()` does not run within a transaction

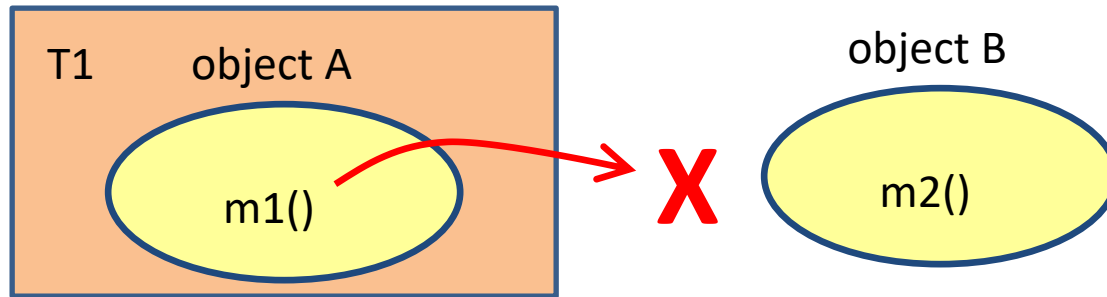


- If the calling method `m1()` does not run in a transaction, then method `m2()` also does not run within a transaction

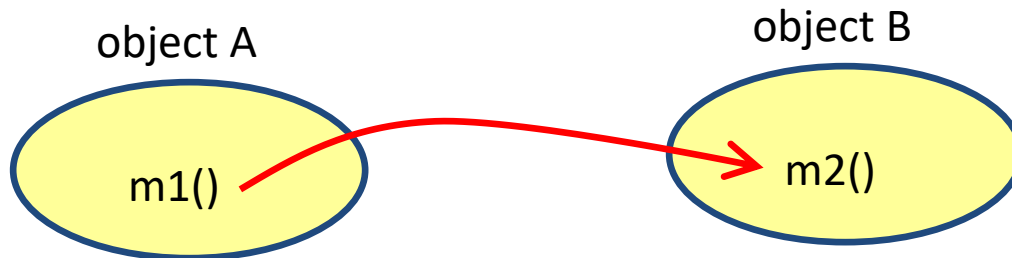


Transaction propagation: NEVER

- If the calling method `m1()` runs in a transaction `T1`, an exception is thrown



- If the calling method `m1()` does not run in a transaction, then method `m2()` also does not run within a transaction



Local or global transaction

Transaction propagation

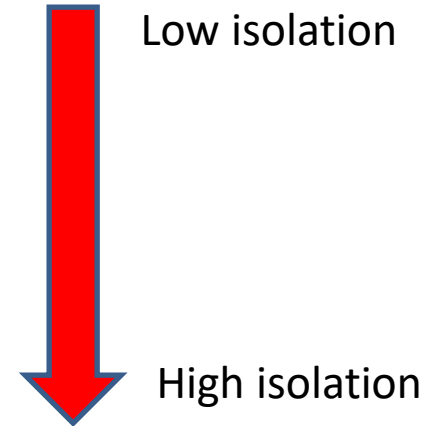
Isolation level

ISOLATION LEVEL

Isolation level

- 4 levels of isolation

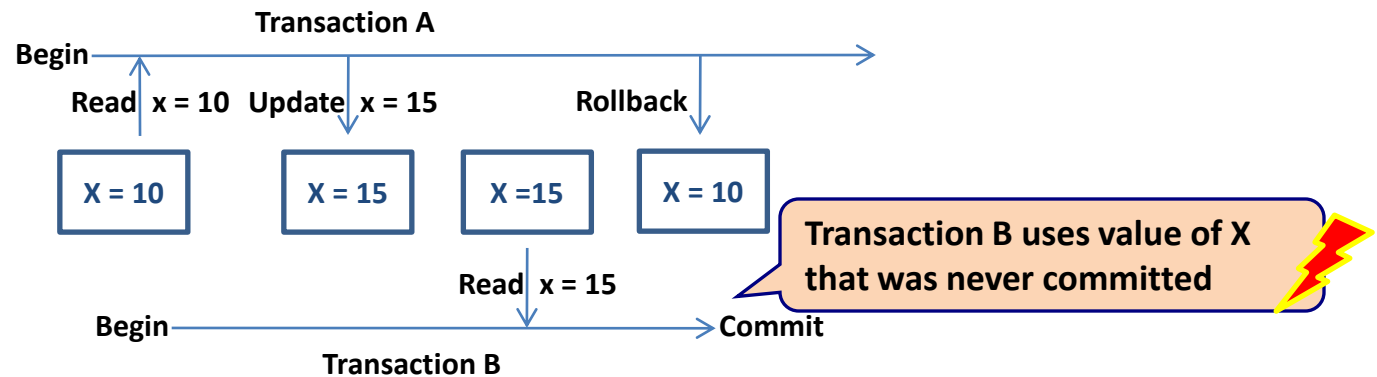
- TransactionReadUncommitted
- TransactionReadCommitted
- TransactionRepeatableRead
- TransactionSerializable



- 3 transaction problems

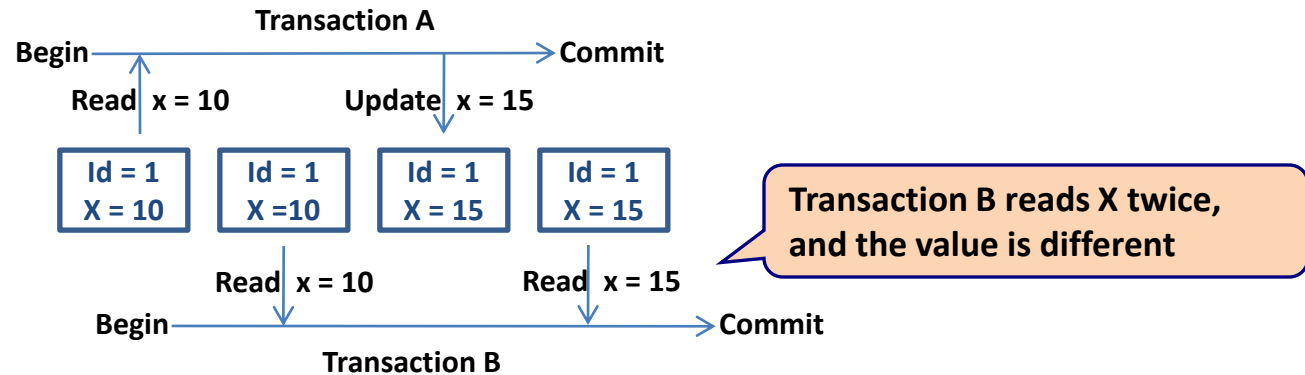
- Dirty read
- Non repeatable read
- Phantom read

Dirty Read



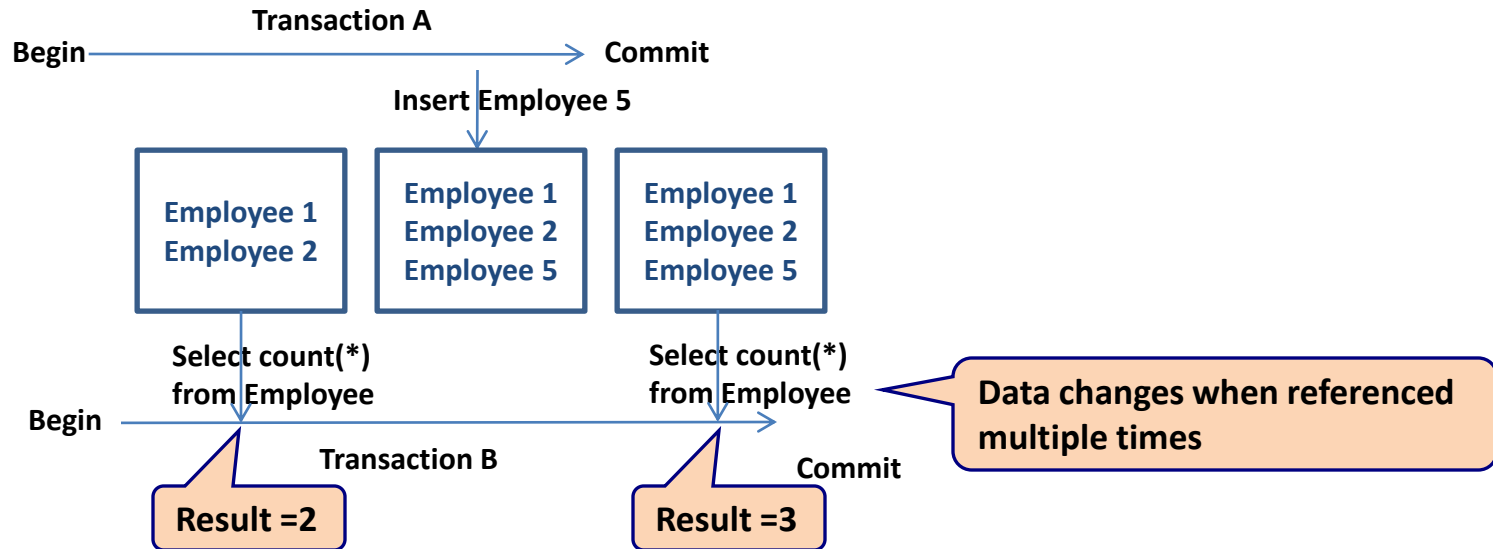
- Transactions A read $X = 10$
- Transaction A first increments X by 5 setting $X = 15$
- Transaction B read $X = 15$
- Transaction A does a rollback, so $X = 10$
- Transaction B uses the wrong value of X

Non Repeatable Read



- Transactions A and B read X = 10
- Transaction A first increments X by 5 setting X = 15
- Transaction B read X=15

Phantom Read



Isolation levels

Isolation	Dirty read	Non repeatable read	Phantom read
TransactionReadUncommitted	✓	✓	✓
TransactionReadCommitted		✓	✓
TransactionRepeatableRead			✓
TransactionSerializable			

- TransactionReadUncommitted
 - Violates the ACID properties
 - Not supported by many database vendors (Oracle)
 - Do not use this level of isolation in a multithreaded system
- TransactionReadcommitted
 - Default for most databases
- TransactionRepeatableRead
- TransactionSerializable
 - Highest level of isolation, lowest level of performance

JDBC transaction

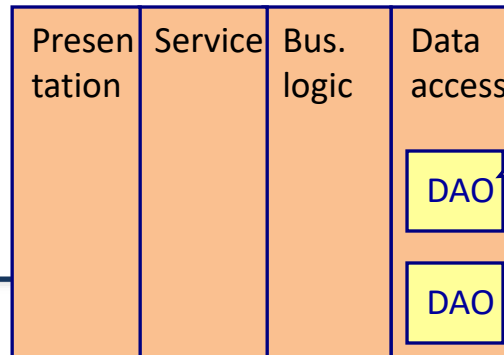
```
public void update(Employee employee) {
    Connection conn = null;
    PreparedStatement prepareUpdateEmployee = null;
    try {
        conn = getConnection();
        conn.setAutoCommit(false);
        prepareUpdateEmployee = conn.prepareStatement("UPDATE Employee SET
            firstname= ?, lastname= ? WHERE employeenumber=?");
        prepareUpdateEmployee.setString(1, employee.getFirstName());
        prepareUpdateEmployee.setString(2, employee.getLastName());
        prepareUpdateEmployee.setLong(3, employee.getEmployeeNumber());

        int updateresult = prepareUpdateEmployee.executeUpdate();
        conn.commit();
    } catch (SQLException e) {
        conn.rollback();
        System.out.println("SQLException in EmployeeDAO update() :" + e);
    } finally {
        try {
            prepareUpdateEmployee.close();
            closeConnection(conn);
        } catch (SQLException e1) {
            // no action needed
        }
    }
}
```

Commit transaction

Start transaction

Rollback transaction



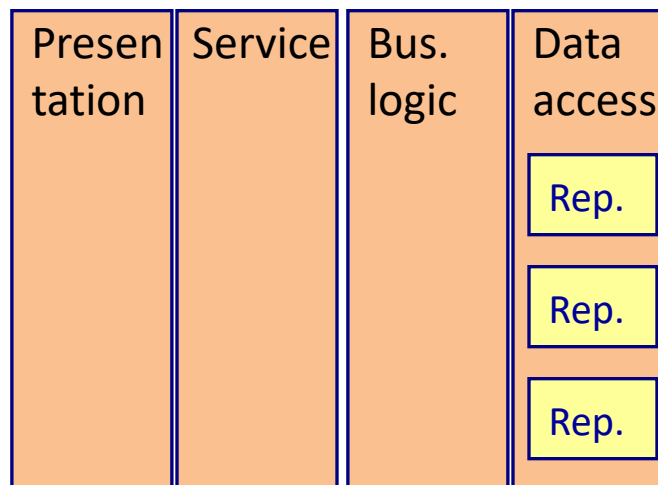
Transaction demarcation in the DAO classes

Spring-JPA transaction

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {  
}
```

Every save runs in its own transaction

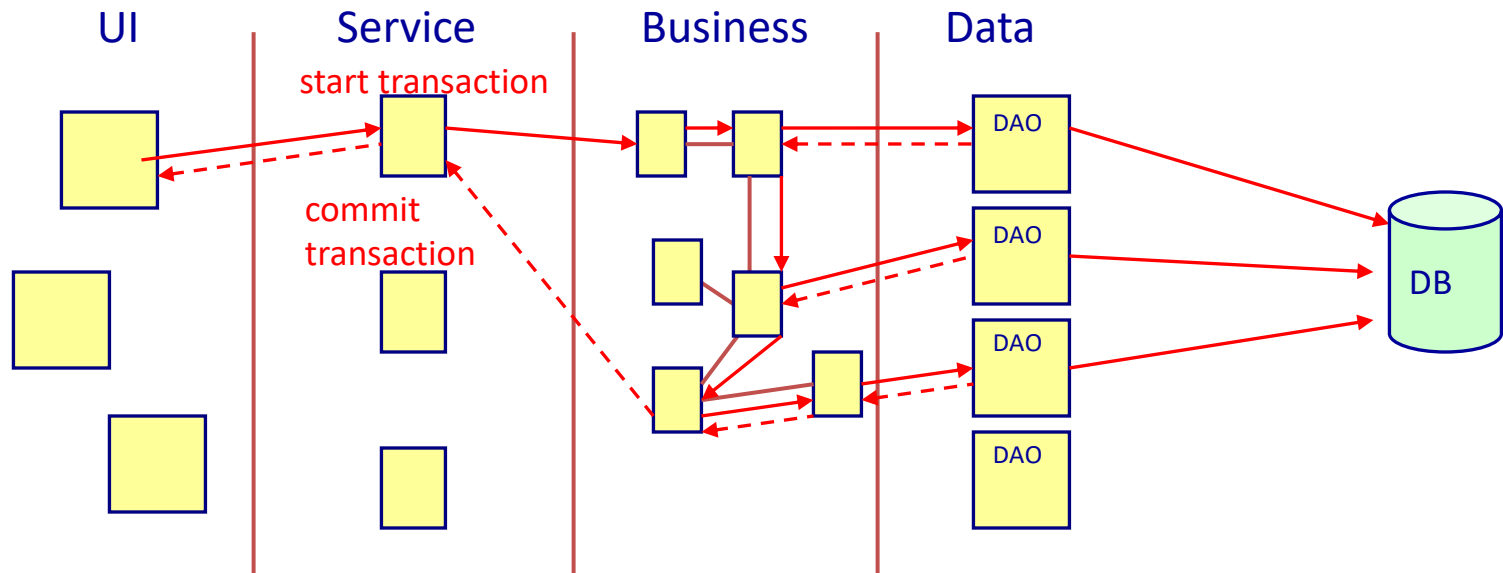
```
@Override  
public void run(String... args) throws Exception {  
    customerrepository.save(new Customer("Jack", "Bauer", "jack@acme.com"));  
    customerrepository.save(new Customer("Chloe", "O'Brian", "chloe@acme.com"));  
}
```



Transaction demarcation in the Repository classes

Typical transaction demarcation

- Transaction demarcation is typical at the level of the service classes
 - Multiple DAOs can be involved in one transaction
- Spring allows us to perform transaction demarcation for service level methods



Spring transaction support

- Spring is not a transaction manager
 - We still need a transaction manager
 - JDBC transaction manager
 - Hibernate transaction manager
 - XA transaction manager (JTS)
- Spring provides an abstraction for transaction management
 - Spring talks to the underlying transaction manager

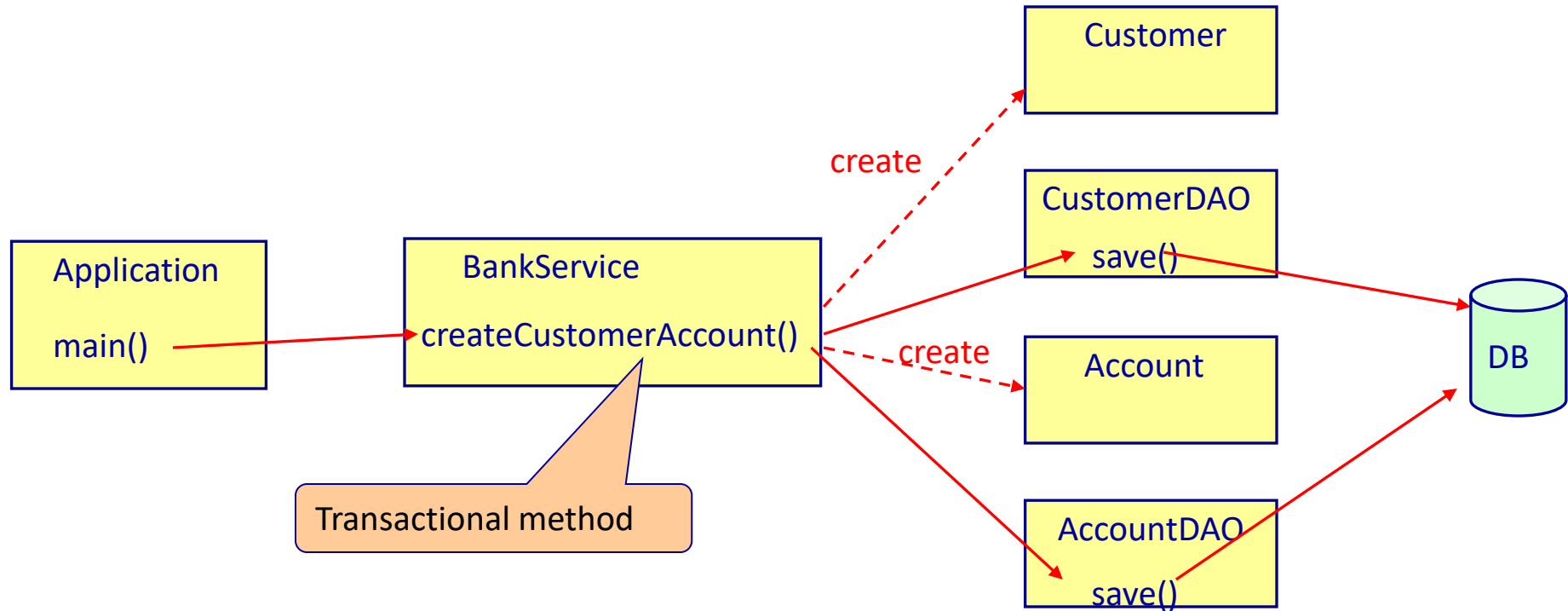
CONFIGURING TRANSACTIONS IN SPRING

Transactions in Spring

```
public class ...{  
  
    @Transactional  
    public void transactionalMethod();  
}  
}
```

All methods annotated with
@Transactional are transactional

Transaction Example



Transaction Example

```
@Service
public class BankingService {
    @Autowired
    private CustomerRepository customerRepository;
    @Autowired
    private AccountRepository accountRepository;
```

Service method executes within a transaction

```
@Transactional
public void createCustomerAccount(int customerid, String customerName, int
    accountnumber, double balance, boolean throwException) throws Exception {
    Customer customer = new Customer(customerid, customerName);
    customerRepository.save(customer);
    if(throwException) {
        throw new RuntimeException();
    }
    Account account = new Account(accountnumber, balance);
    accountRepository.save(account);
}
```

Transaction Example

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {  
}
```

```
@Entity  
public class Customer {  
    @Id  
    private int id;  
    private String name;  
    ...  
}
```

```
public interface AccountRepository extends JpaRepository<Account, Integer> {  
}
```

```
@Entity  
public class Account {  
    @Id  
    private int accountnumber;  
    private double balance;  
    ...  
}
```

Transaction Example

```
@SpringBootApplication
@EnableJpaRepositories("repositories")
@EntityScan("domain")
@ComponentScan("service")
public class Application implements CommandLineRunner{

    @Autowired
    BankService bankService;

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        bankService.createCustomerAndAccount(12, "Jack Bauer", "1223",false);
        bankService.createCustomerAndAccount(14, "Frank Brown", "1248",true);
    }
}
```

application.properties

```
spring.datasource.url=jdbc:hsqldb:hsql://localhost/trainingdb
spring.datasource.username=SA
spring.datasource.password=
spring.datasource.driver-class-name=org.hsqldb.jdbcDriver

spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.HSQLDialect
```

Transaction Example

With transaction

```
@Transactional
public void createCustomerAccount(int customerid, String customerName, int
accountnumber, double balance, boolean throwException) throws Exception {
    Customer customer = new Customer(customerid, customerName);
    customerRepository.save(customer);
    if(throwException) {
        throw new RuntimeException();
    }
    Account account = new Account(accountnumber, balance);
    accountRepository.save(account);
}
```

ID	NAME
120	Frank Brown

ACCOUNTNUMBER	BALANCE
312	0

Without transaction

```
public void createCustomerAccount(int customerid, String customerName, int
accountnumber, double balance, boolean throwException) throws Exception {
    Customer customer = new Customer(customerid, customerName);
    customerRepository.save(customer);
    if(throwException) {
        throw new RuntimeException();
    }
    Account account = new Account(accountnumber, balance);
    accountRepository.save(account);
}
```

ID	NAME
120	Frank Brown
121	John Doe

ACCOUNTNUMBER	BALANCE
312	0

Rollback with checked exceptions

- The transaction manager by default only does a rollback for runtime exceptions.
- If you want to rollback for checked exceptions, you have to explicitly specify this.

Checked exception rollback

```
public class BankingService implements IBankingService{
    private CustomerDAO customerDao;
    private AccountDAO accountDao;

    @Transactional(rollbackFor = {DAOException.class})
    public void createCustomerAccount(String customerName, int accountnumber) throws Exception{
        Customer customer= new Customer(customerName);
        customerDao.save(customer);
        Account account = new Account(accountnumber);
        accountDao.save(account);
    }
    public void setCustomerDao(CustomerDAO customerDao) {
        this.customerDao = customerDao;
    }
    public void setAccountDao(AccountDAO accountDao) {
        this.accountDao = accountDao;
    }
}
```

Rollback for a runtime exception and a DAOException

Set Propagation and Isolation with Spring

Set propagation to
REQUIRES_NEW

Set isolation to
REPEATABLE_READ

```
public class BankingService implements IBankingService{
    private CustomerDAO customerDao;
    private AccountDAO accountDao;

    @Transactional(propagation=Propagation.REQUIRES_NEW, isolation=Isolation.REPEATABLE_READ)
    public void createCustomerAccount(String customerName, int accountnumber) throws Exception{
        Customer customer= new Customer(customerName);
        customerDao.save(customer);
        Account account = new Account(accountnumber);
        accountDao.save(account);
    }

    public void setCustomerDao(CustomerDAO customerDao) {
        this.customerDao = customerDao;
    }

    public void setAccountDao(AccountDAO accountDao) {
        this.accountDao = accountDao;
    }
}
```

Main point

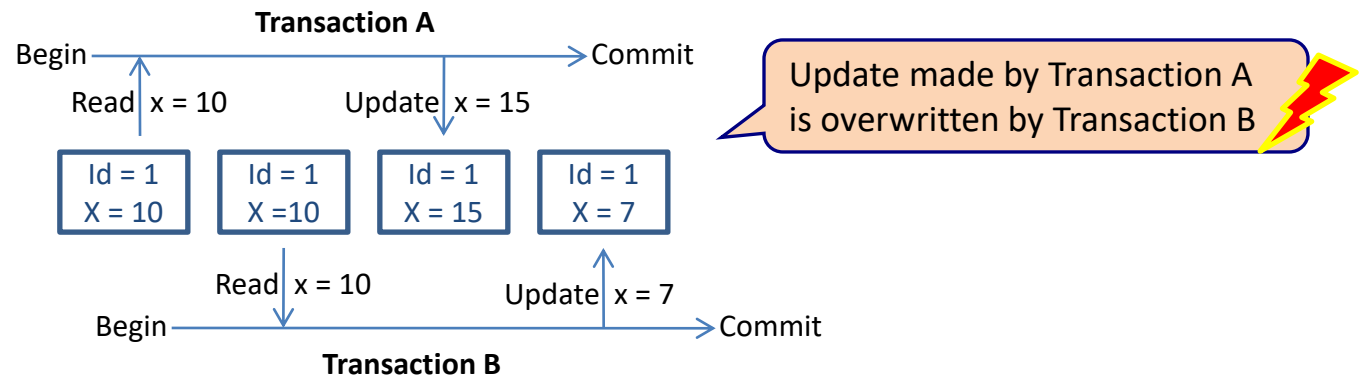
- The Spring framework makes it very easy to specify transactions on methods of Spring beans.

Science of Consciousness: Do Less and Accomplish More, the transactions are automatically applied in an additional AOP layer.

Transactions and Concurrency

OPTIMISTIC CONCURRENCY

Lost Update



- Transactions A and B read $X = 10$
 - Transaction A first increments X by 5 setting $X = 15$
 - Transaction B next decrements X by 3 and sets $X = 7$
- The update made by A is permanently lost, and neither A nor B is aware that it happened

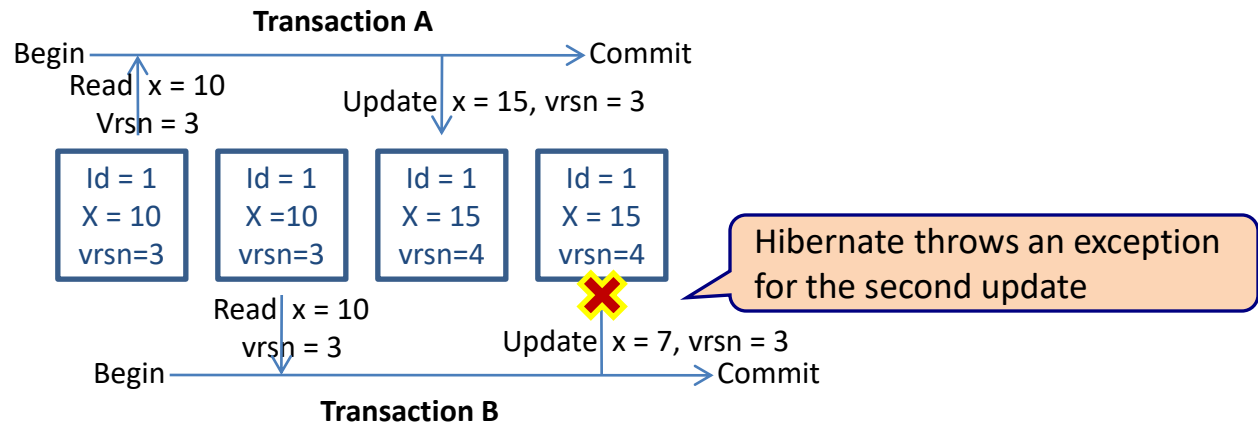
Optimistic Concurrency

- Optimistic concurrency assumes that lost update conflicts generally don't occur
 - But keeps versions# so that it knows when they do
 - Uses read committed transaction level
 - Guarantees best performance and scalability
 - The default way to deal with concurrency
- **First commit wins** instead of **last commit wins**
 - An exception is thrown if a conflict would occur



Versioning

- Additional version column to tracks updates



- Update Fails due to the version check
 - UPDATE table SET `x = 15, vrsn = 4` WHERE `id = 1 AND vrsn = 3`
 - If the version has changed, the update is not executed
 - Hibernate throws an exception when the update fails

Version Column

- The best way to enable versioning for a class is by using an additional version column
 - Should have no semantic value in the table

```
@Entity
public class Customer {
    @Id
    @GeneratedValue
    private int id;
    private String firstname;
    private String lastname;

    @Version
    private int version;

    ...
}
```

Use the @Version annotation to specify the version column

StaleObjectStateException

- When a version conflict occurs Hibernate throws a StaleObjectStateException
 - Catching this exception allows you to notify the user about the conflict
 - The user can then reload the data and apply their updates against the latest data

```
org.hibernate.StaleObjectStateException: Row was updated  
or deleted by another transaction (or unsaved-value  
mapping was incorrect): [optimistic.nocolumn.Customer#1]  
...
```

First or Last commit wins

```
public class Application1 {  
    public static void main(String[] args) throws InterruptedException {  
        SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();  
        Session session = sessionFactory.openSession();  
        Transaction tx = session.beginTransaction();  
        Car car = (Car) session.get(Car.class, "SDA231");  
        Thread.sleep(10000);  
        car.setBrand("Audi");  
        tx.commit();  
        session.close();  
    }  
}
```

```
public class Application2 {  
    public static void main(String[] args) throws InterruptedException {  
        SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();  
        Session session = sessionFactory.openSession();  
        Transaction tx = session.beginTransaction();  
        Car car = (Car) session.get(Car.class, "SDA231");  
        Thread.sleep(20000);  
        car.setBrand("Skoda");  
        tx.commit();  
        session.close();  
    }  
}
```

Last commit wins

```
@Entity
public class Car {
    @Id
    private String licencenumber;
    private String brand;
    private String year;
    ...
}
```

Initial database state

LICENCENUMBER	BRAND	YEAR
SDA231	BMW	2008

Run application1

Change to Audi

Run application2

Change to Skoda

No exception thrown

Final database state

LICENCENUMBER	BRAND	YEAR
SDA231	Skoda	2008

First commit wins

```
public class Car {  
    @Id  
    private String licencenumber;  
    private String brand;  
    private String year;  
    @Version  
    private int version;  
    ...  
}
```

Initial database state

LICENCENUMBER	BRAND	VERSION	YEAR
SDA231	BMW	0	2008

Run application1

Change to Audi

Run application2

Change to Skoda

StaleObjectStateException thrown

Final database state

LICENCENUMBER	BRAND	VERSION	YEAR
SDA231	Audi	1	2008

Connecting the parts of knowledge with the wholeness of knowledge

1. When defining transactions boundaries in your application it is important to define the correct transaction propagation
 2. The TransactionReadCommitted isolation level is the default level of most databases.
-

3. **Transcendental consciousness** is the foundation of all thoughts.
4. **Wholeness moving within itself:** In Unity Consciousness, we experience how both the silence at the basis of thought, and the most expressed thoughts and actions are nothing but the Self.

