JAVA SPRING DATA

Prof. Júlio Machado

julio.machado@pucrs.br

JPA

Controle de Concorrência

Padrões de Concorrência Offline

- **Bloqueio Otimista**
  - Detecta conflitos entre transações de negócio concorrentes e desfazendo a transação.
  - Pressupõe que a chance de conflitos é baixa.
  - Faz uma validação de que os dados não foram alterados por outra sessão antes de confirmar a operação (operação de *commit*).

- **Bloqueio Pessimista**
  - Previne conflitos entre transações de negócio concorrentes permitindo que apenas uma transação de negócio acesse os dados de cada vez.
  - Pressupõe que a chance de conflitos é alta.
  - Bloqueia acessos originados a partir de outras transações.

Bloqueio Otimista

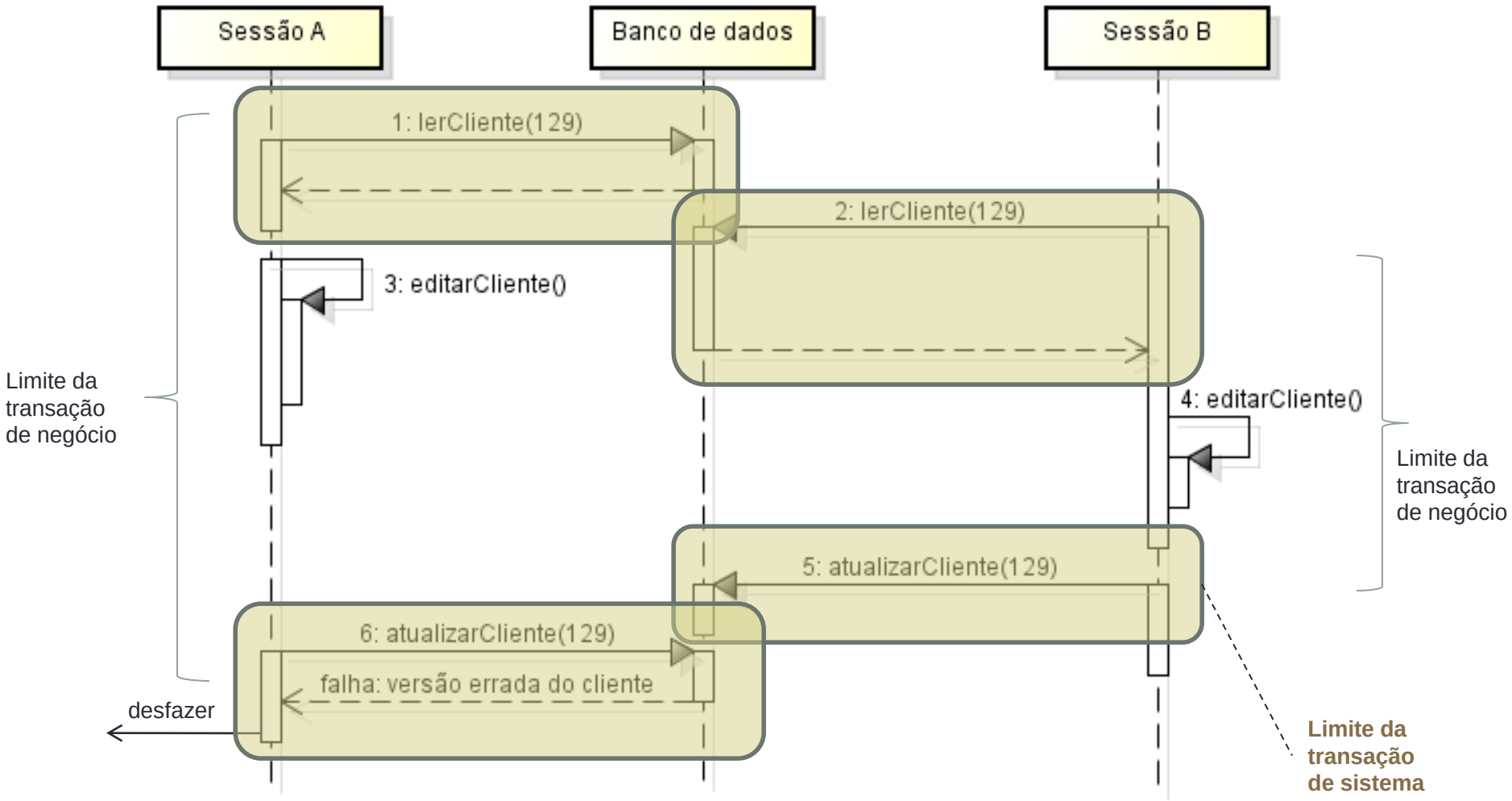
- Pode-se implementar usando um controle de versões por linha. Cada linha da tabela terá um campo extra indicando um número seqüencial de VERSÃO. Ao ler um registro, guardamos o valor em uma variável chamada **versaoLida**. Sempre que se faz uma alteração em uma linha, incrementa-se esse número. Dessa forma, usamos um comando UPDATE assim:

```
UPDATE Tabela
SET c1=v1, c2=v2, ..., VERSAO = versaoLida + 1
WHERE id=? AND VERSAO = versaoLida

int nroReg = stmt.executeUpdate();
```

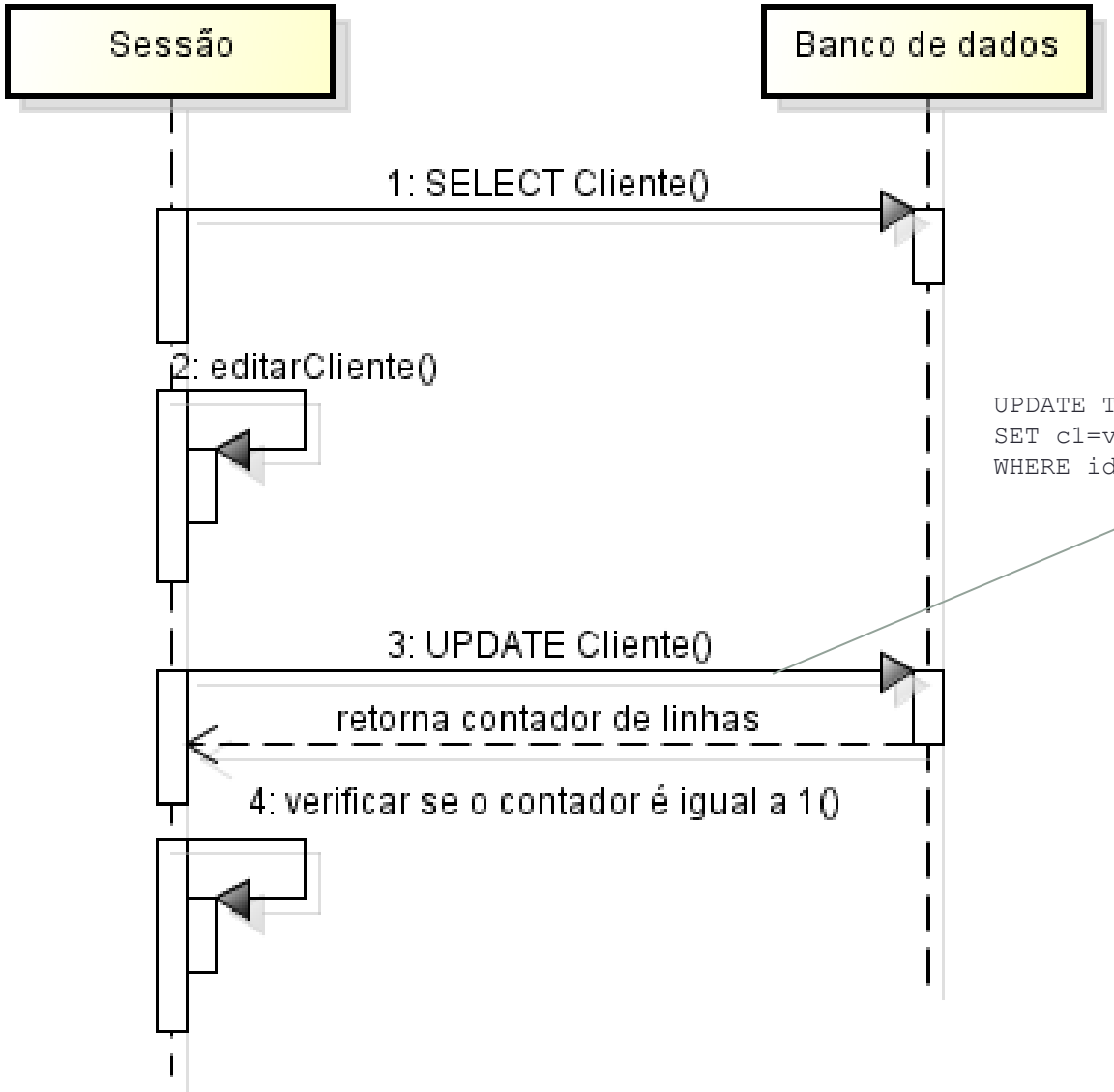
Se o número de registros atingidos (nroReg) for 1, a operação foi bem sucedida.

Se for zero, então é porque outra sessão fez alguma alteração entre a leitura e a tentativa de atualização (neste caso, deve-se desfazer a transação de sistema).

Bloqueio Otimista



Bloqueio Otimista




```
UPDATE Tabela
SET c1=v1, c2=v2, ..., VERSAO = versaoLida + 1
WHERE id=? AND VERSAO = versaoLida
```

Bloqueio Otimista

- JPA suporta o bloqueio otimista
  - Opção padrão dos provedores
- Entidade deve ser habilitada para uso do bloqueio otimista através da anotação *@Version* que define o atributo de controle de versão
  - Valor é controlado pelo EntityManager
  - Somente um atributo por entidade
  - Tipos usuais: *int*, *Integer*, *long*, *Long*, *short*, *Short*, *LocalDateTime*, *OffsetDateTime*, *ZonedDateTime*, *Instant*
- EnityManager ger uma exceção *OptimisticLockException* caso ocorra um conflito otimista

Bloqueio Otimista

- Ex.:

```
@Entity
public static class Person {

        @Id
        @GeneratedValue
        private Long id;

        @Column(name = "`name`")
        private String name;

        @Version
        private long version;

        …

}
```

Bloqueio Otimista

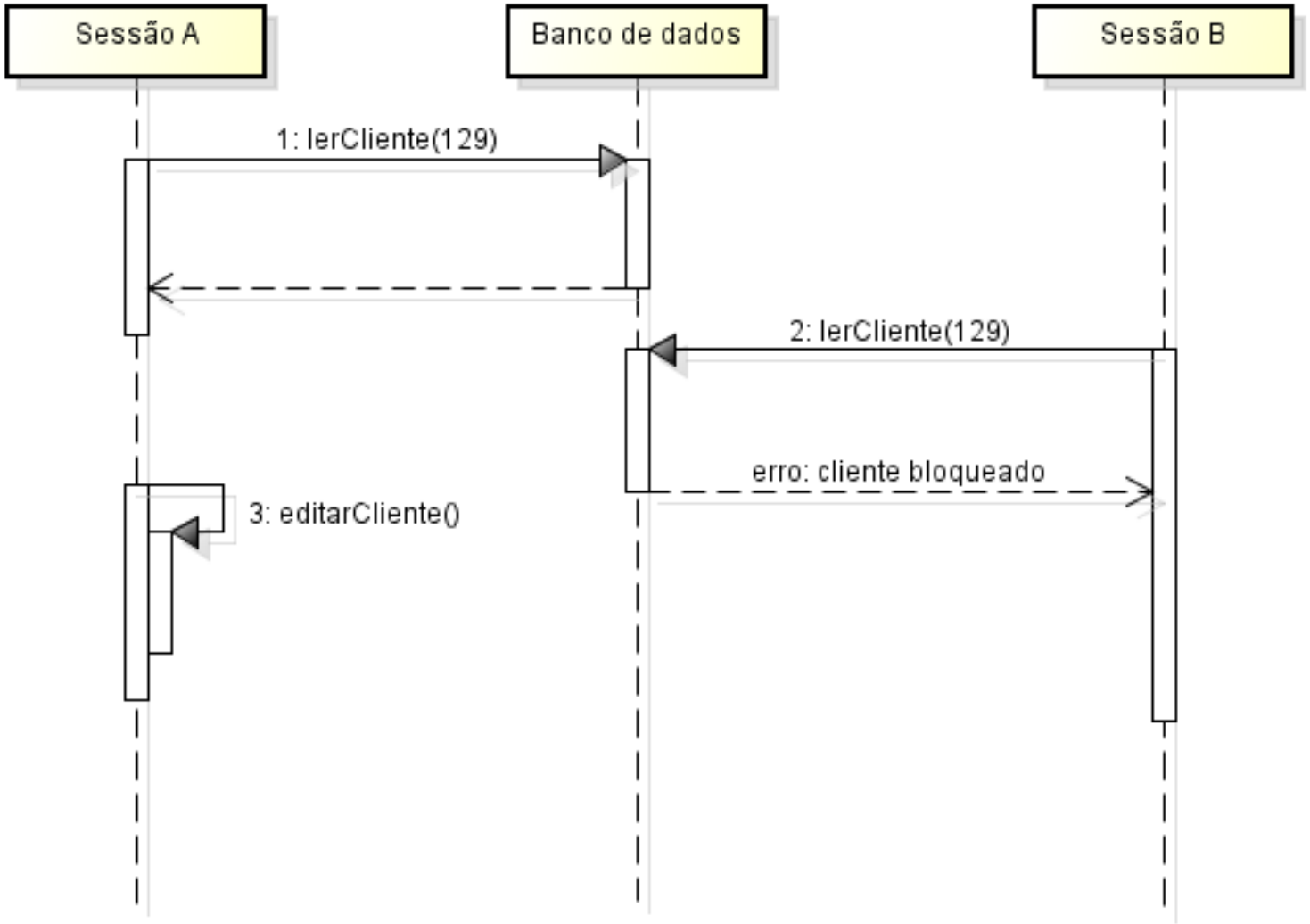
- Ex.:

```
@Entity
public static class Person {

        @Id
        @GeneratedValue
        private Long id;

        @Column(name = "`name`")
        private String name;

        @Version
        private LocalDateTime lastUpdated;

        …

}
```

Bloqueio Pessimista

- O bloqueio pessimista, por sua vez, previne conflitos evitando-os completamente. Uma transação terá certeza de ser terminada sem ser interrompida pelo controle de concorrência.
  - Usar quando a possibilidade de conflitos for alta (ou quando o custo de um conflito for inaceitável, independentemente de sua probabilidade).
  - Na prática, somente usar quando realmente necessário, porque cria muitos problemas de disputa de dados entre diferente sessões.

Bloqueio Pessimista



Bloqueio Pessimista

- JPA suporta o bloqueio pessimista
  - Pode ser utilizado sobre qualquer entidade
  - Pode ser configurado para a obtenção de locks de leitura ou escrita
- EntityManager provê uma enumeração para habilitar o tipo de bloqueio desejado
  - LockModeType.PESSIMISTIC_READ – lock de leitura; outras transações podem ler dados, mas não alterar ou remover;
  - LockModeType.PESSIMISTIC_WRITE – lock de escrita; outras transações não podem ler, alterar ou remover dados;
  - LockModeType.PESSIMISTIC_FORCE_INCREMENT – lock que impede modificação ou remoção de dados sobre uma entidade com campo de versionamento; incremente a versão da entidade ao final;
- EnityManager ger uma exceção *PessimisticLockException* caso ocorra um conflito pessimista ou *LockTimeoutException*

Bloqueio Pessimista

- Ex.: método *lock()*

```
EntityManager em = ...;
Person person = ...;
em.lock(person, LockModeType.PESSIMISTIC_READ);
```

Bloqueio Pessimista

- Ex.: método *find()*

```
EntityManager em = ...;
String personPK = ...;
Person person = em.find(Person.class, personPK,
LockModeType.PESSIMISTIC_WRITE);
```

Bloqueio Pessimista

- Ex.: método *refresh()*

```
EntityManager em = ...;
String personPK = ...;
Person person = em.find(Person.class, personPK);
...
em.refresh(person,
LockModeType.PESSIMISTIC_FORCE_INCREMENT);
```

Bloqueio Pessimista

- Ex.: método *setLockMode()*

```
EntityManager em = ...;
Query q = em.createQuery(...);
q.setLockMode(LockModeType.PESSIMISTIC_FORCE_INCREMENT);
```

Bloqueio Pessimista

- Ex.: elemento *lockMode*

```
@NamedQuery(name="lockPersonQuery",
  query="SELECT p FROM Person p WHERE p.name LIKE :name",
  lockMode=PESSIMISTIC_READ)
```

SPRING FRAMEWORK

Conceitos Adicionais

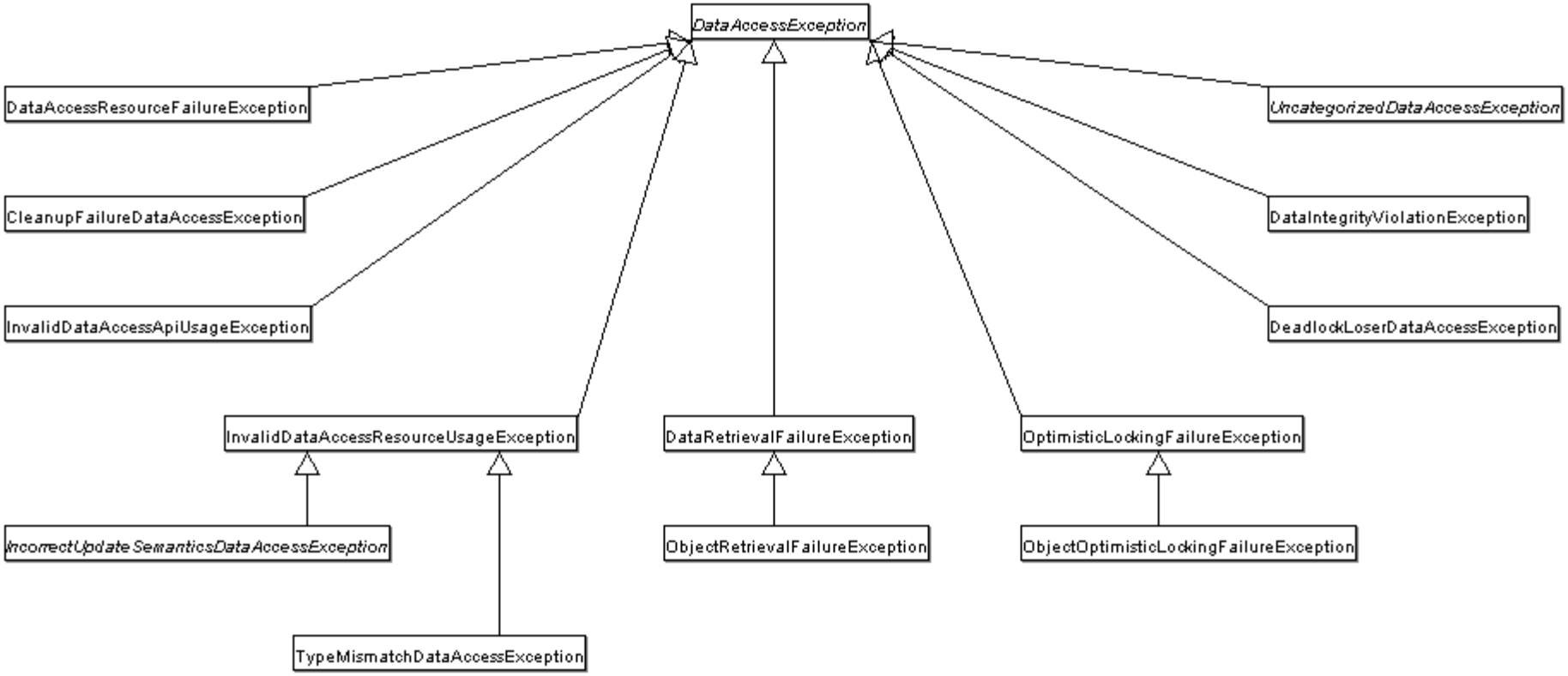
Spring - Repositórios

- Anotação *@Repository* é utilizada para habilitar o suporte do Spring Framework
- Ex.:

```
@Repository
public class SomeMovieFinder implements MovieFinder
{
...
}
```

Spring - Repositórios e Exeções

- Spring utiliza um mapeador para as exceções de uma tecnologia específica (como *SqlException*, *PersistenceException*, etc) em exceções *DataAccessException*



Spring - Repositórios e EntityManager

- Repositórios podem utilizar injeção de dependências padrão sem dependências adicionais sobre o Spring Framework:
  - *@PersistenceUnit* para obter *EntityManagerFactory*
  - *@PersistenceContext* para obter *EntityManager*

Spring - Repositórios e EntityManager

- Ex.:

```
@Repository
public class ProductRepositoryImpl implements ProductRepository {
    @PersistenceContext
    private EntityManager em;
    public Collection loadProductsByCategory(String category) {
        Query query = em.createQuery("from Product as p where
p.category = :category");
        query.setParameter("category", category);
        return query.getResultList();
    }
}
```

Spring - Transações

- Spring fornece um gerenciador de transações
  - Importante: a configuração do gerenciador de transações muitas vezes depende da configuração da fonte de dados
- As transações podem ser manipuladas de forma programática ou de forma declarativa via anotações
- O gerenciador de transações é implementado sobre o suporte de *AOP – Aspect Oriented Programming* do Spring Framework

Spring - Transações

- Gerenciamento pelo contêiner
  - Opção padrão indicada pelo uso da anotação *@Transactional* sobre uma classe ou individualmente sobre métodos
  - Contexto de persistência está ligado a transações JTA (Jakarta Transactions API)
  - Cada método pode ser associado a uma transação
    - Transação inicia e realiza o commit de forma automática
    - Transação realiza rollback quando ocorre uma exceção

Spring - Transações

- Gerenciamento pela aplicação
  - Permite o controle fino sobre as transações e o ciclo de vida do EntityManager
  - Transações JTA suportadas através do objeto *jakarta.transaction.UserTransaction* e seus métodos *begin()*, *commit()* e *rollback()*
  - Para obter o objeto de transação:
    - Utilizar método *getTransaction()* do EntityManager
    - Utilizar injeção de dependência via *@Resource* em um objeto *UserTransaction*

Spring - Transações

- Interface *PlatformTransactionManager*

```
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(
            TransactionDefinition definition) throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

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

Spring - Transações

- Interface *TransactionStatus*

```
public interface TransactionStatus extends SavepointManager {
    boolean isNewTransaction();
    boolean hasSavepoint();
    void setRollbackOnly();
    boolean isRollbackOnly();
    void flush();
    boolean isCompleted();
}
```

Spring - Transações

- Anotação @*Transactional* sobre classes ou métodos possui variadas configurações
  - Valor padrão de *propagation* é *PROPAGATION_REQUIRED*
  - Valor padrão de *isolation* é *ISOLATION_DEFAULT*
  - Valor padrão de *readOnly* é *false*
  - Valor padrão de *timeout* é o valor configurado no gerenciador de transações

Spring - Transações

- Ex.:

```
@Transactional(readOnly = true)
public class DefaultFooService implements FooService {
    public Foo getFoo(String fooName) {
    }
    @Transactional(readOnly = false, propagation =
Propagation.REQUIRES_NEW)
    public void updateFoo(Foo foo) {
    }
}
```

Spring - Transações

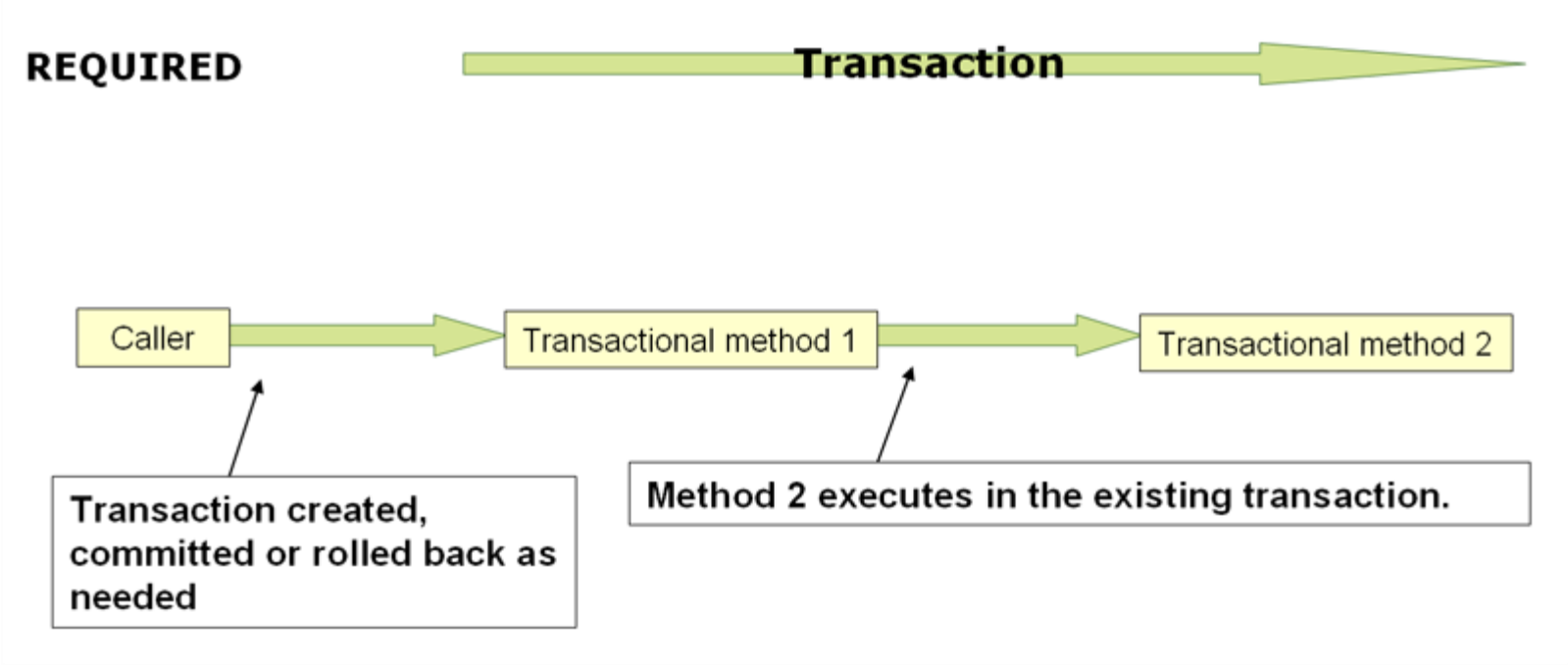
- Spring suporta um tipo especial de transação somente de leitura, especificado através do valor da propriedade *readOnly*
- Com isso, o framework pode utilizar otimizações na execução das transações

Spring - Transações

- A política de propagação define os limite de uma transação
- Políticas definidas na enumeração *Propagation*:
  - *PROPAGATION_REQUIRED*
  - *PROPAGATION_REQUIRES_NEW*
  - *PROPAGATION_NESTED*
  - *PROPAGATION_MANDATORY*
  - *PROPAGATION_NEVER*
  - *PROPAGATION_NOT_SUPPORTED*
  - *PROPAGATION_SUPPORTS*

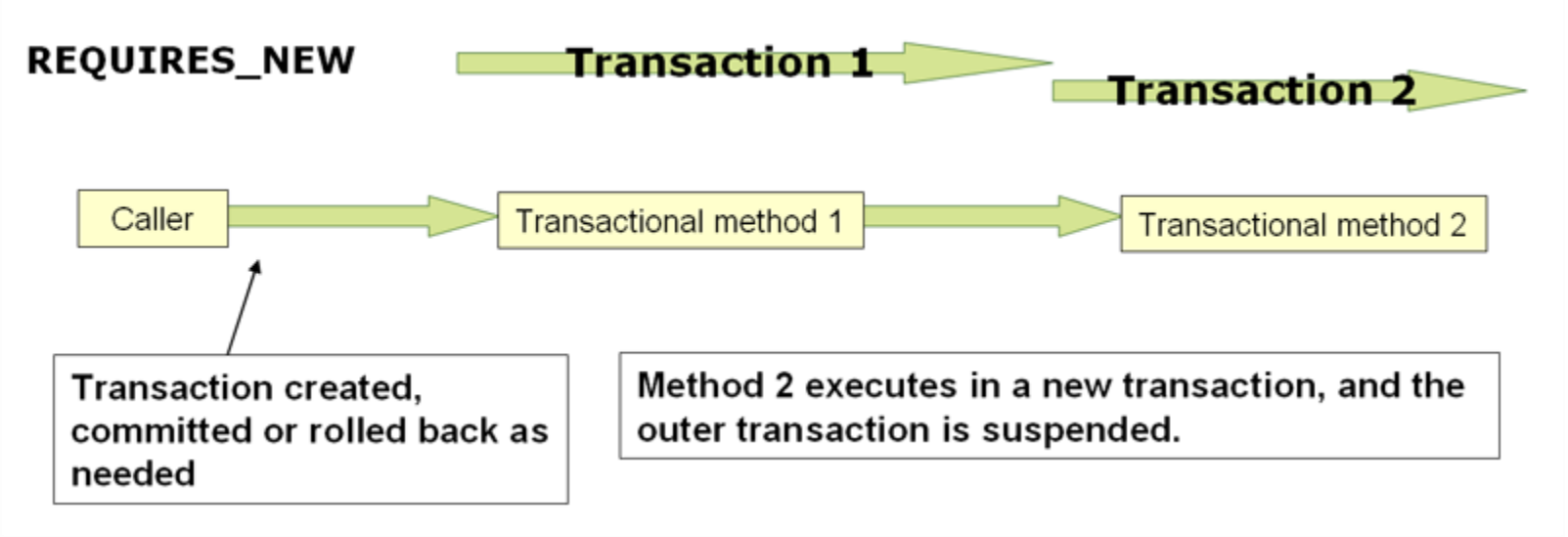
Spring - Transações

- PROPAGATION_REQUIRED:
  - Método só pode ser executado em uma transação
  - Caso seja invocado a partir de uma, então fará parte desta
  - Se não existir uma transação em execução, uma nova será criada



Spring - Transações

- PROPAGATION_REQUIRES_NEW:
  - Método só pode ser executado dentro de sua própria transação
  - Caso seja invocado a partir de uma, esta será paralisada e retomada ao final
  - Uma nova transação é sempre criada



Spring - Transações

- PROPAGATION_MANDATORY:
  - Método só pode ser executado em uma transação
  - Se não existir uma transação em execução, uma exceção será disparada
- PROPAGATION_NESTED:
  - Método executa dentro de uma transição aninhada se uma transição corrente já existe
  - Caso contrário, se comporta domo PROPAGATION_REQUIRED

Spring - Transações

- PROPAGATION_NEVER:
  - Método jamais deve ser executado em uma transação
  - Caso seja invocado a partir de uma, uma exceção será disparada
- PROPAGATION_NOT_SUPPORTED:
  - Método não deve ser executada em uma transação
  - Caso seja invocado a partir de uma, esta será paralisada e retomada ao final
  - Não é criada nenhuma transação
- PROPAGATION_SUPPORTS:
  - Tanto faz se o método é executado em uma transação ou não

Spring - Transações

- A política de isolamento define a influência de outra transação concorrente
- Políticas definidas na enumeração *Isolation*:
  - *ISOLATION_DEFAULT*
  - *ISOLATION_READ_UNCOMMITED*
  - *ISOLATION_READ_COMMITED*
  - *ISOLATION_REPEATABLE_READ*
  - *ISOLATION_SERIALIZABLE*

Spring - Transações

- ISOLATION_DEFAULT:
  - A transação adotará o comportamento padrão do BD
- ISOLATION_READ_UNCOMMITED:
  - Permite à transação ler registros que ainda não tenham sido comitados por outra transação
- ISOLATION_READ_COMMITED:
  - Permite à transação ler registros que ainda somente tenham sido comitados por outra transação
- ISOLATION_REPEATABLE_READ:
  - A transação somente obterá resultados diferentes caso sejam registros alterados durante a sua execução
- ISOLATION_SERIALIZABLE:
  - Isolamento total