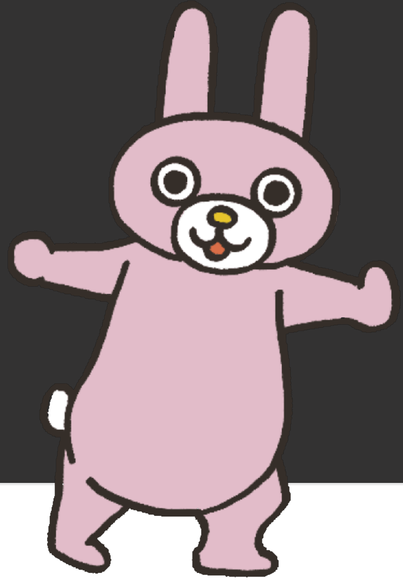
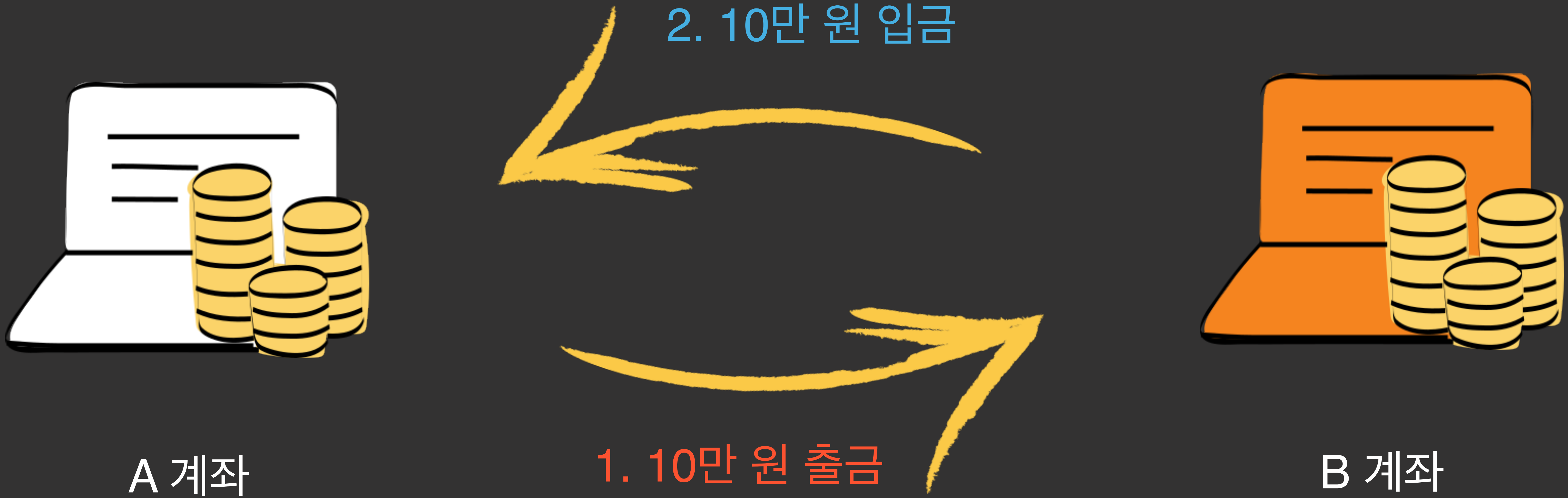
Spring @Transactional



발표자: 한지원

Transaction이 필요한 이유

계좌 이체



2. 10만 원 입금

1. 10만 원 출금

A 계좌

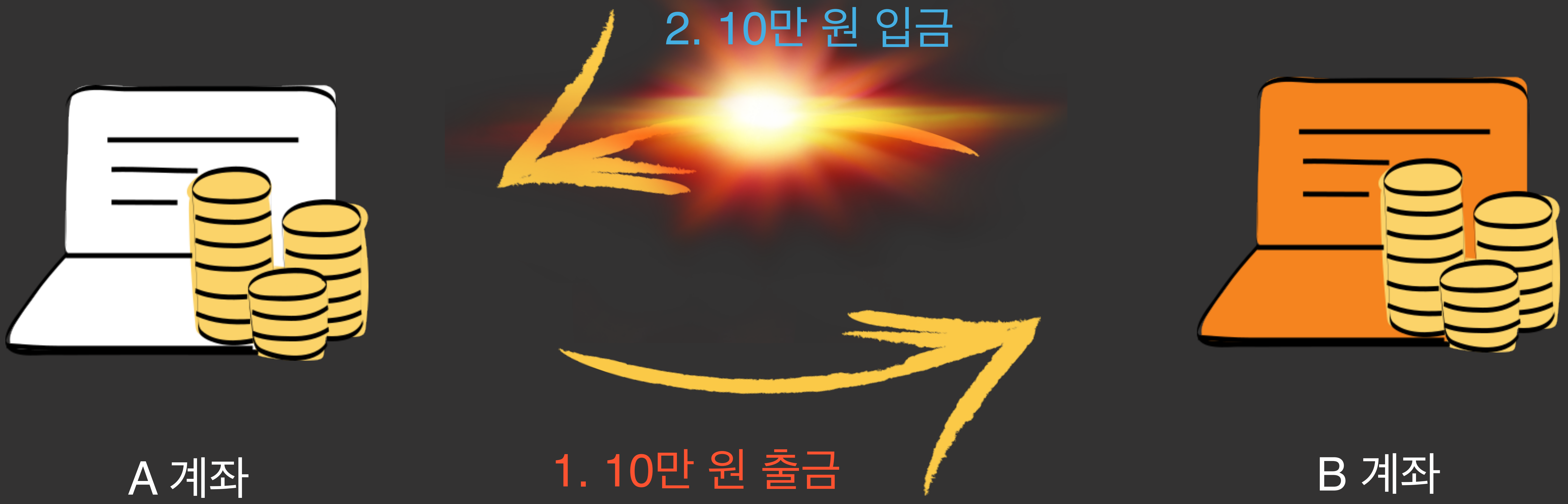
+ 10만 원

B 계좌

- 10만 원

Transaction이 필요한 이유

계좌 이체



2. 10만 원 입금

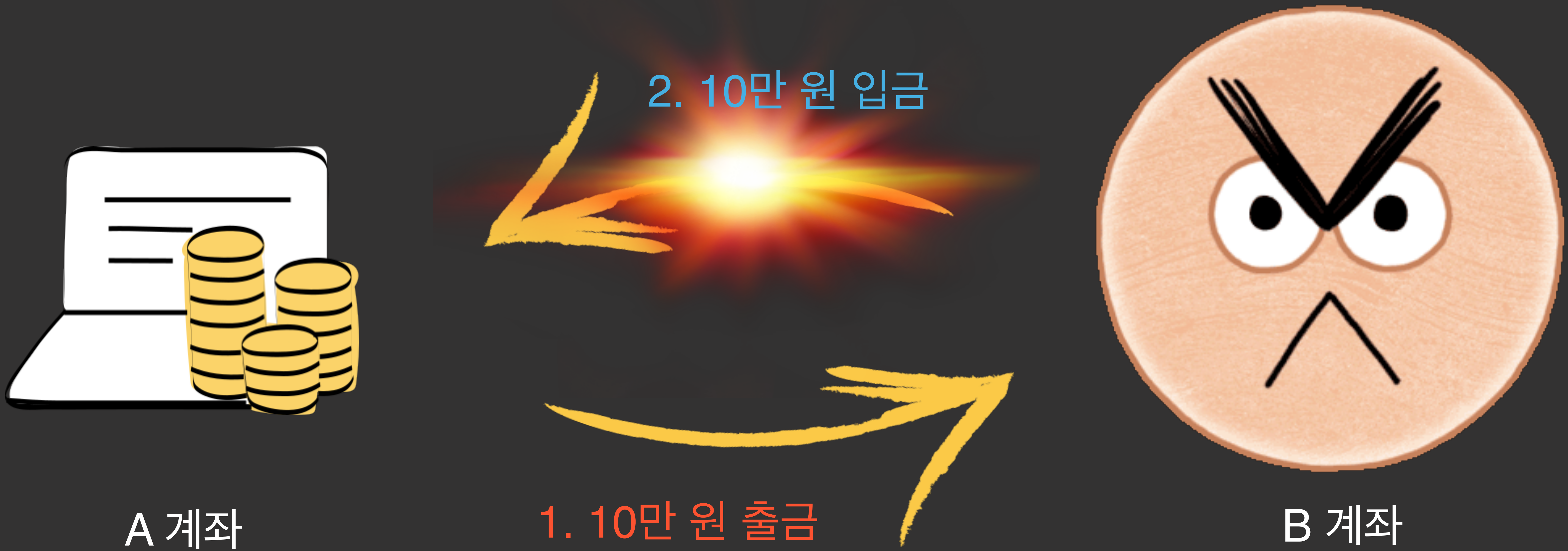
A 계좌

1. 10만 원 출금

B 계좌

Transaction이 필요한 이유

계좌 이체



2. 10만 원 입금

A 계좌

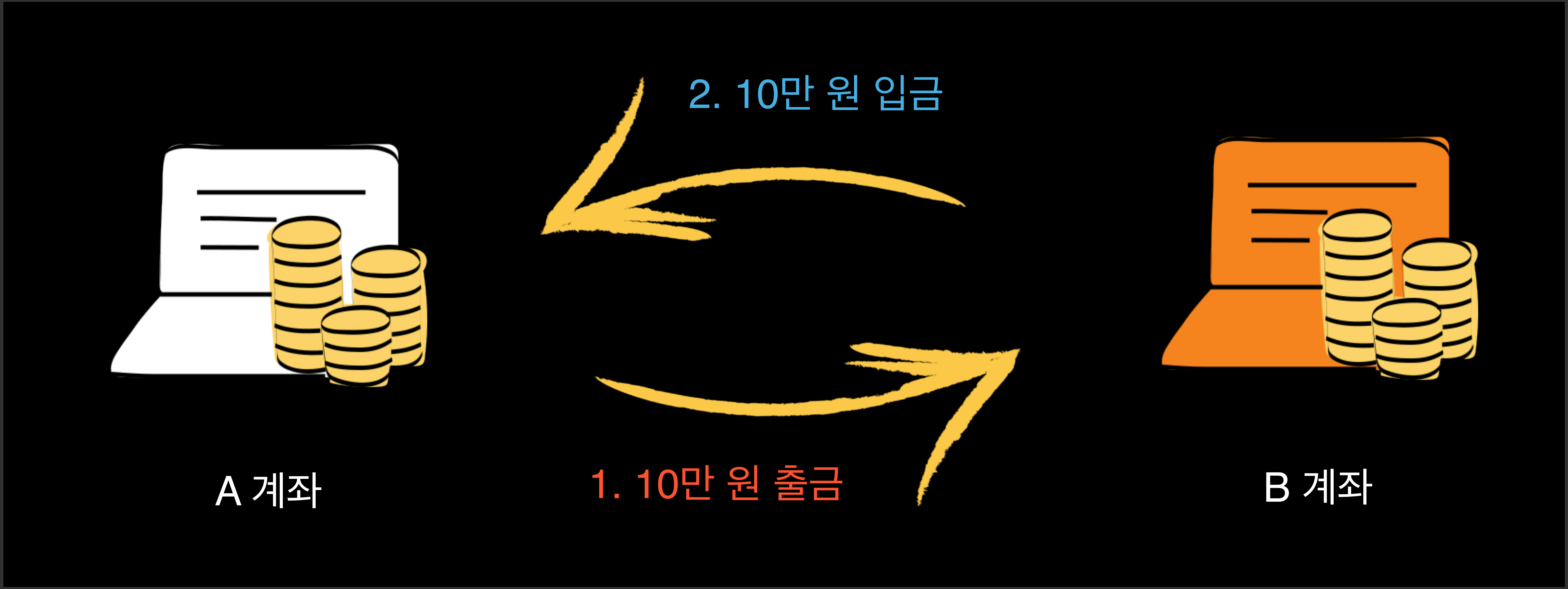
1. 10만 원 출금

B 계좌

- 10만 원

Transaction이 필요한 이유

계좌 이체(하나의 작업 단위)



How to Use Transaction

Transaction(하나의 작업 단위)

1. auto commit = false 로 설정

2. commit();

3. rollback();

How to Use Database in Spring

저수준

고수준

1. JDBC 직접 사용

2. PlatformTransactionManager 사용

3. TransactionTemplate 사용

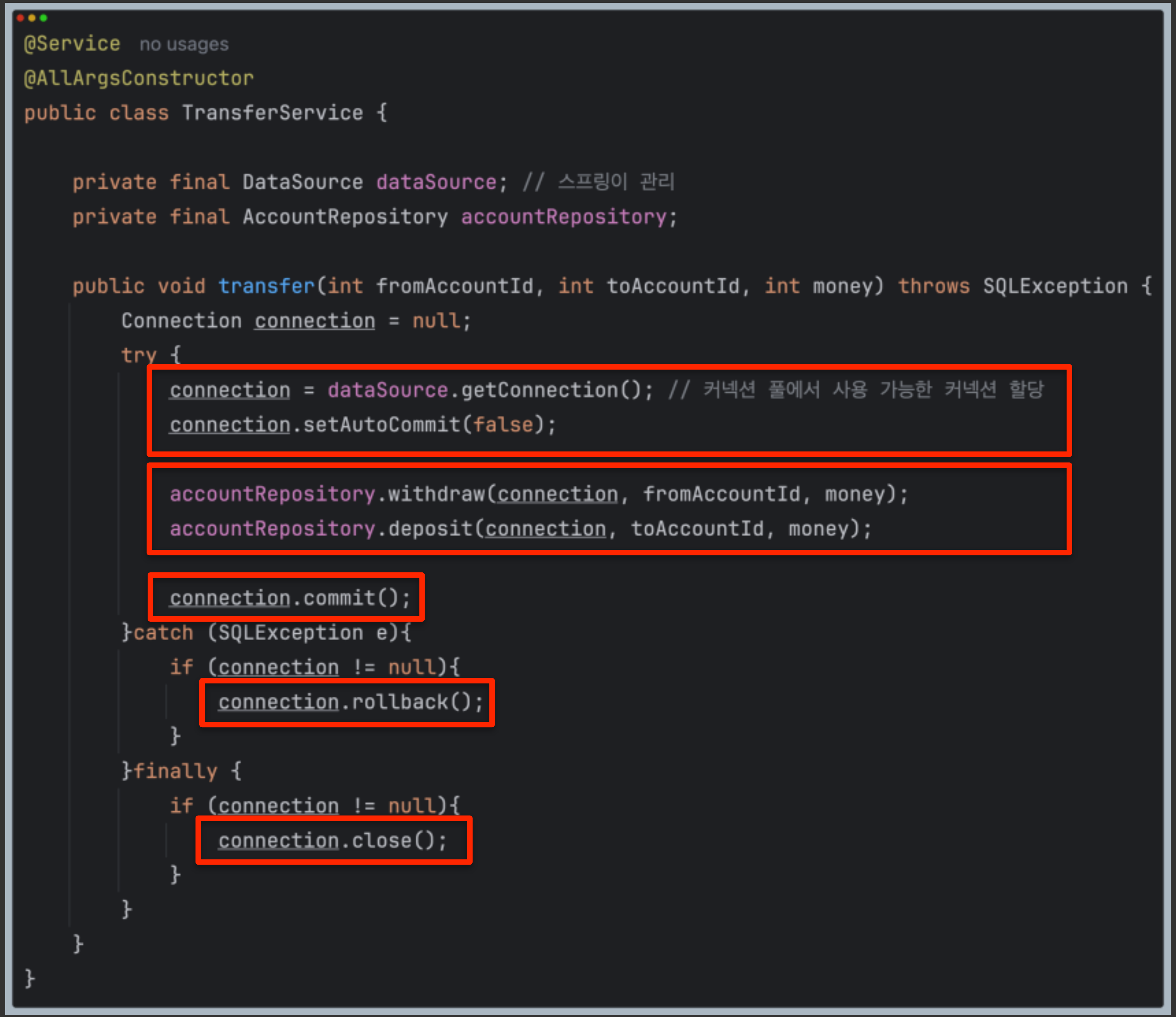
4. @Transactional 애노테이션 사용

How to Use Transaction in Spring

Programmatic vs Declarative

Programmatic Transaction Management

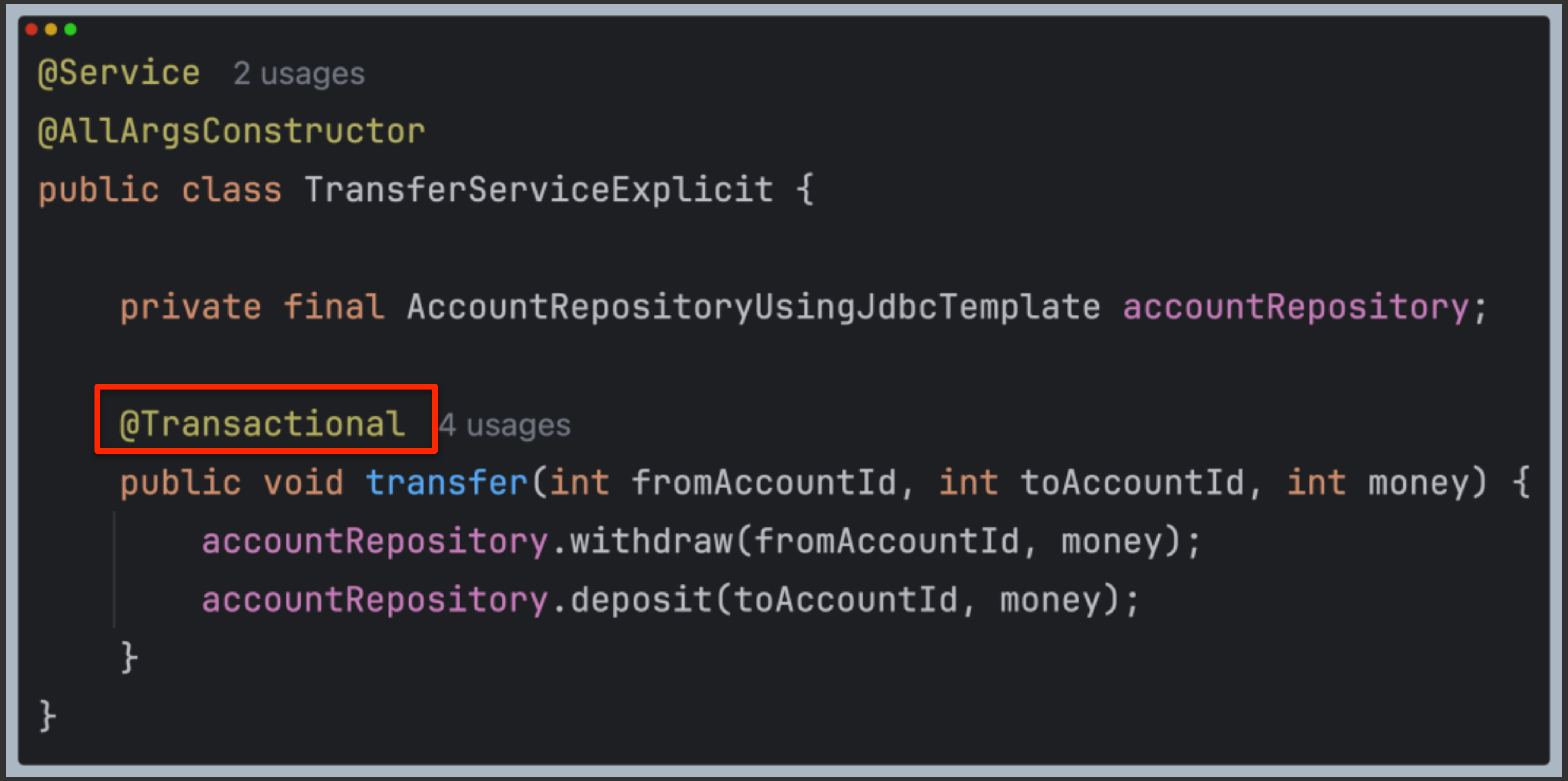
JDBC 직접 사용



```java
@Service  no usages
@AllArgsConstructor
public class TransferService {

    private final DataSource dataSource; // 스프링이 관리
    private final AccountRepository accountRepository;

    public void transfer(int fromAccountId, int toAccountId, int money) throws SQLException {
        Connection connection = null;
        try {
            connection = dataSource.getConnection(); // 커넥션 풀에서 사용 가능한 커넥션 할당
            connection.setAutoCommit(false);

            accountRepository.withdraw(connection, fromAccountId, money);
            accountRepository.deposit(connection, toAccountId, money);

            connection.commit();
        }catch (SQLException e){
            if (connection != null){
                connection.rollback();
            }
        }finally {
            if (connection != null){
                connection.close();
            }
        }
    }
}
```

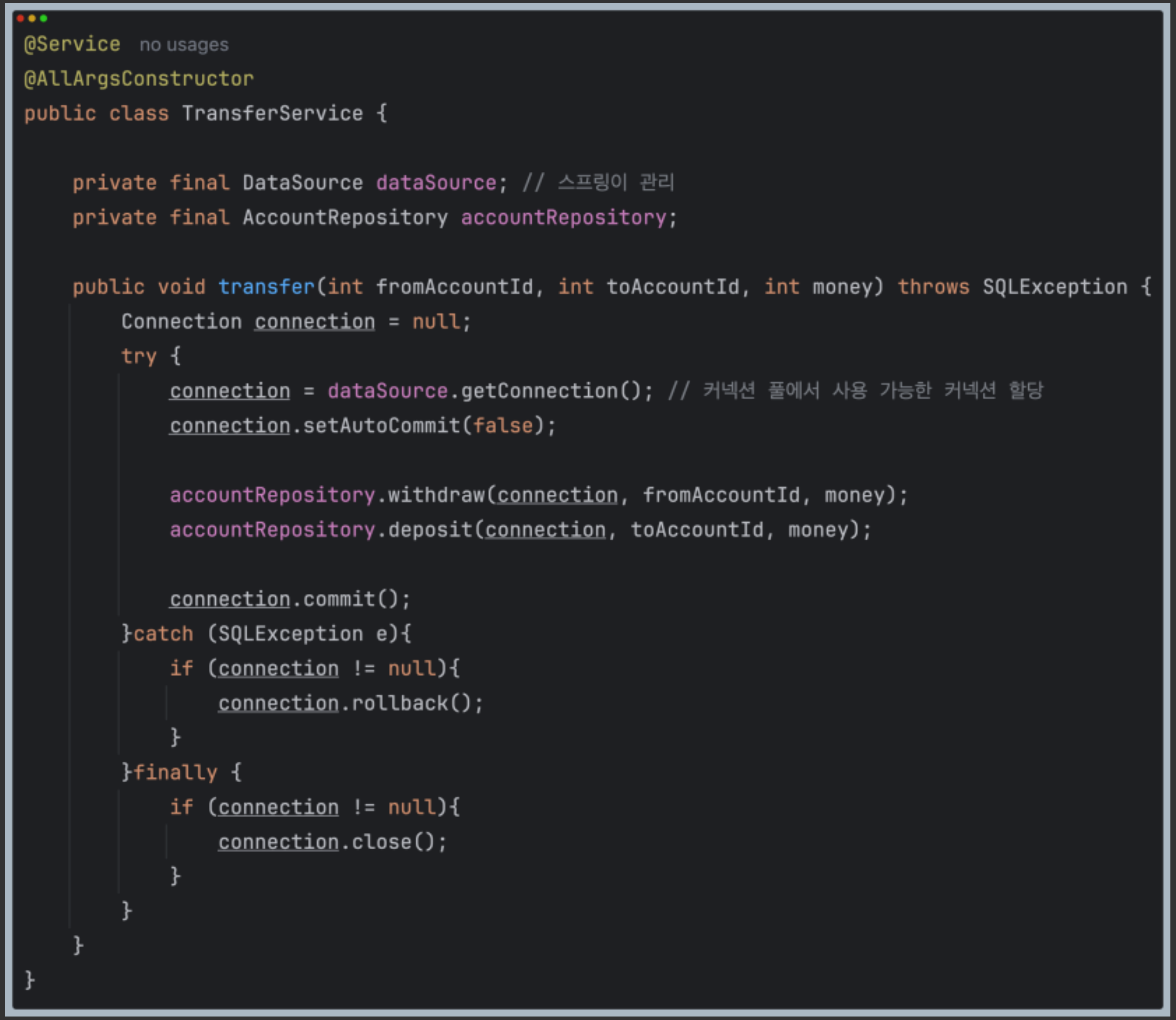
Declarative Transaction Management

애노테이션 사용

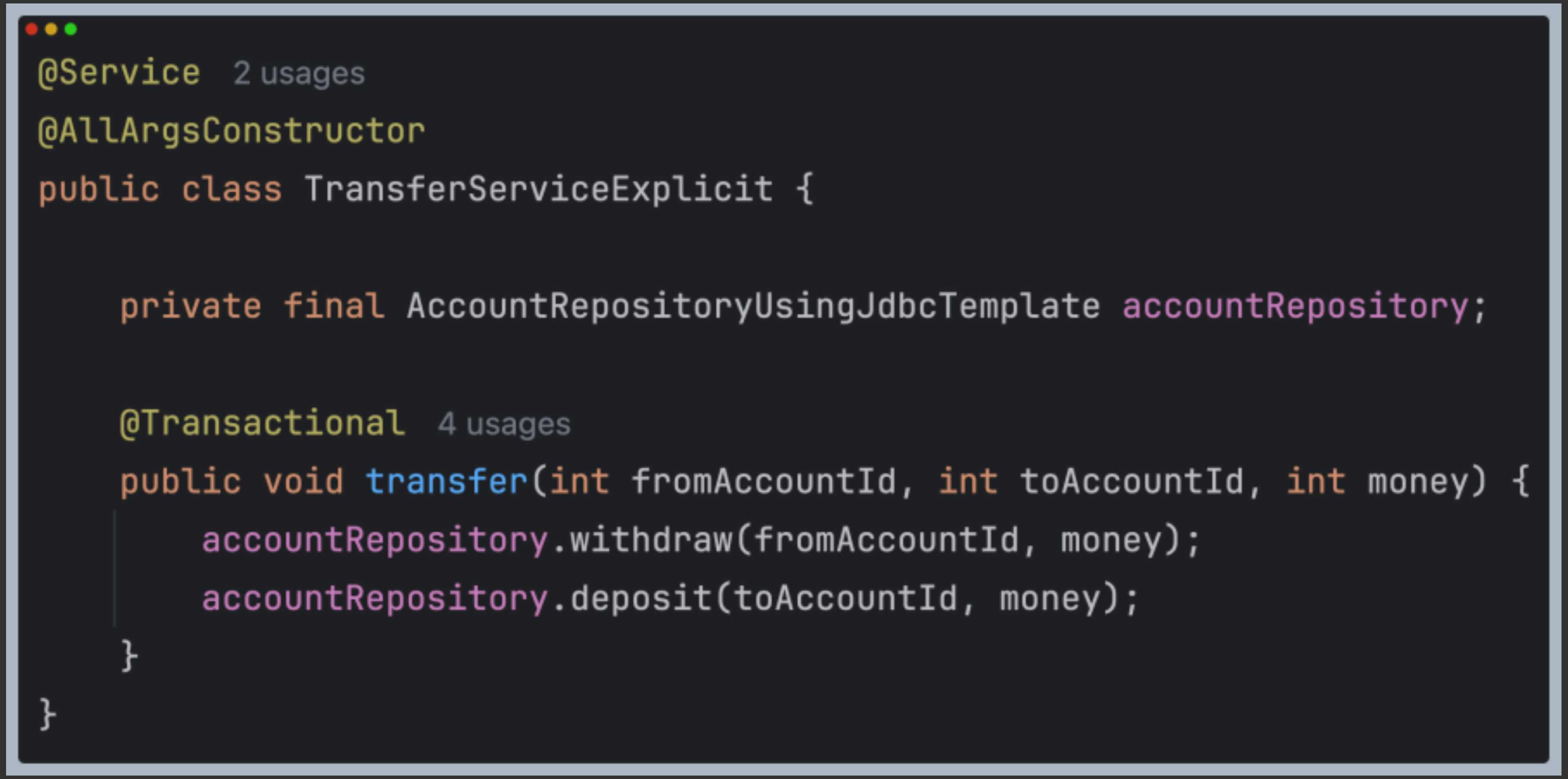


```
@Service  2 usages
@AllArgsConstructor
public class TransferServiceExplicit {

    private final AccountRepositoryUsingJdbcTemplate accountRepository;

    @Transactional  4 usages
    public void transfer(int fromAccountId, int toAccountId, int money) {
        accountRepository.withdraw(fromAccountId, money);
        accountRepository.deposit(toAccountId, money);
    }
}
```

Programmatic vs Declarative



```java
@Service  no usages
@AllArgsConstructor
public class TransferService {

    private final DataSource dataSource; // 스프링이 관리
    private final AccountRepository accountRepository;

    public void transfer(int fromAccountId, int toAccountId, int money) throws SQLException {
        Connection connection = null;
        try {
            connection = dataSource.getConnection(); // 커넥션 풀에서 사용 가능한 커넥션 할당
            connection.setAutoCommit(false);

            accountRepository.withdraw(connection, fromAccountId, money);
            accountRepository.deposit(connection, toAccountId, money);

            connection.commit();
        }catch (SQLException e){
            if (connection != null){
                connection.rollback();
            }
        }finally {
            if (connection != null){
                connection.close();
            }
        }
    }
}
```

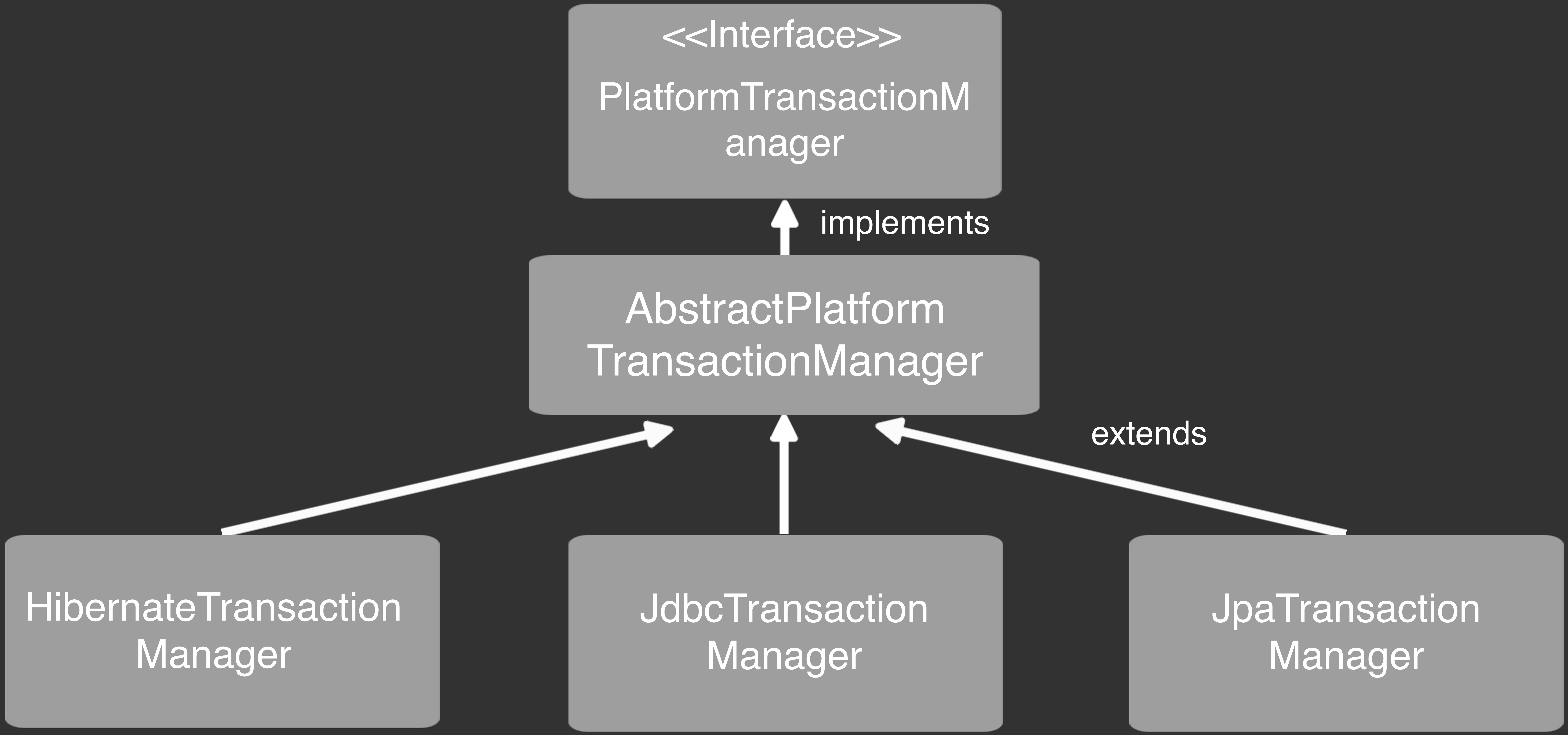


```java
@Service  2 usages
@AllArgsConstructor
public class TransferServiceExplicit {

    private final AccountRepositoryUsingJdbcTemplate accountRepository;

    @Transactional  4 usages
    public void transfer(int fromAccountId, int toAccountId, int money) {
        accountRepository.withdraw(fromAccountId, money);
        accountRepository.deposit(toAccountId, money);
    }
}
```

- 코드 가독성
- 유지보수 용이성
- 기술에 대한 종속성

@Transactional은 어떻게 작동할까?

Spring Framework Transaction Abstraction



<<Interface>>

PlatformTransactionManager

implements

AbstractPlatform TransactionManager

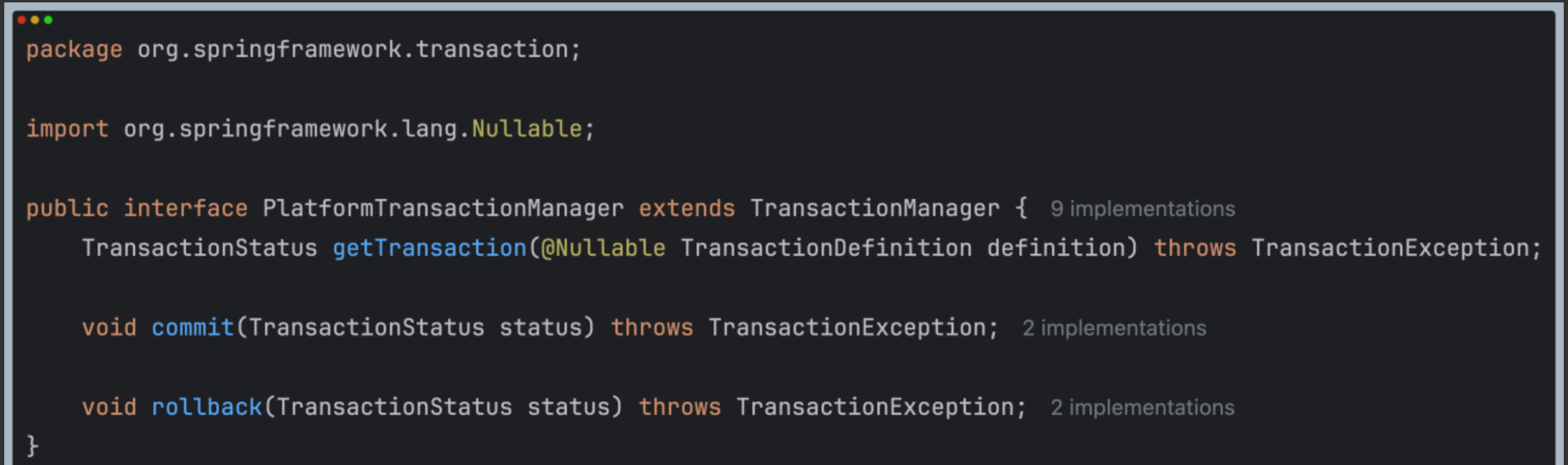
extends

HibernateTransaction Manager

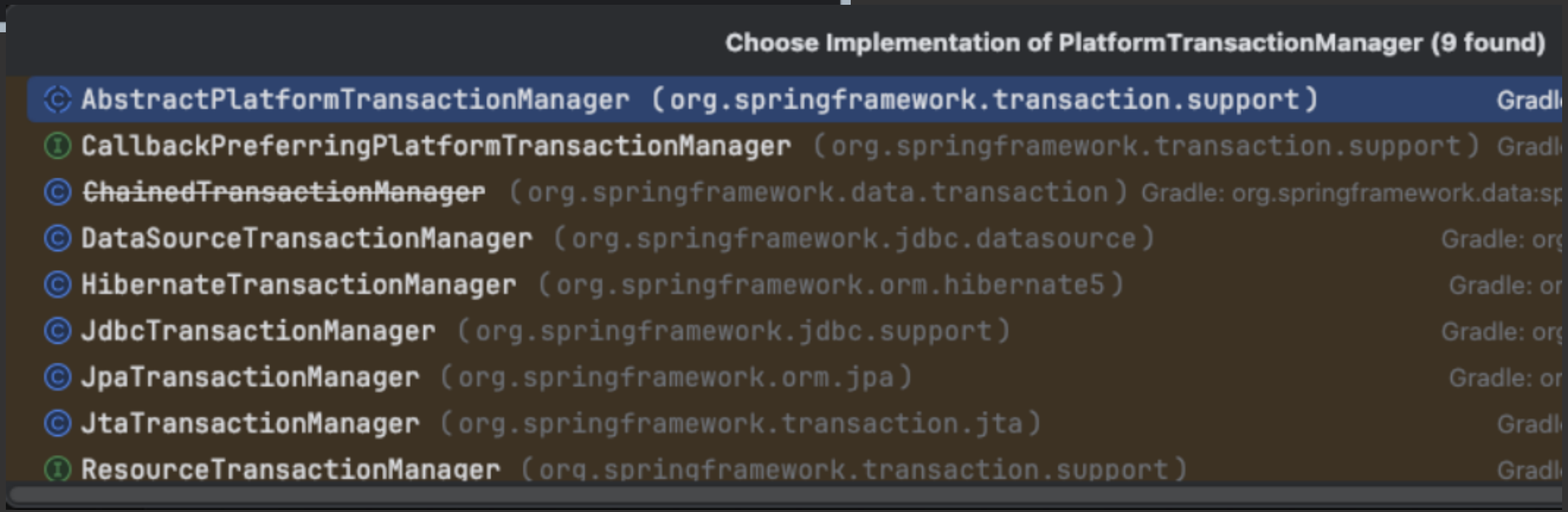
JdbcTransaction Manager

JpaTransaction Manager

Spring Framework Transaction Abstraction



```
package org.springframework.transaction;

import org.springframework.lang.Nullable;

public interface PlatformTransactionManager extends TransactionManager {  9 implementations
    TransactionStatus getTransaction(@Nullable TransactionDefinition definition) throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;  2 implementations

    void rollback(TransactionStatus status) throws TransactionException;  2 implementations
}
```



Choose Implementation of PlatformTransactionManager (9 found)

AbstractPlatformTransactionManager (org.springframework.transaction.support) Gradl

CallbackPreferringPlatformTransactionManager (org.springframework.transaction.support) Gradl

~~ChainedTransactionManager~~ (org.springframework.data.transaction) Gradle: org.springframework.data:sp

DataSourceTransactionManager (org.springframework.jdbc.datasource) Gradle: org

HibernateTransactionManager (org.springframework.orm.hibernate5) Gradle: or

JdbcTransactionManager (org.springframework.jdbc.support) Gradle: org

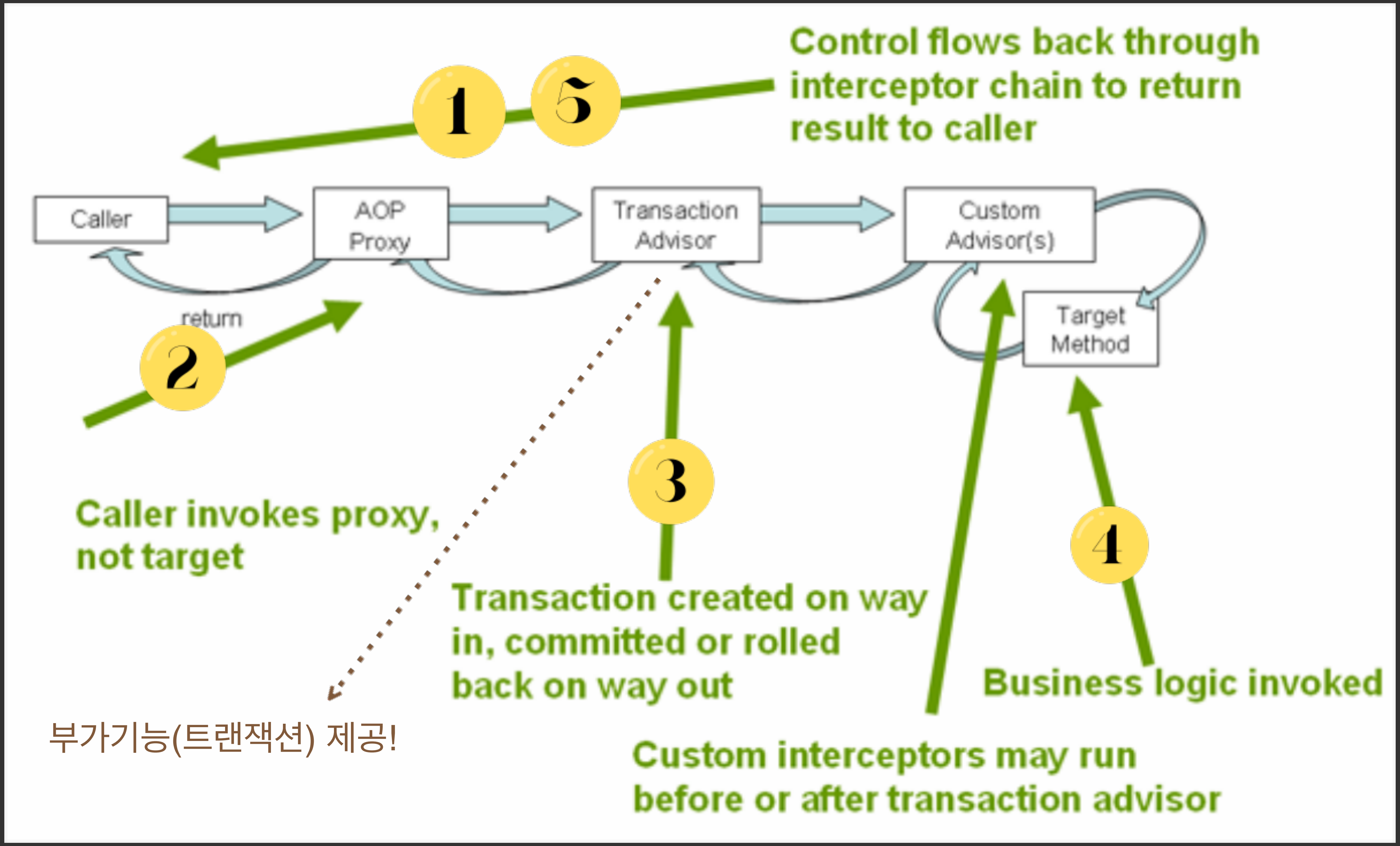
JpaTransactionManager (org.springframework.orm.jpa) Gradle: or

JtaTransactionManager (org.springframework.transaction.jta) Gradl

ResourceTransactionManager (org.springframework.transaction.support) Gradl

* 런타임 예외 발생시에만 롤백이 실행된다.

Spring AOP - Proxy



Control flows back through interceptor chain to return result to caller

Caller invokes proxy, not target

Transaction created on way in, committed or rolled back on way out

Business logic invoked

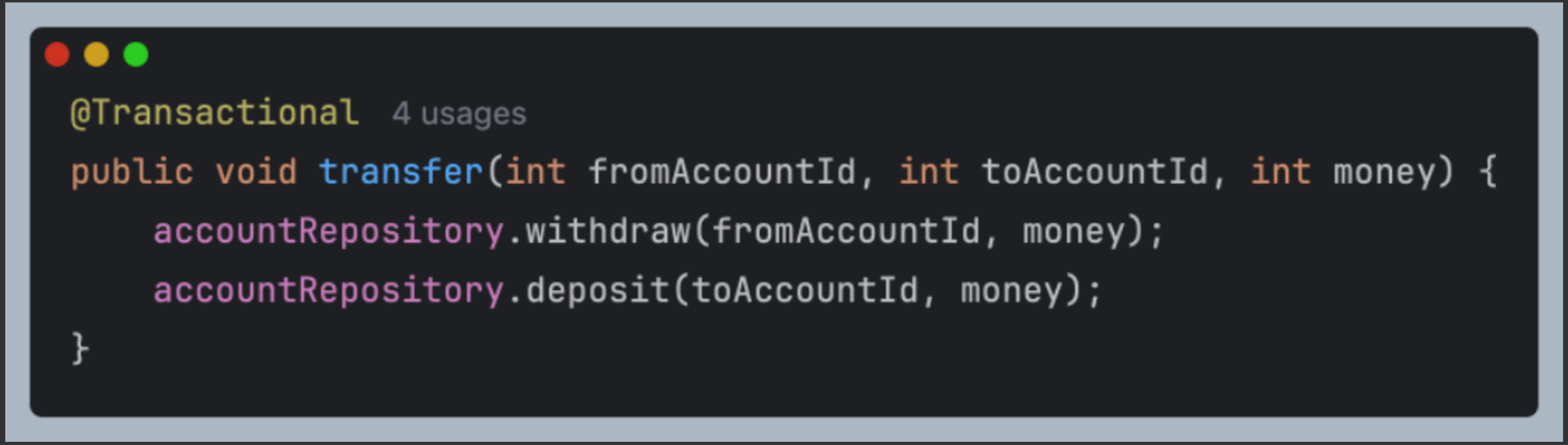
Custom interceptors may run before or after transaction advisor

부가기능(트랜잭션) 제공!

참고: 스프링 공식 문서

Spring AOP

서비스의 @Transactional 메서드 호출 시,

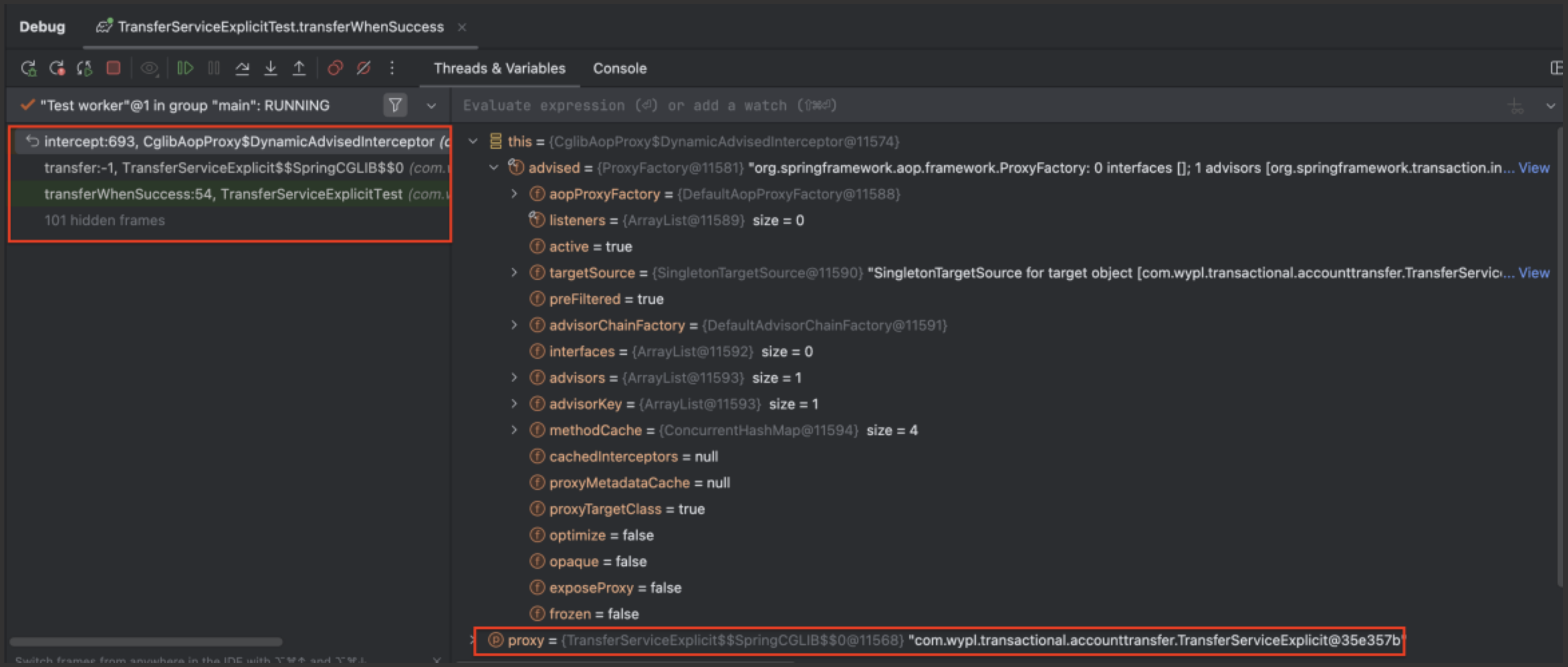


```java
@Transactional  4 usages
public void transfer(int fromAccountId, int toAccountId, int money) {
    accountRepository.withdraw(fromAccountId, money);
    accountRepository.deposit(toAccountId, money);
}
```

Spring AOP

1. PostProcessAfterInitialization

2. 프록시 객체로 변경



Debug
TransferServiceExplicitTest.transferWhenSuccess
Threads & Variables
Console
Evaluate expression (⏎) or add a watch (⇧⌘⏎)

```
"Test worker"@1 in group "main": RUNNING
intercept:693, CglibAopProxy$DynamicAdvisedInterceptor (c
transfer:-1, TransferServiceExplicit$$SpringCGLIB$$0 (com.
transferWhenSuccess:54, TransferServiceExplicitTest (com.
101 hidden frames
```

```
this = {CglibAopProxy$DynamicAdvisedInterceptor@11574}
  advised = {ProxyFactory@11581} "org.springframework.aop.framework.ProxyFactory: 0 interfaces []; 1 advisors [org.springframework.transaction.in... View
    aopProxyFactory = {DefaultAopProxyFactory@11588}
    listeners = {ArrayList@11589}  size = 0
    active = true
    targetSource = {SingletonTargetSource@11590} "SingletonTargetSource for target object [com.wypl.transactional.accounttransfer.TransferServic... View
    preFiltered = true
    advisorChainFactory = {DefaultAdvisorChainFactory@11591}
    interfaces = {ArrayList@11592}  size = 0
    advisors = {ArrayList@11593}  size = 1
    advisorKey = {ArrayList@11593}  size = 1
    methodCache = {ConcurrentHashMap@11594}  size = 4
    cachedInterceptors = null
    proxyMetadataCache = null
    proxyTargetClass = true
    optimize = false
    opaque = false
    exposeProxy = false
    frozen = false
  proxy = {TransferServiceExplicit$$SpringCGLIB$$0@11568} "com.wypl.transactional.accounttransfer.TransferServiceExplicit@35e357b"
```

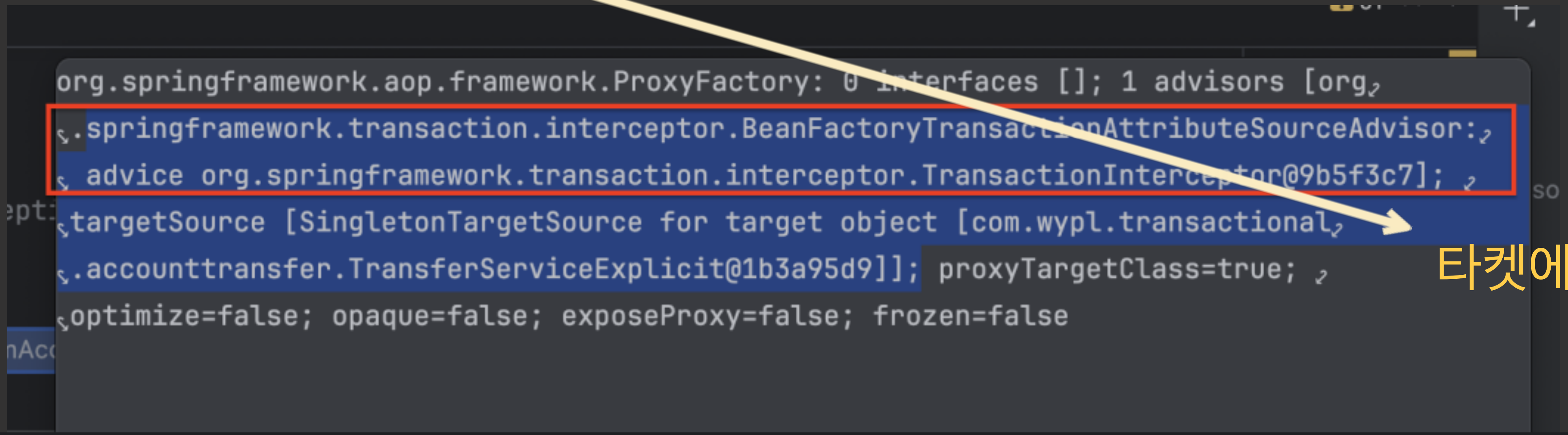
서비스의 @Transactional 메서드 호출 시, 프록시 객체 호출

Spring AOP

1. PostProcessAfterInitialization

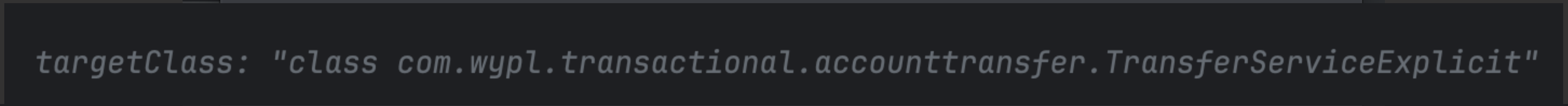
2. 프록시 객체로 변경

3. Advisors의 Advice로 등록된 TransactionInterceptor 호출

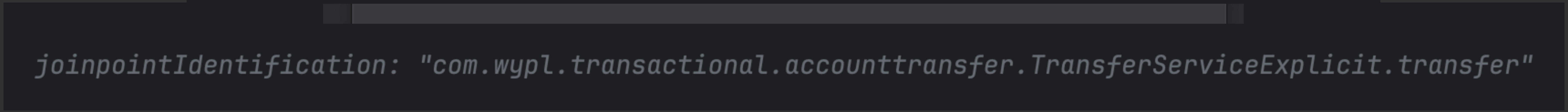


```
org.springframework.aop.framework.ProxyFactory: 0 interfaces []; 1 advisors [org.
.springframework.transaction.interceptor.BeanFactoryTransactionAttributeSourceAdvisor:
 advice org.springframework.transaction.interceptor.TransactionInterceptor@9b5f3c7];
targetSource [SingletonTargetSource for target object [com.wypl.transactional
.accounttransfer.TransferServiceExplicit@1b3a95d9]]; proxyTargetClass=true;
optimize=false; opaque=false; exposeProxy=false; frozen=false
```

타켓에 제공할 부가기능을 담고 있는 모듈!



```
targetClass: "class com.wypl.transactional.accounttransfer.TransferServiceExplicit"
```



```
joinpointIdentification: "com.wypl.transactional.accounttransfer.TransferServiceExplicit.transfer"
```

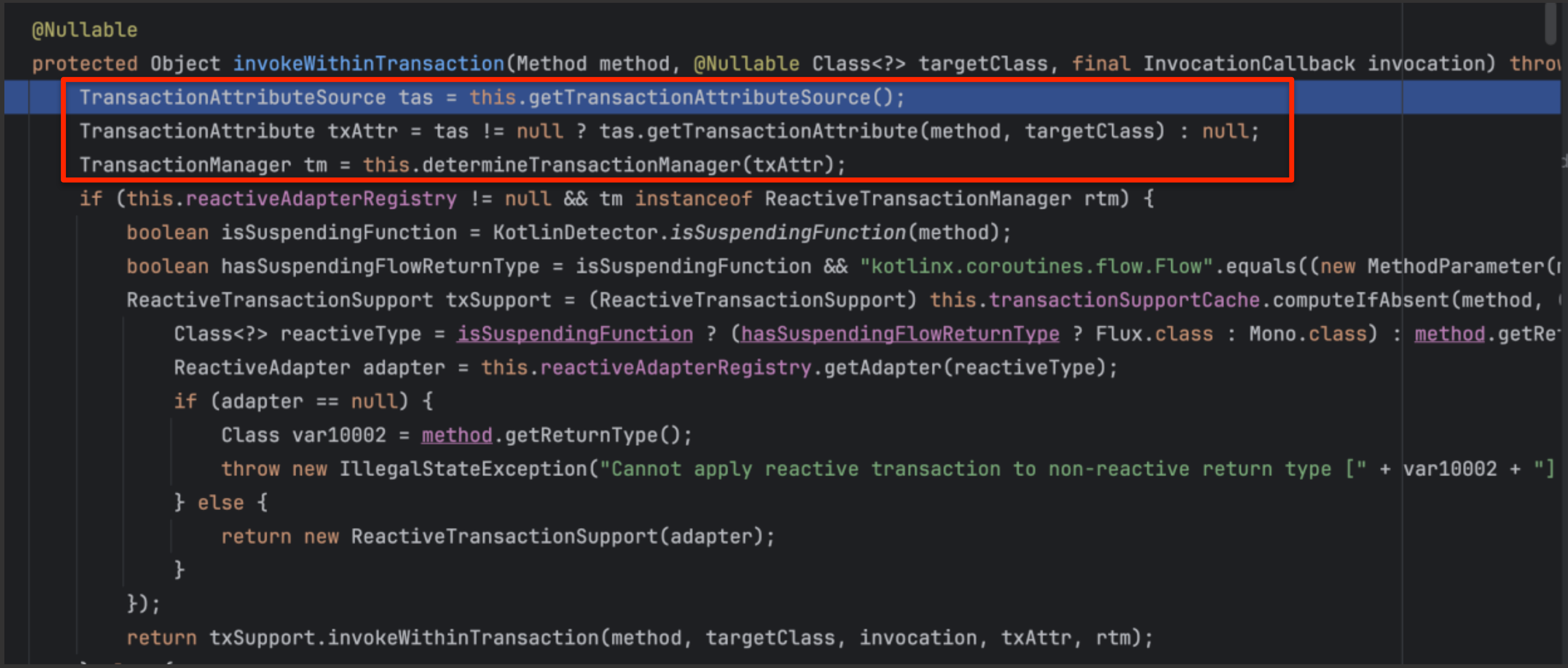
Spring AOP

1. PostProcessAfterInitialization

2. 프록시 객체로 변경

3. Advisors의 Advice로 등록된 TransactionInterceptor 호출

4. invokeWithInTransaction 호출



```java
@Nullable
protected Object invokeWithinTransaction(Method method, @Nullable Class<?> targetClass, final InvocationCallback invocation) thro
    TransactionAttributeSource tas = this.getTransactionAttributeSource();
    TransactionAttribute txAttr = tas != null ? tas.getTransactionAttribute(method, targetClass) : null;
    TransactionManager tm = this.determineTransactionManager(txAttr);
    if (this.reactiveAdapterRegistry != null && tm instanceof ReactiveTransactionManager rtm) {
        boolean isSuspendingFunction = KotlinDetector.isSuspendingFunction(method);
        boolean hasSuspendingFlowReturnType = isSuspendingFunction && "kotlinx.coroutines.flow.Flow".equals((new MethodParameter(
        ReactiveTransactionSupport txSupport = (ReactiveTransactionSupport) this.transactionSupportCache.computeIfAbsent(method,
            Class<?> reactiveType = isSuspendingFunction ? (hasSuspendingFlowReturnType ? Flux.class : Mono.class) : method.getRe
            ReactiveAdapter adapter = this.reactiveAdapterRegistry.getAdapter(reactiveType);
            if (adapter == null) {
                Class var10002 = method.getReturnType();
                throw new IllegalStateException("Cannot apply reactive transaction to non-reactive return type [" + var10002 + "]
            } else {
                return new ReactiveTransactionSupport(adapter);
            }
        });
        return txSupport.invokeWithinTransaction(method, targetClass, invocation, txAttr, rtm);
```

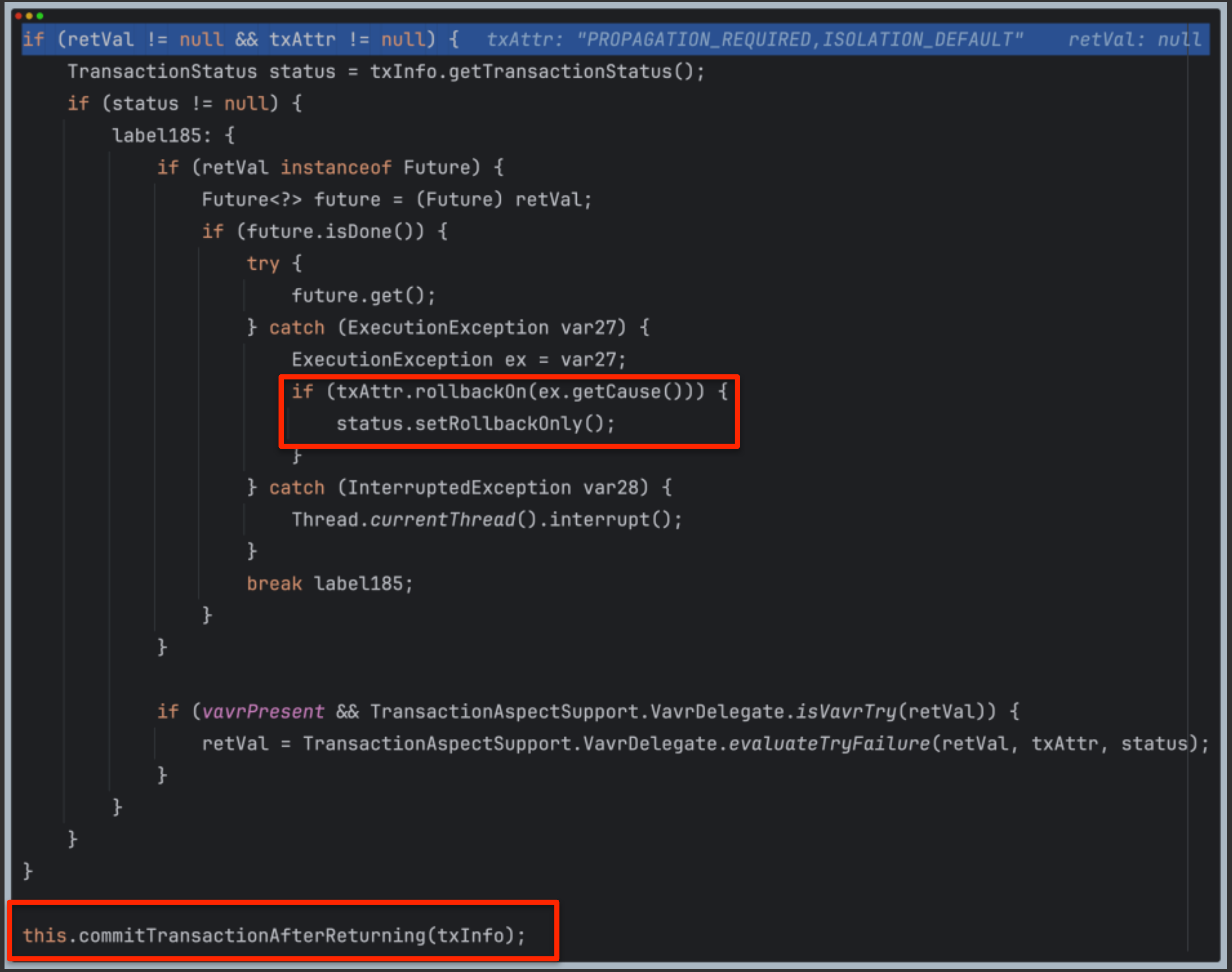
Spring AOP

1. PostProcessAfterInitialization

2. 프록시 객체로 변경

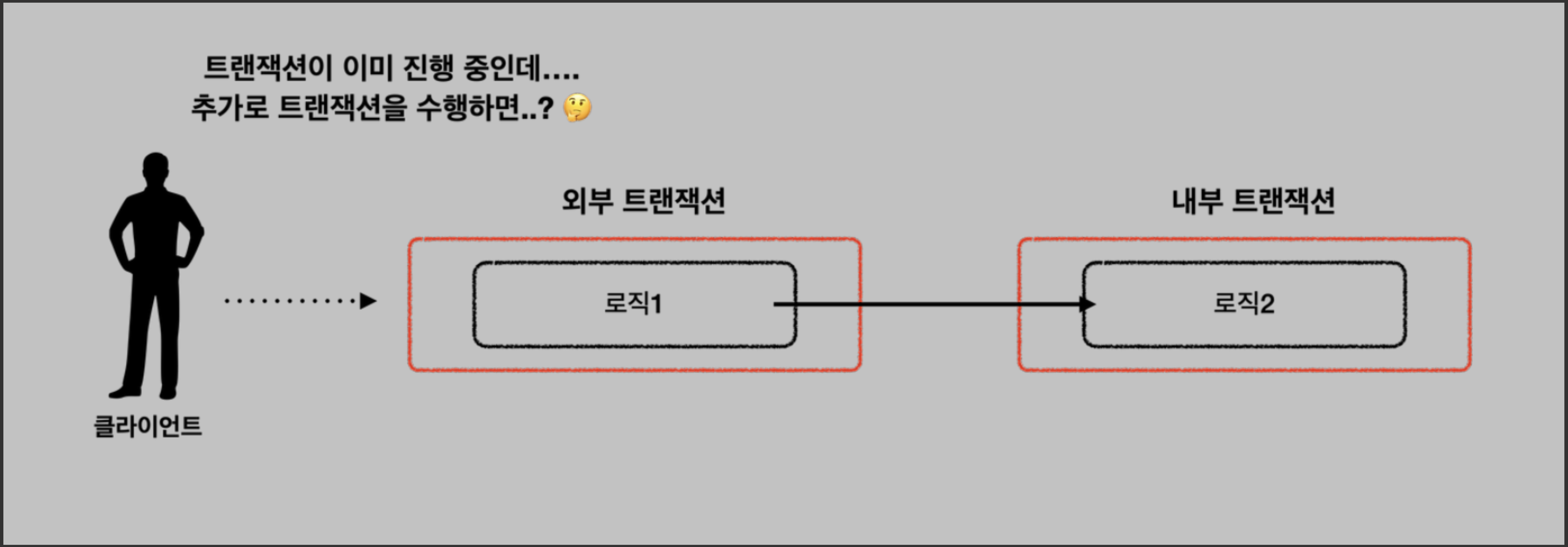
3. Advisors의 Advice로 등록된 TransactionInterceptor 호출

4. invokeWithInTransaction 호출



```
if (retVal != null && txAttr != null) {   txAttr: "PROPAGATION_REQUIRED,ISOLATION_DEFAULT"    retVal: null
    TransactionStatus status = txInfo.getTransactionStatus();
    if (status != null) {
        label185: {
            if (retVal instanceof Future) {
                Future<?> future = (Future) retVal;
                if (future.isDone()) {
                    try {
                        future.get();
                    } catch (ExecutionException var27) {
                        ExecutionException ex = var27;
                        if (txAttr.rollbackOn(ex.getCause())) {
                            status.setRollbackOnly();
                        }
                    } catch (InterruptedException var28) {
                        Thread.currentThread().interrupt();
                    }
                    break label185;
                }
            }

            if (vavrPresent && TransactionAspectSupport.VavrDelegate.isVavrTry(retVal)) {
                retVal = TransactionAspectSupport.VavrDelegate.evaluateTryFailure(retVal, txAttr, status);
            }
        }
    }
}

this.commitTransactionAfterReturning(txInfo);
```

Transaction Propagation



트랜잭션이 이미 진행 중인데....
추가로 트랜잭션을 수행하면..? 🤔
외부 트랜잭션
내부 트랜잭션
로직1
로직2
클라이언트

참고: 스프링-트랜잭션-전파- 힘들면 힘을 내자 블로그

Transaction Propagation

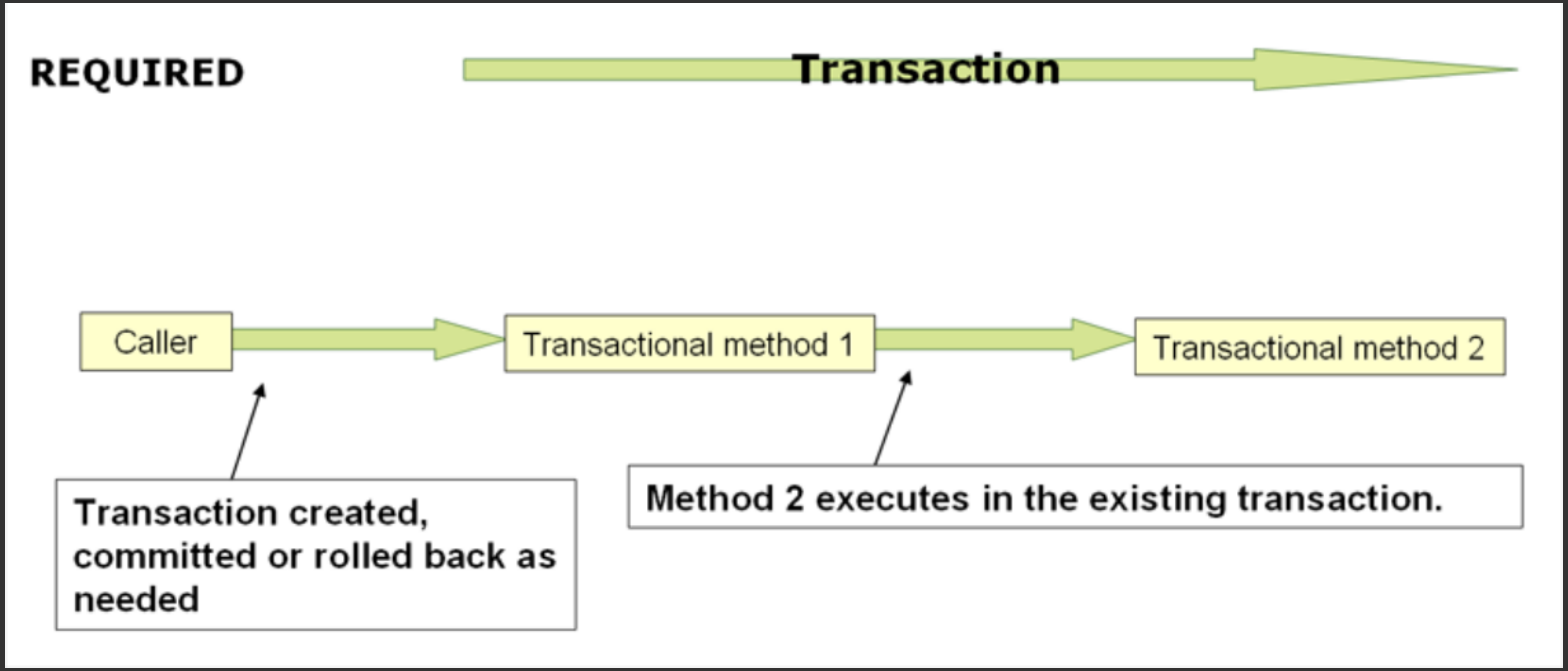
REQUIRED

REQUIRES_NEW

NESTED

SUPPORTS

Transaction Propagation: REQUIRED



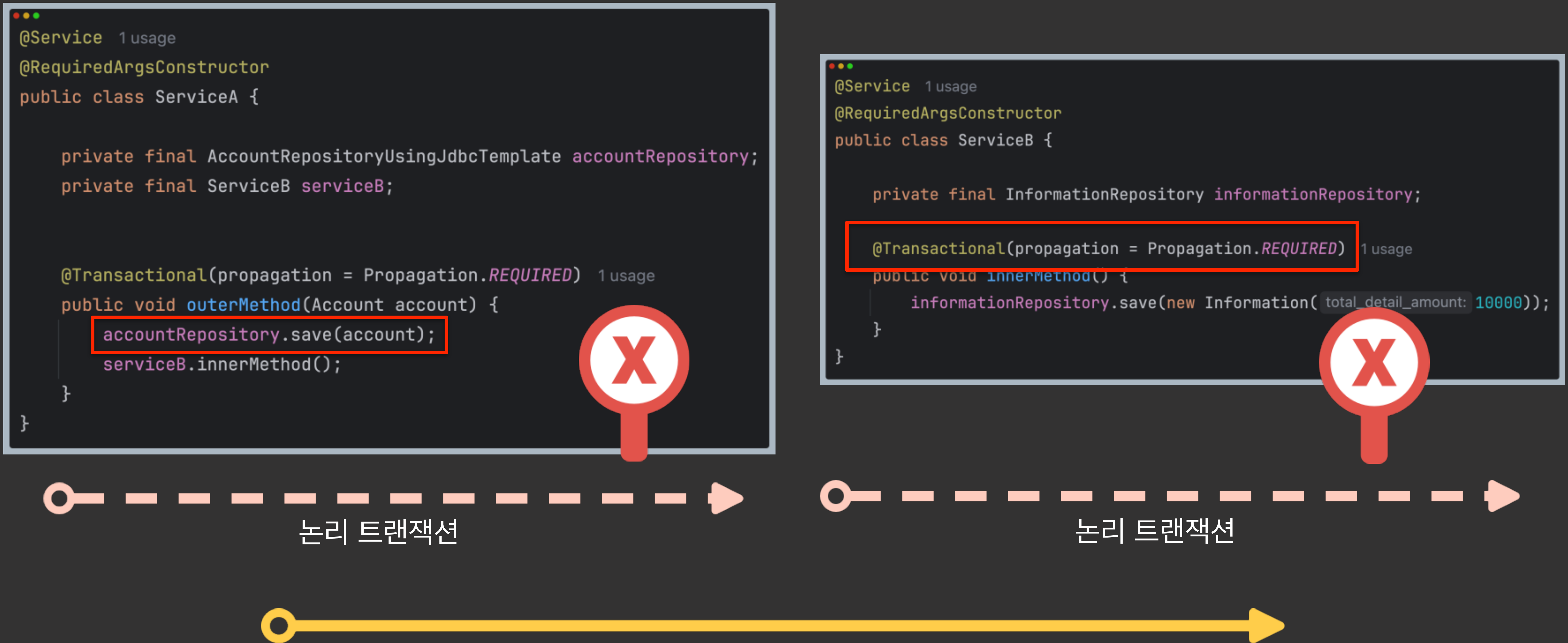
REQUIRED

Transaction created, committed or rolled back as needed

Method 2 executes in the existing transaction.

참고: 스프링 공식 문서

Transaction Propagation: REQUIRED



```java
@Service  1 usage
@RequiredArgsConstructor
public class ServiceA {

    private final AccountRepositoryUsingJdbcTemplate accountRepository;
    private final ServiceB serviceB;


    @Transactional(propagation = Propagation.REQUIRED)  1 usage
    public void outerMethod(Account account) {
        accountRepository.save(account);
        serviceB.innerMethod();
    }
}
```

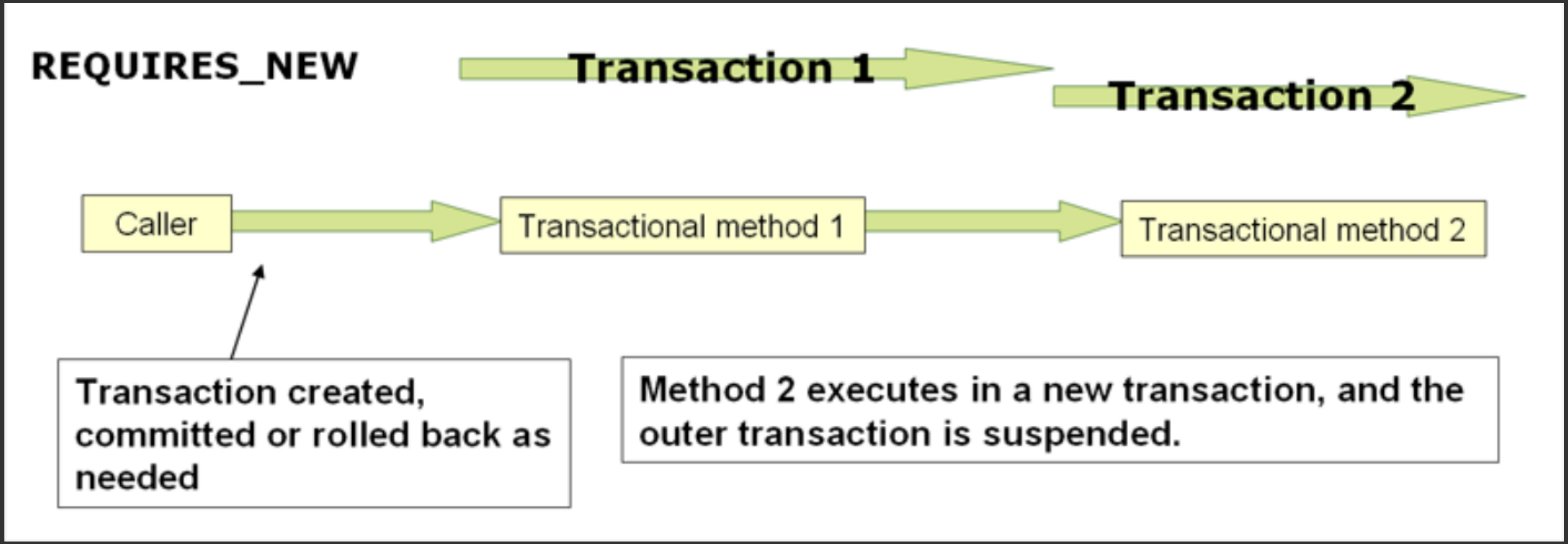
```java
@Service  1 usage
@RequiredArgsConstructor
public class ServiceB {

    private final InformationRepository informationRepository;

    @Transactional(propagation = Propagation.REQUIRED)  1 usage
    public void innerMethod() {
        informationRepository.save(new Information( total_detail_amount: 10000));
    }
}
```

논리 트랜잭션

논리 트랜잭션

물리 트랜잭션

Transaction Propagation: REQUIRES_NEW



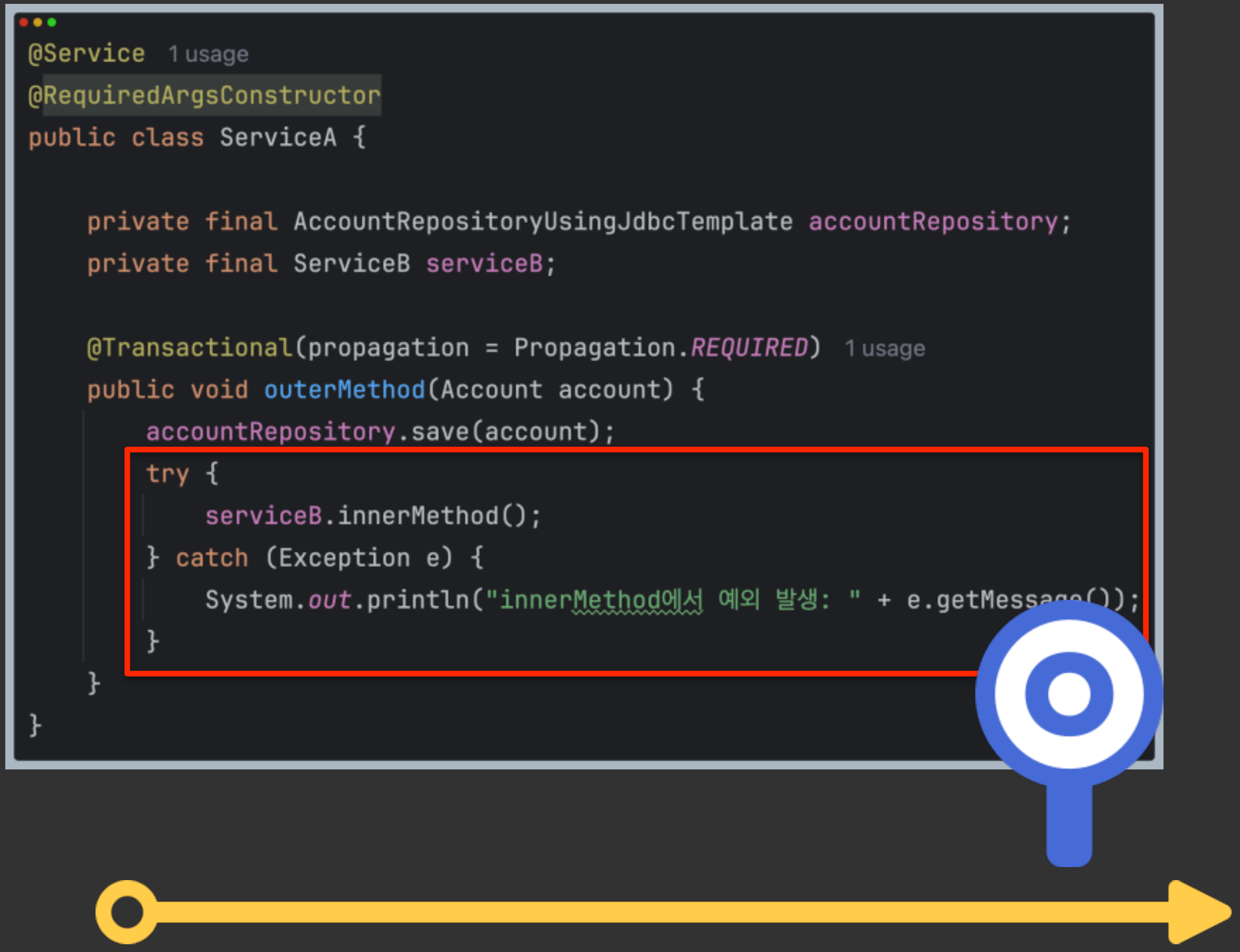
Transaction created, committed or rolled back as needed

Method 2 executes in a new transaction, and the outer transaction is suspended.

참고: 스프링 공식 문서

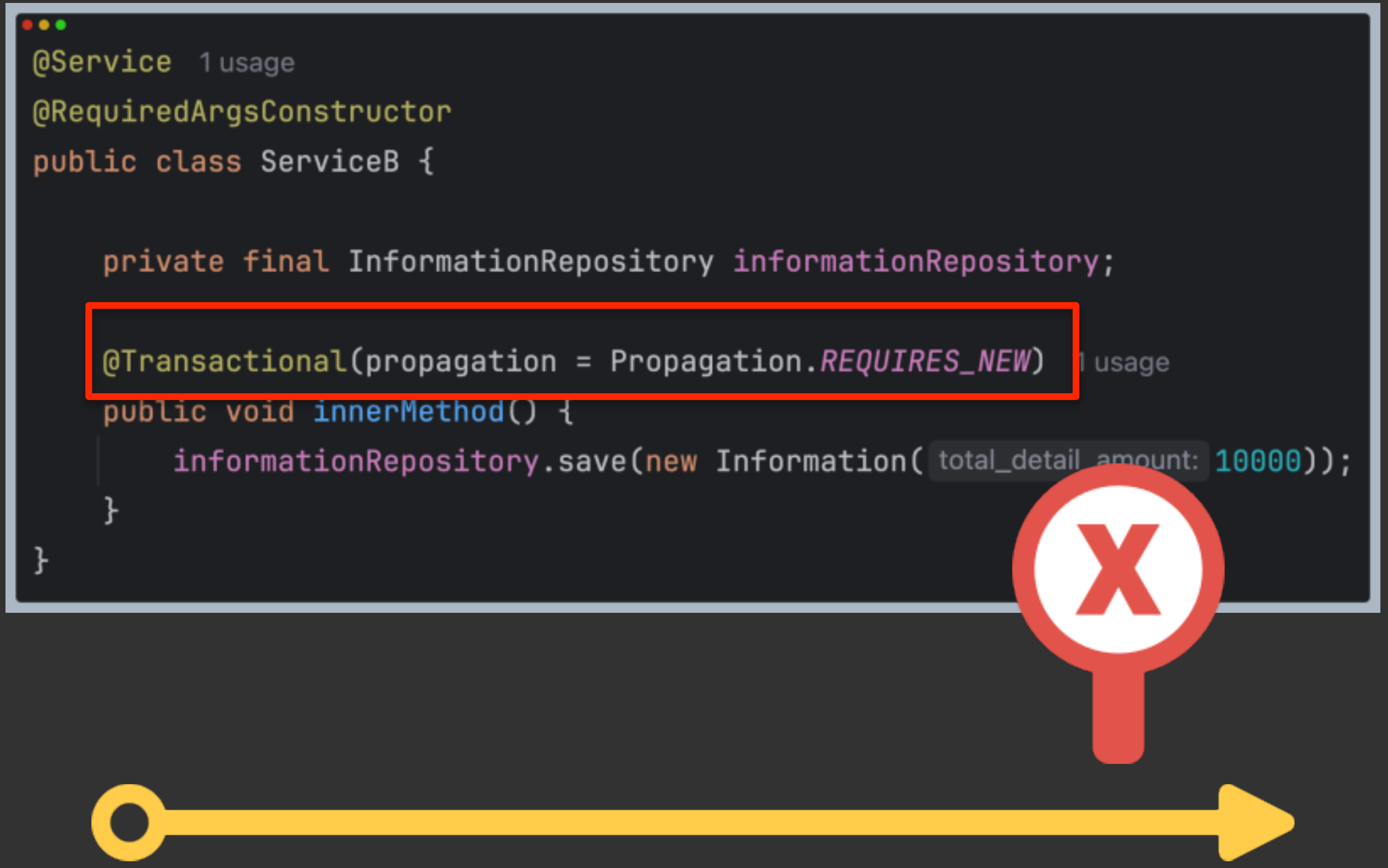
항상 새로운 트랜잭션을 시작, 새로운 데이터베이스 커넥션 사용

Transaction Propagation: REQUIRES_NEW



```java
@Service  1 usage
@RequiredArgsConstructor
public class ServiceA {

    private final AccountRepositoryUsingJdbcTemplate accountRepository;
    private final ServiceB serviceB;

    @Transactional(propagation = Propagation.REQUIRED)  1 usage
    public void outerMethod(Account account) {
        accountRepository.save(account);
        try {
            serviceB.innerMethod();
        } catch (Exception e) {
            System.out.println("innerMethod에서 예외 발생: " + e.getMessage());
        }
    }
}
```

물리 트랜잭션



```java
@Service  1 usage
@RequiredArgsConstructor
public class ServiceB {

    private final InformationRepository informationRepository;

    @Transactional(propagation = Propagation.REQUIRES_NEW)  1 usage
    public void innerMethod() {
        informationRepository.save(new Information( total_detail_amount: 10000));
    }
}
```

물리 트랜잭션

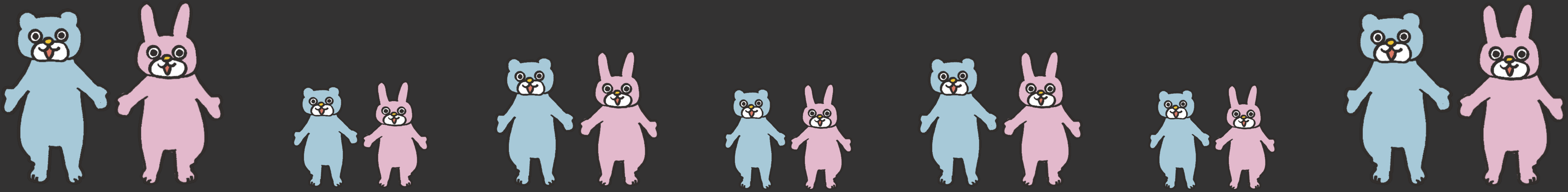
Review

Transaction 구현 방법 : 1. auto commit = false 로 설정 -> commit(); 또는 rollback();

선언적 트랜잭션 관리 vs **프로그래밍적 트랜잭션 관리**

@Transactional 동작 과정 : **프록시 객체** 사용 -> AOP

트랜잭션 전파 속성 : REQUIRED와 REQUIRES_NEW 의 차이



더 공부하면 좋을 내용

- Spring Data Access 추상화(Datasource, Jdbc, ORM)
- AOP 내부 동작 과정
- Spring Boot 동적 프록시 기술

References

- 스프링 트랜잭션 관리 - 스프링 공식 문서
- AOP 개념 및 특징
- 우아한테크-리차드의 @Transactional

The End ...