12 Transações

Helder da Rocha (helder@acm.org)

Objetivos

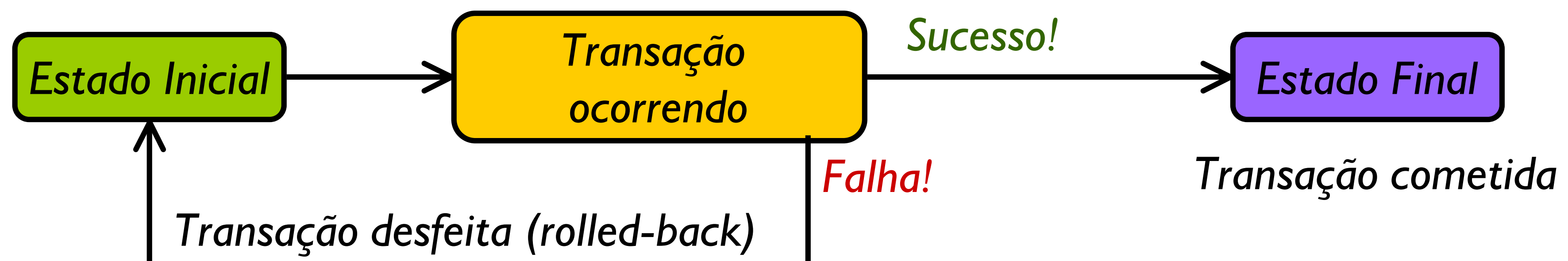
- *Este módulo explora os principais assuntos relativos ao controle de transações em aplicações EJB*
  - *Transações ACID*
  - *Controle de transações em Entity Beans, Session Beans e Message-driven beans*
  - *Formas de controle de transações em EJB: Bean-Managed Transactions (BMT) e Container-Managed Transactions (CMT)*
  - *Transações iniciadas pelo cliente*
  - *Atributos de escopo e propagação de transações*
  - *Níveis de isolamento em JDBC para transações*
- *São explorados também*
  - *Relacionamento entre exceções e transações*

O que são transações?

- Transações são uma técnica para simplificar a programação de aplicações
  - Programador não precisa se preocupar com recuperação de falhas e programação para sistemas multiusuário
- São unidades atômicas de procedimento
  - Sistema de transações garante que o procedimento ou termina com sucesso ou é completamente desfeita
- Suporte a transações é um componente essencial da arquitetura EJB
  - Programador EJB pode escolher entre duas formas de demarcação de transações: explícita, ou programática (Bean-Managed - BMT), e implícita, ou declarativa (Container-Managed - CMT)

Commit e Rollback

- *Uma transação engloba uma unidade de trabalho.*
  - *Durante o processo, várias sub-operações são realizadas e resultados temporários são obtidos.*
  - *No final, a transação é cometida (isto é o commit) e o resultado final é feito durável se cada sub-operação funcionar com sucesso.*
  - *Caso alguma sub-operação falhe, um processo para reverter as alterações temporárias realizadas é iniciado. Isto é o rollback, e os dados envolvidos voltam aos estados que tinham antes do início da transação*

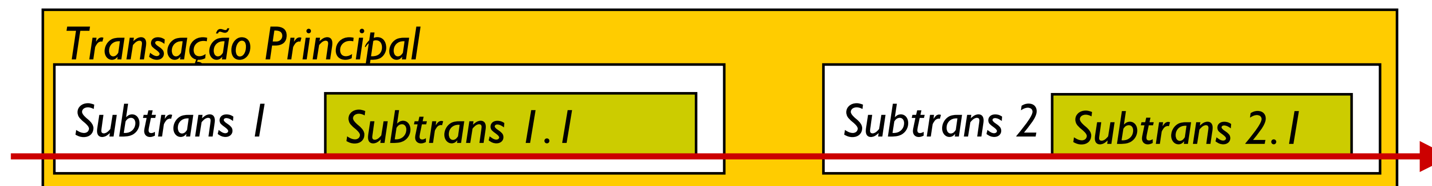


Transações são Ácidas!

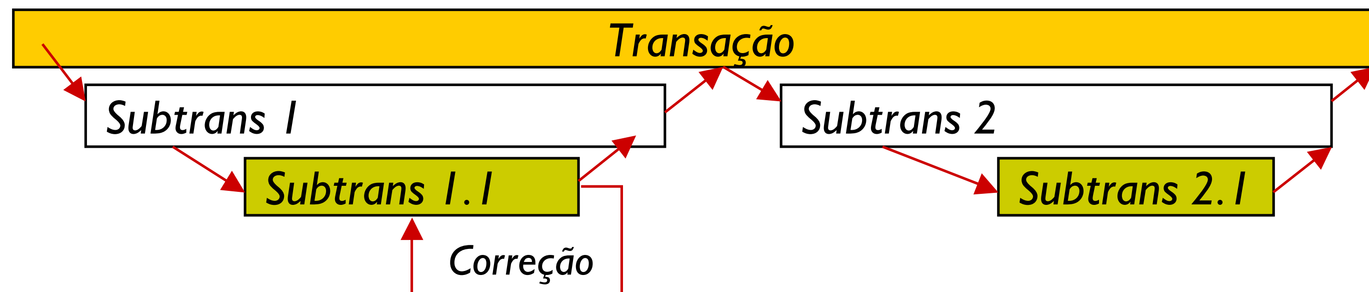
- ACID - características essenciais de uma transação: ela deve ser **A**tômica, **C**onsistente, **I**solada e **D**urável
- Atômica
  - Garante que todas as operações sejam tratadas como uma única unidade de trabalho. Todas as tarefas de uma unidade transacional devem funcionar sem erros ou todo o processo é revertido.
- Consistente
  - O estado do sistema após uma transação deve manter-se consistente (transações devem englobar processos de negócio completos)
- Isolada
  - Transação deve poder executar sem interferência de outros processos. Isto é possível utilizando sincronização.
- Durável
  - Dados alterados durante a transações devem ser guardados em meio persistente até que a transação complete com sucesso

Modelos transacionais

- *Flat transacions*
  - *Resultado "tudo ou nada": Se qualquer operação da seqüência falhar, operação inteira é abortada*



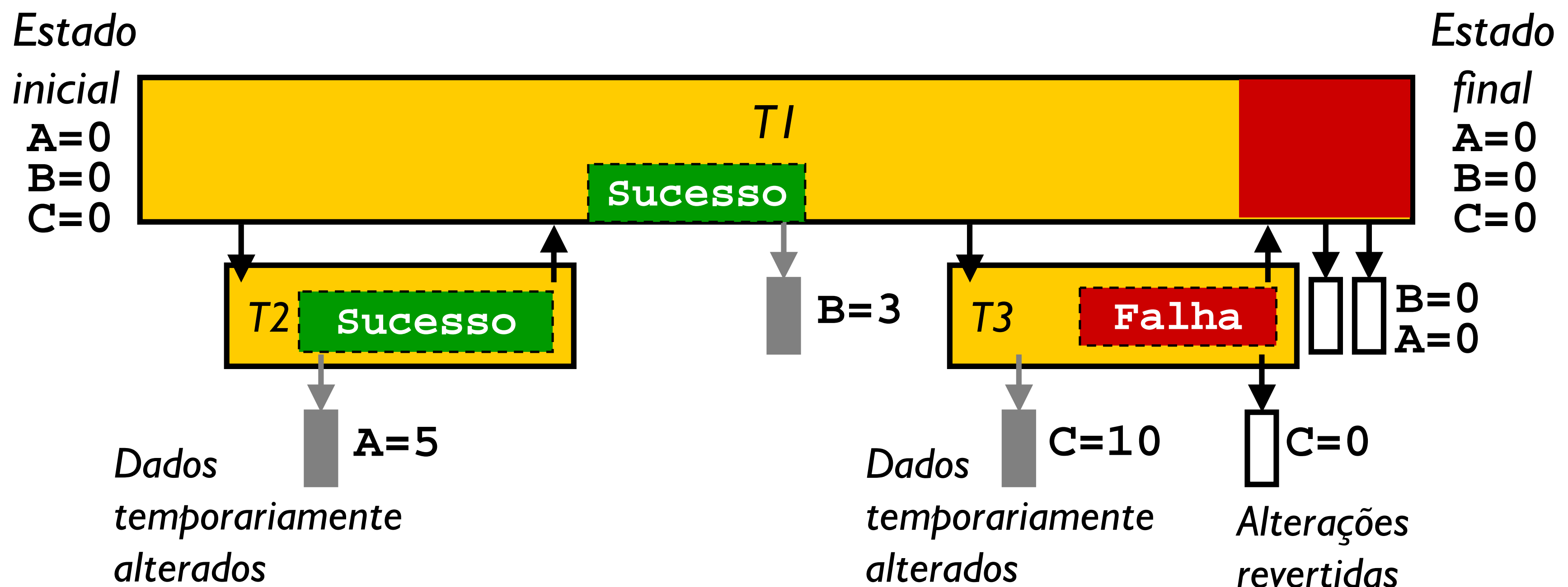
- *Nested transactions*
  - *Sub-transações são executadas separadamente e podem ser corrigidas e re-tentadas sem afetar o resultado final*



- *Chained transactions*
  - *Transações executam em etapas reversíveis (na falha, volta-se a um ponto previamente definido)*

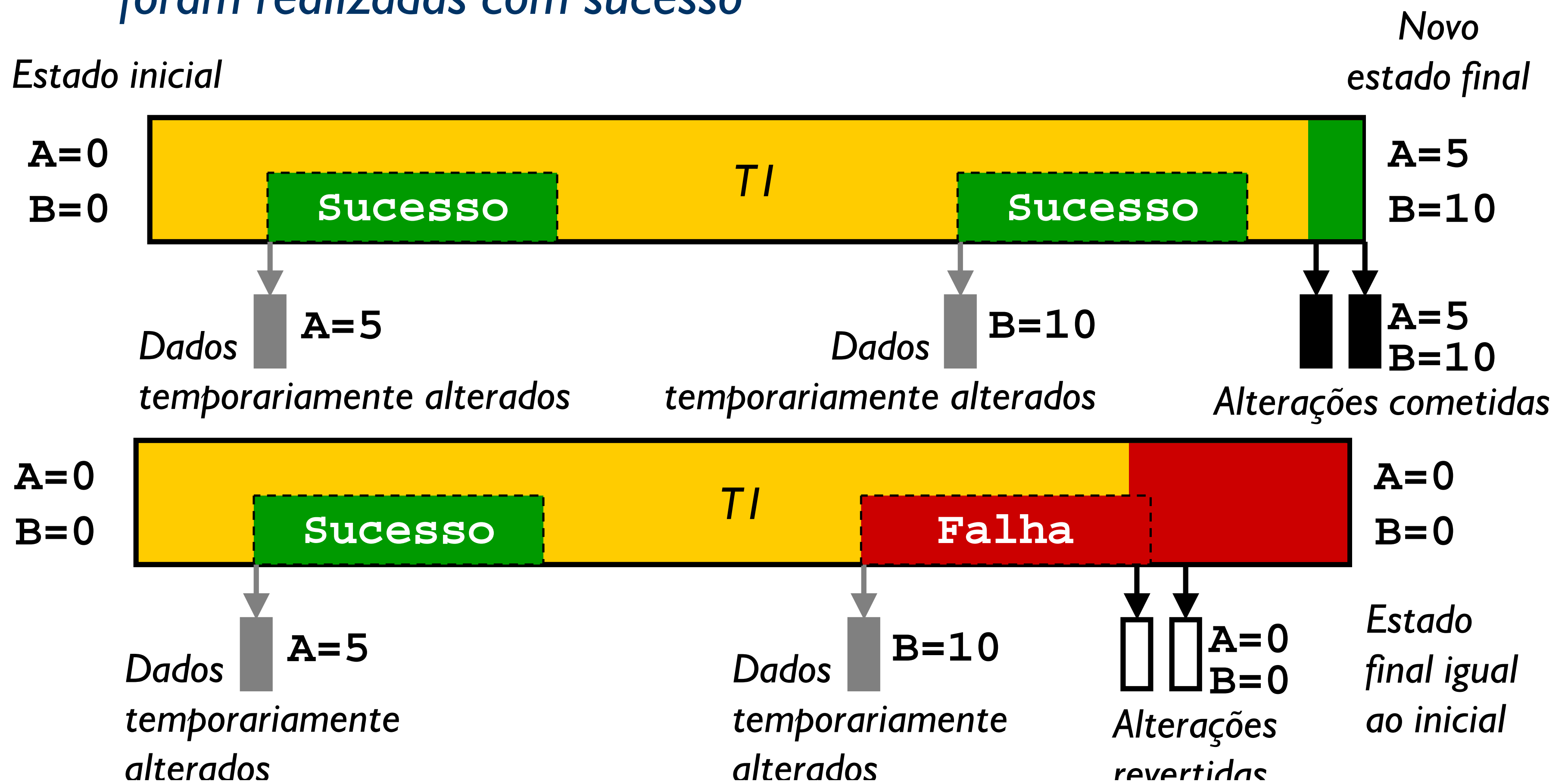
Nested Transactions

- Transações novas podem ocorrer no contexto de outras transações existentes
  - Se uma sub-transação falhar, ela pode tentar se recuperar e impedir que a transação principal falhe
  - Se ela não conseguir se recuperar, e causar o rollback da transação principal, todas as transações e sub-transações iniciadas no contexto da transação principal são desfeitas



Flat Transactions

- Neste modelo, as operações ocorrem na mesma dimensão. Se alguma delas falhar, todo o processo falha e é revertido
  - Operações chamadas são incluídas na transação
  - As alterações só são cometidas depois que todas as operações foram realizadas com sucesso

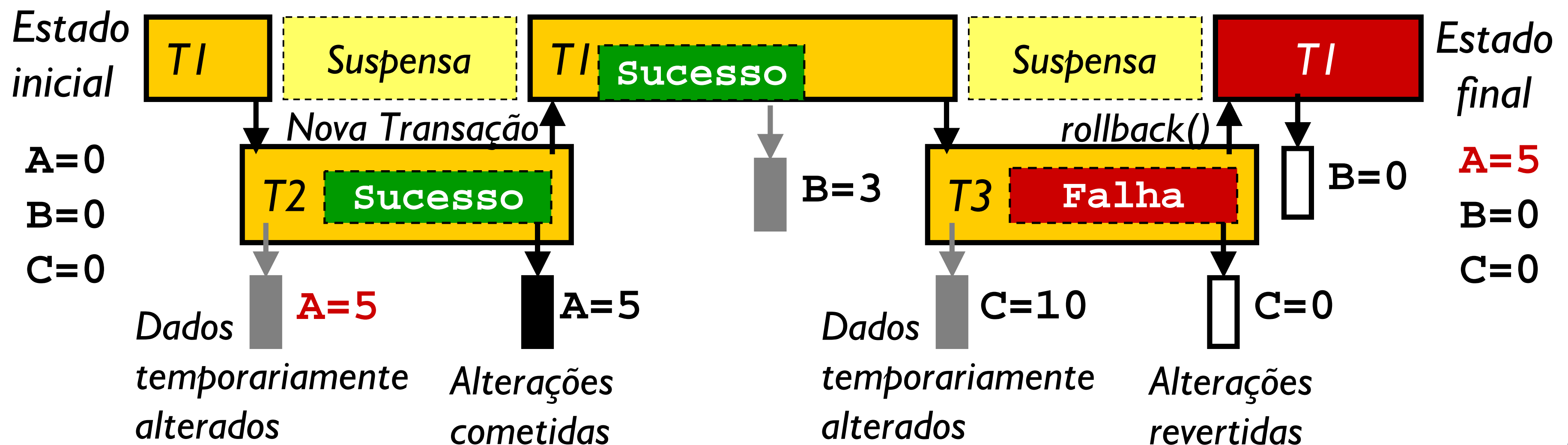


Modelo de transações suportado em EJB

- A especificação EJB não exige que o container suporte transações que ocorrem abaixo do contexto de outras transações: transações aninhadas
- Apenas "flat transactions" são suportadas (servidor pode suportar outros modelos, mas a portabilidade não é garantida)
- O comportamento de uma operação durante uma transação pode ser controlada pelo programador
  - Se operação continua transação anterior
  - Se operação interrompe transação anterior
  - Se operação inicia uma nova transação (não mantém contexto da transação anterior, já que não aninha)

Conseqüências de Flat Transactions

- Se método é chamado dentro de uma transação, ele pode continuar transação anterior (no mesmo nível)
  - Se ele falhar, transação inteira é revertida
- Se nova transação for iniciada no método que já faz parte de uma transação, ela é independente
  - Se falhar, pode sinalizar rollback() para desfazer transação externa (ou não, e permitir que transação externa continue)
  - O possível rollback da transação externa não afetará outras transações (T2) cujos resultados já foram cometidos

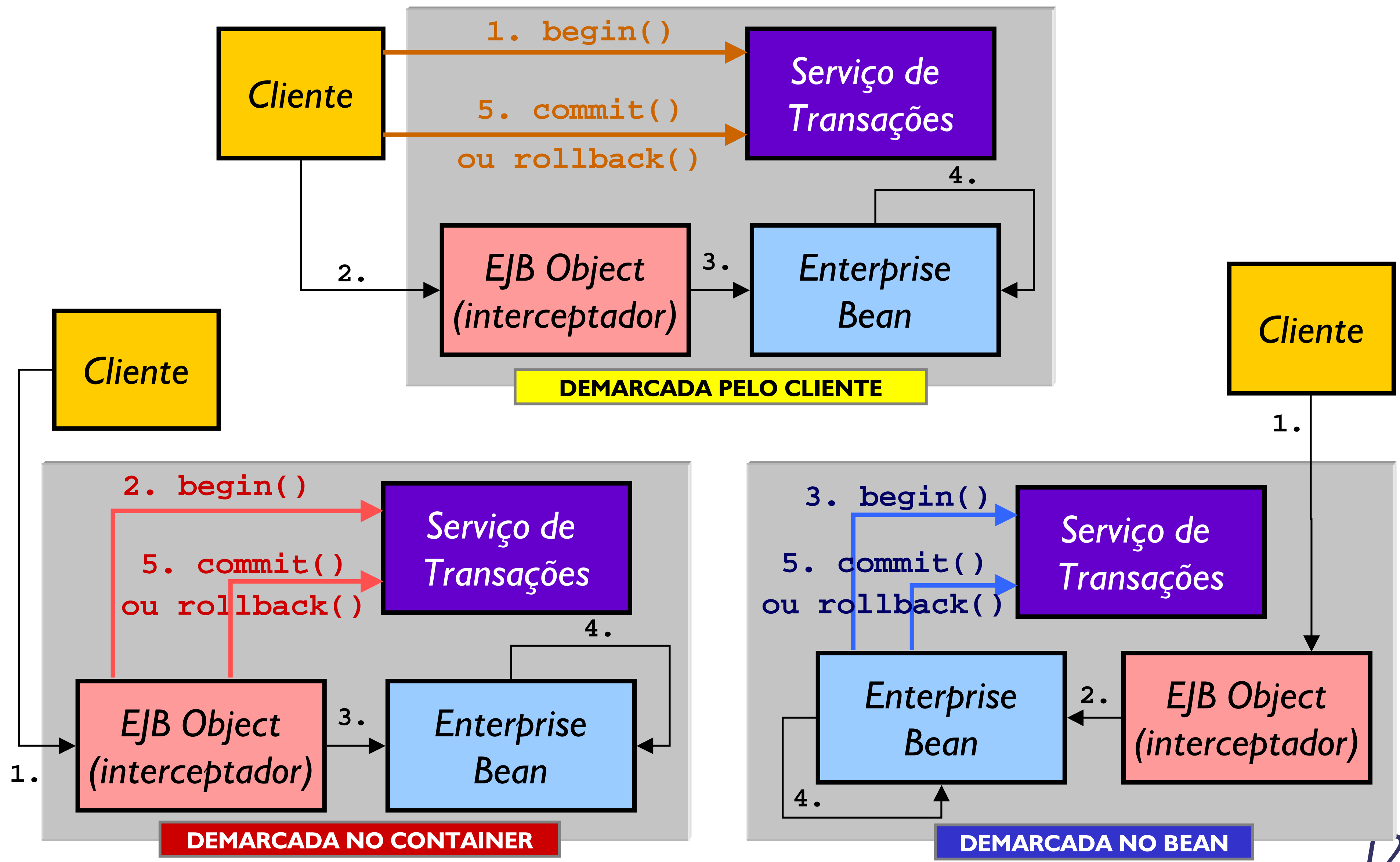


Demarcação de transações

- *O controle de transações em EJB resume-se a demarcação de transações*
  - *Consiste apenas do controle de quando ela será iniciada e quando será concluída (ou desfeita)*
  - *Não são controlados aspectos de baixo-nível (não existe interação entre o cliente e os gerenciadores de recursos ou de transações)*
- *Há várias formas de demarcar transações*
  - *Podem ser demarcadas no cliente (comuns, servlets, beans, etc.) e propagadas para os componentes*
  - *Podem ser demarcadas no servidor, de duas formas: no container (implícita), usando declarações no DD, ou no bean (explícita), usando APIs como JTA, JDBC ou JMS*

Demarcação de transações

- Formas de demarcar dependem de quem chama o serviço



Serviço de Transações

- Servidores J2EE oferecem serviço de transações distribuídas CORBA: Object Transaction Service (OTS)
  - O serviço está disponível para qualquer cliente CORBA (escrito em Java ou não) e pode ser obtido através do serviço de nomes (via JNDI ou COS Naming)
- Clientes também podem obter o serviço de transações do recurso que estão utilizando (se houver)
  - Bancos de dados (através de JDBC)
  - Sistemas de messaging (através de JMS)
- Para ter acesso a esses serviços, existem APIs.
  - JDBC ou JMS, para serviços do recurso utilizado
  - JTS ou JTA, para acesso ao OTS (distribuído)
- Beans também podem realizar controle declarativo

JTS e JTA

- JTS - Java Transaction Service é um mapeamento Java-CORBA da especificação Object Transaction Service (OTS 1.1)
  - JTS é usado por fabricantes de containers
  - Desenvolvedores de EJBs não precisam usar JTS (suporte por parte do container é opcional)
  - Pacote: org.omg.CosTransactions
- JTA - Java Transaction API é uma especificação de interfaces para o sistema de transações
  - JTA é utilizado por desenvolvedores de beans que têm controle explícito (programático) de transações (BMT)
  - Suporte por parte do container é obrigatório
  - Classes: javax.transaction.UserTransaction e outras

Demarcação explícita de transações

- Consiste em utilizar alguma API de controle de transações diretamente no código
  - Uso de métodos begin(), commit(), rollback(), etc.) para definir o início e o fim das unidades atômicas de código
- Suportada por clientes, session beans e message-driven beans (Entity Beans não podem ter transações controladas explicitamente)
- É preciso informar ao container que bean está usando Bean-Managed Transactions (BMT) através do DD:

```
<transaction-type>Bean</transaction-type>
```

- No código, use JTA, JDBC ou JMS
  - Métodos de Connection (java.sql)
  - Métodos de QueueSession ou TopicSession (javax.jms)
  - Métodos de UserTransaction (javax.transaction)

Transações com JDBC - exemplo

```
(...)
public void ship (String productId,
                  String orderId,
                  int quantity) {
  try {
     con.setAutoCommit(false); // con é java.sql.Connection
       updateOrderItem(productId, orderId);
       updateInventory(productId, quantity);
     con.commit();
  } catch (Exception e) {
    try {
      con.rollback();
      throw new EJBException("Transaction failed: "
                                    + e.getMessage());
    } catch (SQLException sqx) {
      throw new EJBException("Rollback failed: "
                                    + sqx.getMessage());
    }
  }
}
```

WarehouseBean.java

UserTransaction

- *É a única classe do JTA que todos os containers J2EE/EJB precisam implementar*
- *Seus métodos são usados para demarcar transações*
  - *begin(): marca o início*
  - *commit(): marca o término*
  - *rollback(): condena a transação*
  - *setRollbackOnly(): marca a transação para rollback*
  - *setTransactionTimeout(int segundos): define o tempo de vida de uma transação*
  - *getStatus(): retorna o status de uma transação. O status é um objeto da classe javax.transaction.Status* e *pode ter os valores STATUS_ACTIVE, STATUS_COMMITTED, STATUS_ROLLEDBACK, etc.*

Transações explícitas com JTA

```
(...)
   public void withdrawCash(double amount) {
      UserTransaction ut = sessionCtx.getUserTransaction();
      try {
         double mbState = machineBalance;
         ut.begin();
          updateChecking(amount);
          machineBalance -= amount;
          insertMachine(machineBalance);
         ut.commit();
      } catch (Exception ex) {
          try {
             ut.rollback();
             machineBalance = mbState;
          } catch (SystemException syex) {
              throw new EJBException
                 ("Rollback failed: " + syex.getMessage());
          }
          throw new EJBException
             ("Transaction failed: " + ex.getMessage());
      }
   }
```

EJBContext do bean (SessionContext ou MessageDrivenContext)

Estado de variável de instância NÃO será revertido em caso de rollback()

TellerBean.java

Transações iniciadas no cliente

- *O cliente pode iniciar transações usando as APIs JDBC e JMS*
  - *Cliente pode ser standalone, servlet, outro bean, etc.*
- *Se desejar utilizar o serviço OTS via JTA, é necessário obter uma instância da classe UserTransaction através de JNDI*
  - *O container deve disponibilizar o JTA na localidade java:comp/UserTransaction*

```
Context ctx = new InitialContext();
ut = (javax.transaction.UserTransaction)
      ctx.lookup("java:comp/UserTransaction");
ut.begin();
// realizar operações
ut.commit();
```

Session Beans com BMT

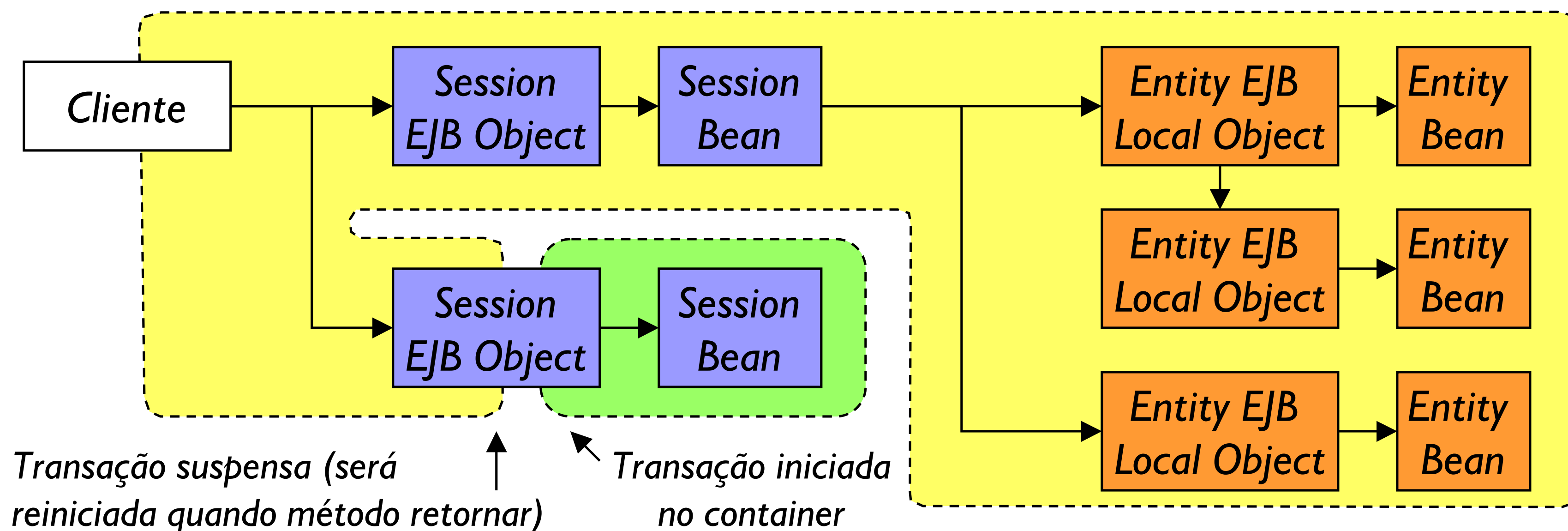
- *O container suspende qualquer transação iniciada pelo cliente ao executar método de um Session Bean que inicie uma transação usando BMT*
  - *Se a transação ocorre no método de um Stateless Session Bean, ela deve terminar antes que o método termine*
  - *Em Stateful Session Beans, uma transação pode ser iniciada em um método e terminada em outro. Ela será temporariamente suspensa no fim da primeira chamada e reiniciada na segunda. Este não é um procedimento recomendado.*
- *Para saber o estado de uma transação existente e decidir como agir, bean pode usar getStatus() de UserTransaction*

Transações em Message-driven beans

- Um cliente não chama um MDB diretamente
  - MDBs lêem mensagens de um destino JMS
  - Não há, portanto, propagação de transações de um cliente para um MDB
- MDBs podem iniciar novas transações
  - O escopo da transação deve iniciar e terminar dentro do método onMessage()
- Sempre que possível, use CMT com MDBs
  - Com CMT, a entrega da mensagem é parte da transação, já que é o container que a inicia: se houver rollback, container pode reenviar a mensagem
  - Com BMT, é preciso provocar um EJBException para evitar o acknowledgement e forçar o reenvio

Propagação de transações

- Transações terminam no mesmo lugar onde começaram
- Pode-se melhorar a performance de uma operação que faz muitas chamadas (principalmente se forem chamadas a Entity Beans) colocando-a no contexto de uma transação
  - O contexto da transação será propagado para todos os métodos chamados (se não iniciarem nova transação)
  - Se métodos chamados iniciarem nova transação, a transação principal será suspensa até que o método termine



Controle implícito (declarativo)

- *Container-Managed Transactions (CMT)*
  - *Controle de transações totalmente gerenciado pelo container*
  - *Não permite o uso de métodos commit() e rollback() de java.sql.Connection ou javax.jms.Session dentro do código*
  - *Única forma de controlar transações em Entity Beans*
- *No deployment descriptor, espeficique o uso de CMT abaixo de <session> ou <message-driven>*

```
<transaction-type>Container</transaction-type>
```

- *Depois, defina a política de transações para cada método*

```
<assembly-descriptor>
   <container-transaction>
      <method> ... </method>
      <trans-attribute>Required</trans-attribute>
   </container-transaction>
   (...)
</assembly-descriptor>
```

Demarcação declarativa - exemplo

```
<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>BankEJB</ejb-name>
            <home>j2eetut.bank.BankHome</home>
            <remote>j2eetut.bank.Bank</remote>
            <ejb-class>j2eetut.bank.BankBean</ejb-class>
            <session-type>Stateful</session-type>
            <transaction-type>Container</transaction-type>
            (...)
        </session>
    </enterprise-beans>
    <assembly-descriptor>
        <container-transaction>
            <method>
                <ejb-name>BankEJB</ejb-name>
                <method-name>getSavingBalance</method-name>
            </method> (... outros <method> ...)
            <trans-attribute>Required</trans-attribute>
        </container-transaction> (...)
    </assembly-descriptor>
</ejb-jar>
```

O método getSavingsBalance(...) do bean BankEJB tem controle de transações demarcado durante a sua execução usando a política Required

bank-ejb-jar.xml

Transações e entity beans

- Se pudéssemos controlar transações em um Entity Bean, iniciaríamos a transação no ejbLoad() para encerrá-la no ejbStore()
  - Mas quem chama esses métodos é o Container! E o bean não pode garantir que as chamadas ocorram nesta ordem.
  - Por isto, transações gerenciadas no componente são ilegais em Entity Beans: é preciso usar transações declarativas!
- Entity Beans não chamam ejbLoad() e ejbStore() em cada chamada de método
  - Eles fazem isto em cada transação!
  - Logo, a sincronização com o banco e, conseqüentemente, a performance do sistema pode ser melhorada iniciando a transação o quanto antes e terminando-a mais tarde

Qual estilo de demarcação usar?

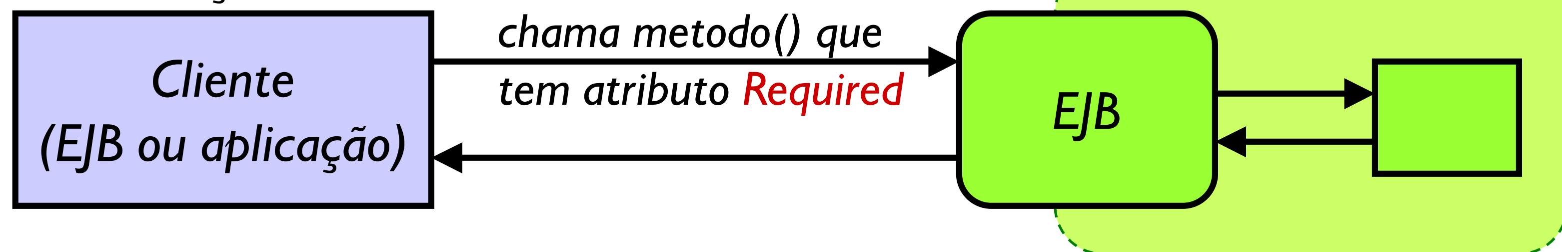
- Vantagens da demarcação explítica (BMT)
  - Maior controle: é possível extrapolar os limites dos métodos (em Stateful Session Beans) ou agrupar conjuntos de instruções menores no interior dos métodos
- Vantagens da demarcação implítica (CMT)
  - Muito mais simples
  - Mais seguro: evita a introdução de código que pode provocar deadlock e outros problemas similares
  - Controle fica com o bean assembler / deployer
  - Mais eficiente: container gerencia melhor os recursos
- Transações iniciadas pelo cliente
  - Vantagem: controle em relação a falhas de rede
  - Desvantagem: transação muito longa - ineficiente

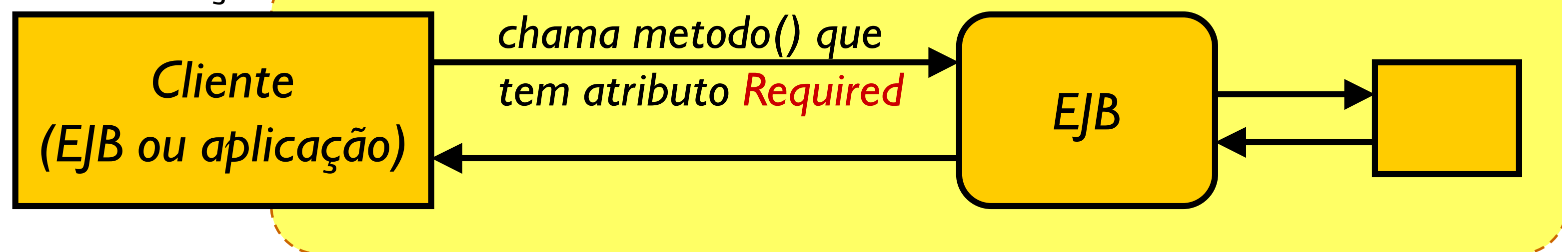
Atributos (políticas transacionais)

- O elemento <trans-attribute> define a política transacional do componente
  - Define como ele irá reagir quando o seu método for chamado por um cliente dentro ou fora do contexto de uma transação
- Os valores suportados para este elemento (depende do tipo de bean) são
  - NotSupported
  - Supports
  - Required
  - RequiresNew
  - Mandatory
  - Never

Required

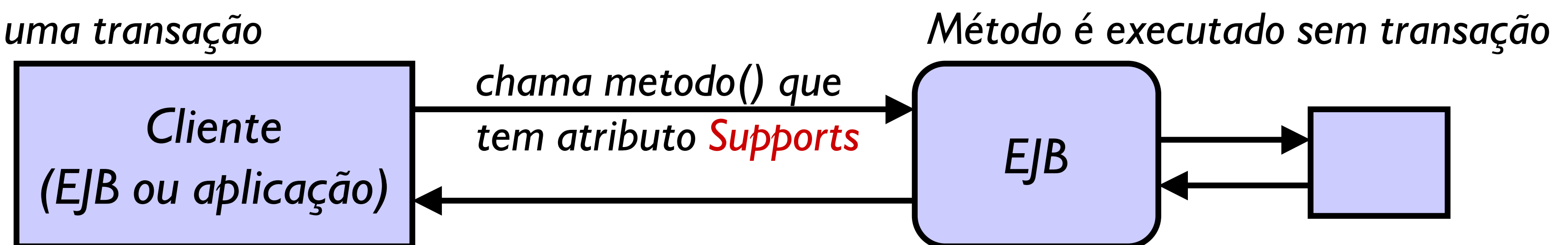
- Indica que o método tem que ser chamado dentro do escopo de uma transação
  - Se não existe transação, uma nova é criada e dura até que o método termine (é propagada para todos os métodos chamados)
  - Se já existe uma transação iniciada pelo cliente, o bean é incluído no seu escopo durante a chamada do método

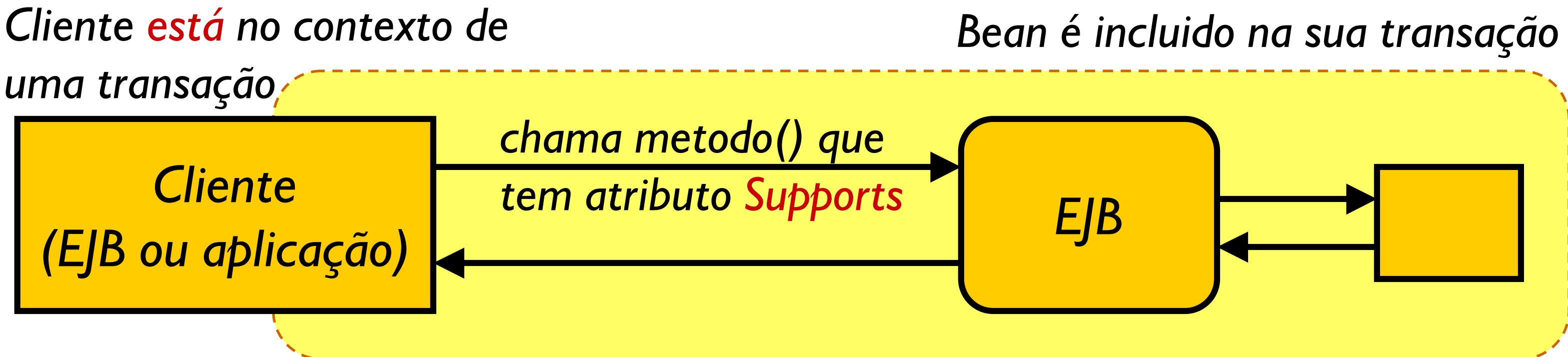




Supports

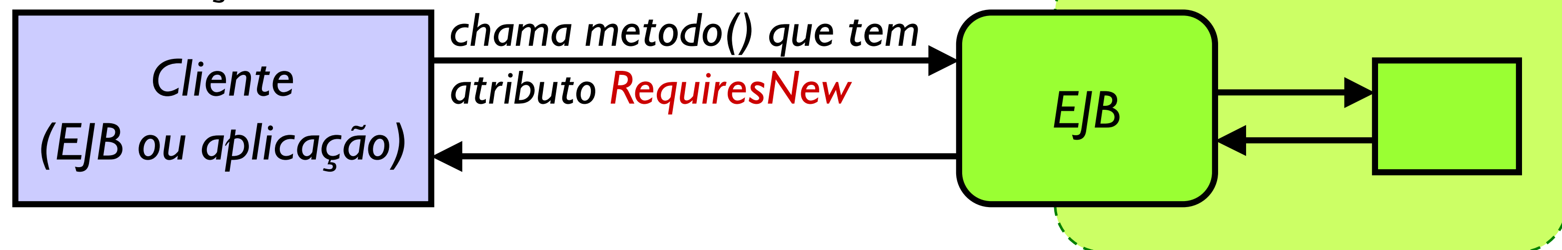
- *Indica que o método suporta transações*
  - *Será incluído no escopo da transação do cliente se existir*
  - *Se ele for chamado fora do escopo de uma transação ele realizará suas tarefa sem transações e pode chamar objetos que não suportam transações*

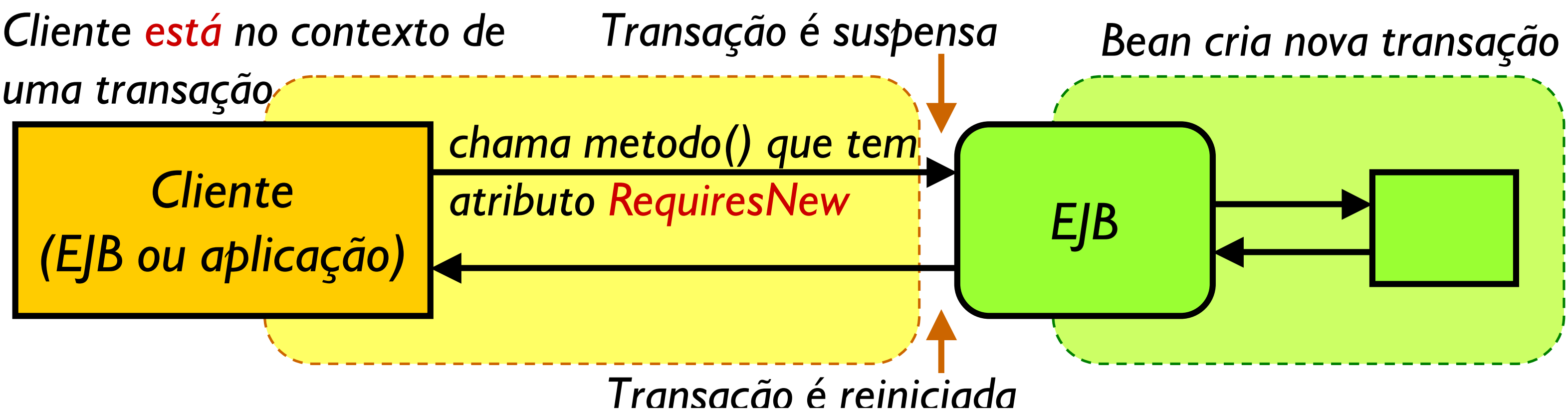




RequiresNew

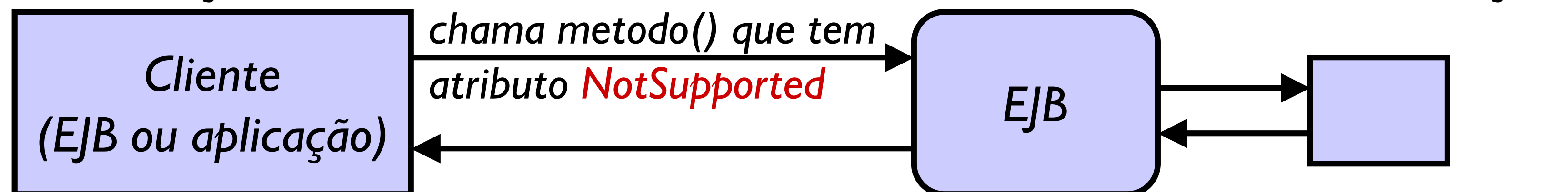
- *Indica que uma nova transação, iniciada no escopo do bean, é sempre criada*
  - *Estando ou não o cliente no escopo de uma transação, o bean irá iniciar uma nova transação que iniciará e terminará no bean.*

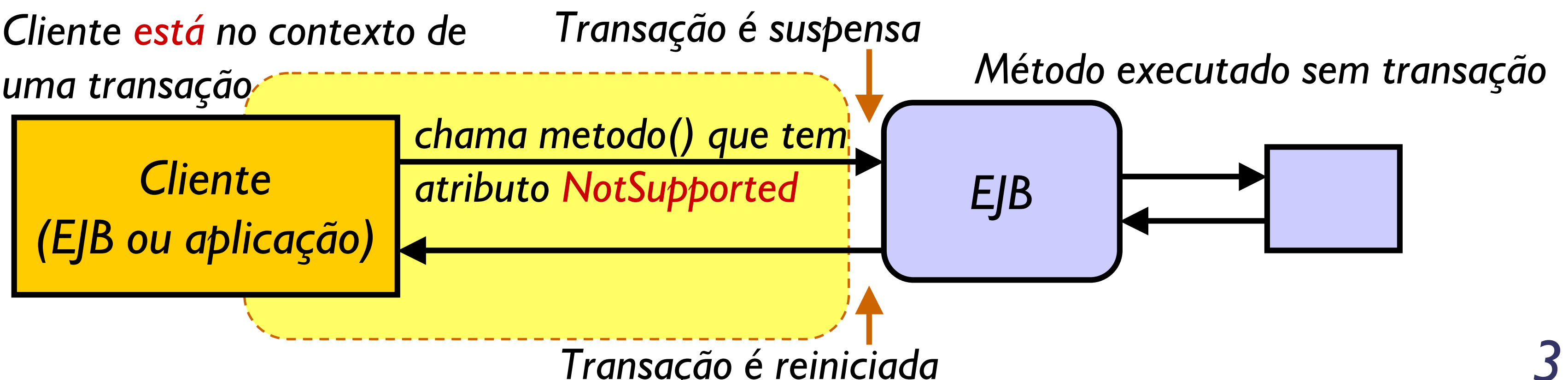




NotSupported

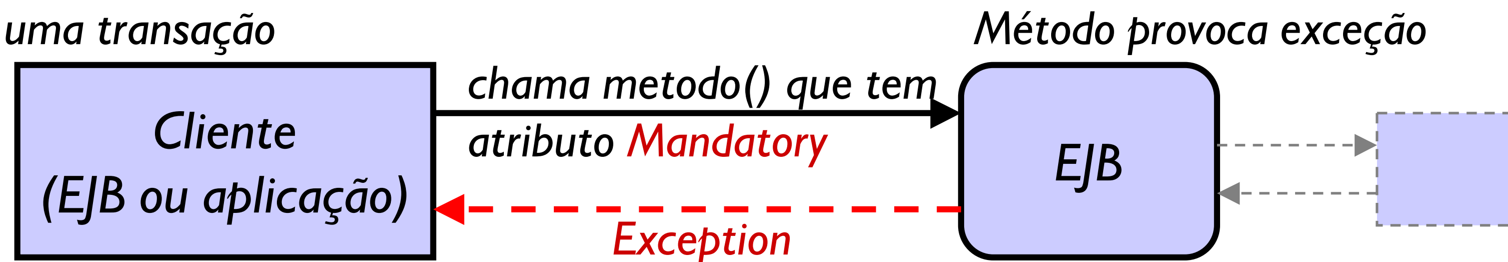
- *Indica que o método não suporta transações*
  - *Se o método for chamado pelo cliente no escopo de uma transação, a mesma será suspensa enquanto durar a chamada do método (não haverá propagação de transações do cliente)*

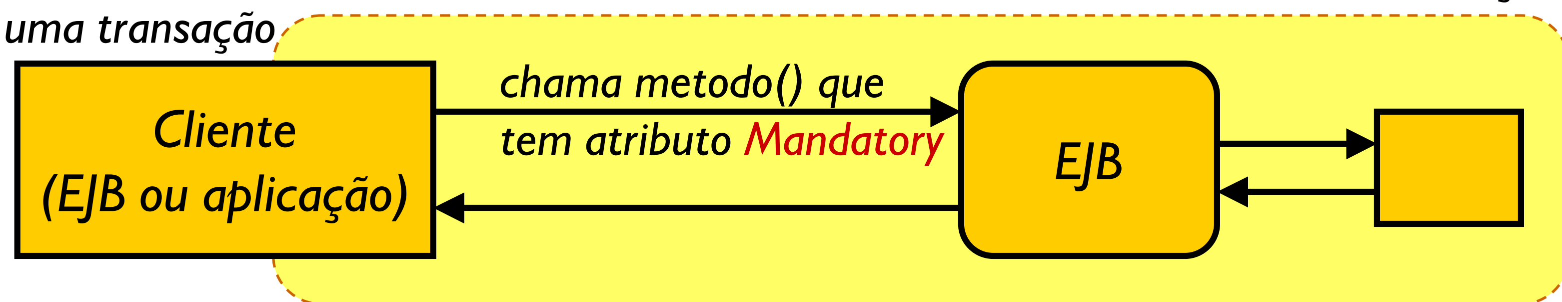




Mandatory

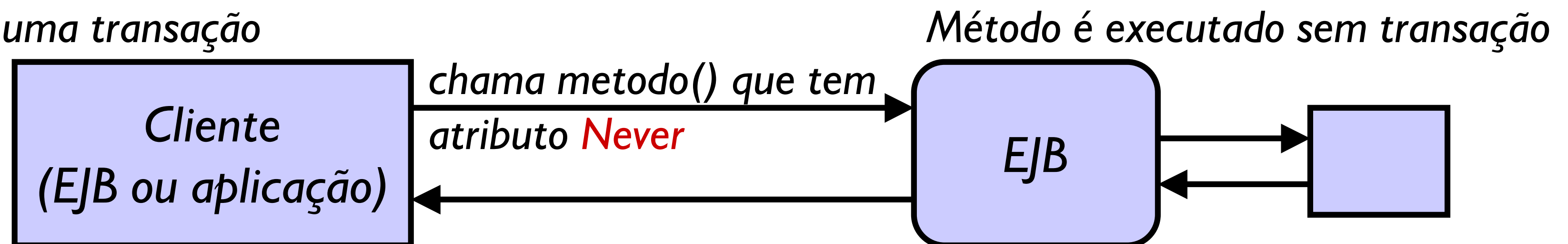
- *Indica que o método só pode ser chamado no escopo de uma transação do cliente*
  - *Se o método for chamado fora de uma transação, ele causará javax.transaction.TransactionRequiredException (ou javax.ejb.TransactionRequiredLocalException)*

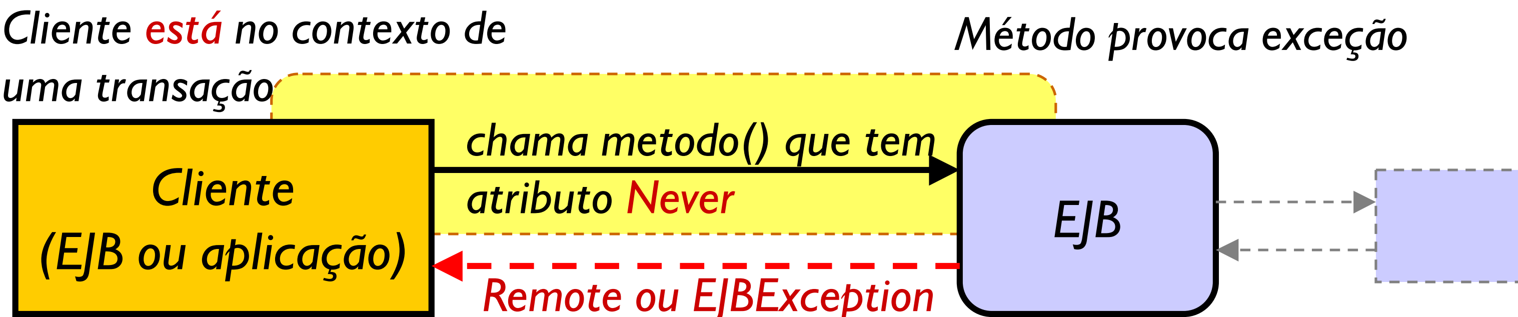




Never

- *Indica que o método nunca pode ser chamado no escopo de uma transação*
  - *Se o cliente que chama o método for parte de uma transação, o bean irá provocar um RemoteException (ou EJBException em clientes locais)*





Atributos suportados por tipo de EJB

- Message Driven Beans usando CMT
  - Apenas os atributos transacionais Required e NotSupported podem ser usados
- Session Beans usando CMT
  - Todos os atributos são suportados
  - Interface Home não deve ter atributos especificados
- Entity Beans (uso de CMT é obrigatório)
  - Todos os atributos são suportados para beans utilizando Bean-Managed Persistence (BMP)
  - Atributos devem ser especificados para métodos da interface Home

Atributos suportados: entity beans com CMP

- Suporte pelo container garantido pela especificação: Required, RequiresNew e Mandatory
- RequiresNew deve ser usado com cautela
  - Força uma nova transação iniciada pelo container, separada da transação do cliente: risco de deadlock se níveis de isolamento mais rígidos forem usados em fonte de dados compartilhada usada pelas duas transações
- Suporte opcional: NotSupported, Supports, Never
  - Suporte visa permitir o uso de CMP com fontes de armazenamento não transacionais
  - Devem ser evitados em pois, além da perda de portabilidade, podem levar a resultados inconsistentes durante o uso por múltiplos clientes simultâneos [2]

Atributos suportados: tipo de session bean

- *Stateful Session Beans que implementam a interface SessionSynchronization*
  - *Suportam apenas Required, RequiresNew ou Mandatory (garante que bean só é chamado em uma transação)*
- *EJB 2.1: Stateless Session Beans usando CMT e funcionando como pontos de acesso para Web Services*
  - *Todos os atributos exceto Mandatory são suportados*
  - *Os atributos devem ser especificados para todos os métodos definidos na interface de acesso do Web Service (endpoint)*

Destino de uma transação em CMT

- *Apenas exceções do sistema (Runtime, Remote, EJBException) provocam automaticamente o rollback de uma transação*
  - *O container não tem como adivinhar que outras exceções sejam graves o suficiente para causar rollback*
- *Pode-se condenar (doom) uma transação CMT usando o método setRollbackOnly() de EJBContext*
  - *O método aciona uma flag usado pelo container para monitorar o estado de uma transação*
  - *Pode-se saber o estado de uma transação usando getRollbackOnly()*

```
try {
   return new ClientePK(clienteDAO.create(clienteDTO));
} catch (UsuarioJaExisteException e) {
   if (!ctx.getRollbackOnly())
      ctx.setRollbackOnly(); // doom this transaction
   throw e;
}
```

Transações em Stateful Session Beans

- Entity Beans mantém seu estado no banco de dados
  - Quaisquer alterações em suas variáveis de instância, durante uma operação, serão revertidas em um rollback
- Já o estado de Stateful Session Beans é mantido em variáveis de instância e não no banco de dados
  - O container não tem como recuperar o estado de suas variáveis em caso de rollback
- É preciso guardar o estado anterior do bean para reverter caso a transação falhe
  - Se o bean inicia a transação (BMT), podemos guardar o estado antes do begin e recuperar após o rollback
  - E se o container iniciar a transação (CMT)?

Interface SessionSynchronization

- *Pode ser implementada por Stateful Session Beans para capturar eventos lançados nos pontos de demarcação*
- *A interface requer a implementação de três métodos*

  `void afterBegin()`:
  - *Chamado logo após o início da transação.*
  - *Neste método, pode-se guardar o estado do bean para recuperação em caso de falha*

  `void beforeCompletion()`:
  - *Chamado antes do commit() ou rollback().*
  - *Geralmente vazio, mas pode ser usado pelo bean para abortar a transação se desejar (usando setRollbackOnly())*

  `void afterCompletion(boolean state)`:
  - *Chamado após o commit() ou rollback().*
  - *Se a transação terminou com sucesso, o parâmetro state é true.*
  - *Caso contrário, é false e neste caso deve-se restaurar o estado do bean aos valores guardados em afterBegin()*

Resumo

- *Atributos CMT de política transacional suportados por tipo de bean*

Atributo	Session Beans			Entity Beans	Message Driven Beans
	Stateful com SessionSynchronization	*Stateful*	*Stateless*		
Required	✓	✓	✓	✓	✓
RequiresNew	✓	✓	✓	✓	✗
Supports	✗	✓	✓	✗	✗
NotSupported	✗	✓	✓	✗	✓
Mandatory	✓	✓	✓ *	✓	✗
Never	✗	✓	✓	✗	✗

Fonte: [2]

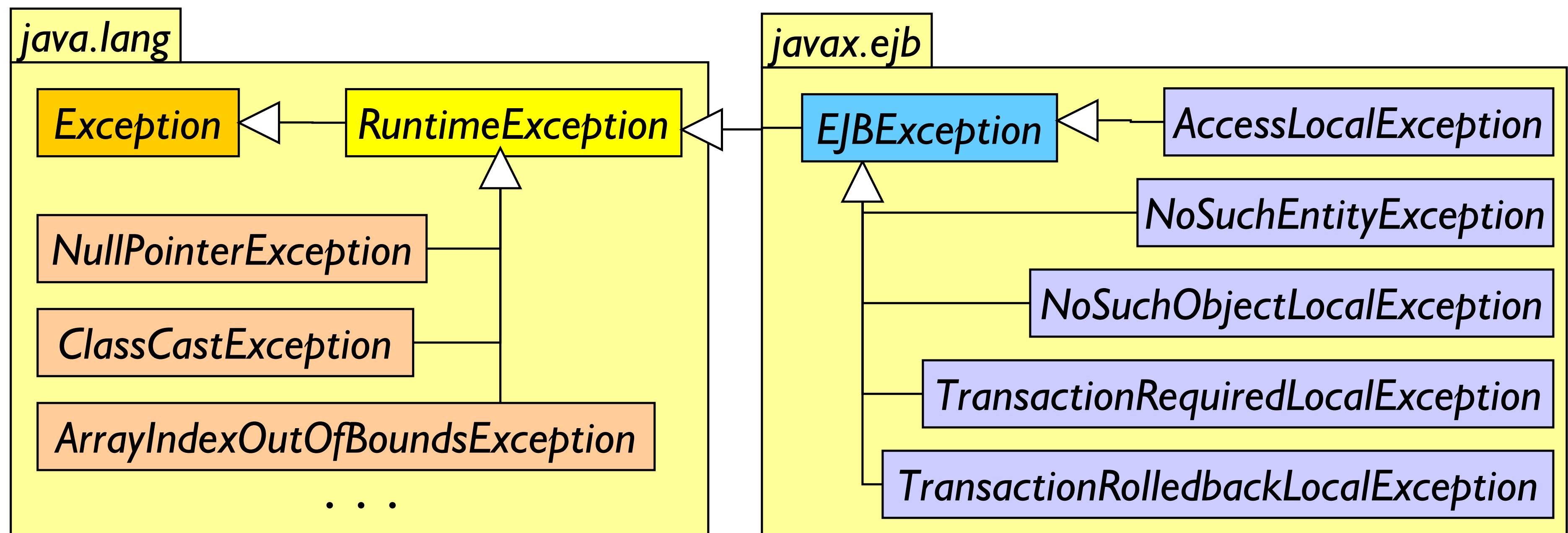
* *Desde que bean não esteja atuando como Web Service Endpoint*

Transações e exceções

- A especificação J2EE classifica exceções em 2 tipos
  - System exceptions
  - Application exceptions
- System exceptions
  - São exceções que indicam erros no servidor e serviços usados pela aplicação
  - São instâncias de RemoteException e RuntimeException
  - Causam rollback automático de transações
- Application exceptions
  - São exceções que indicam erros na lógica da aplicação
  - Consistem de todas as subclasses de Exception exceto as classes que são System Exceptions
  - Por mais graves que sejam, não causam rollback

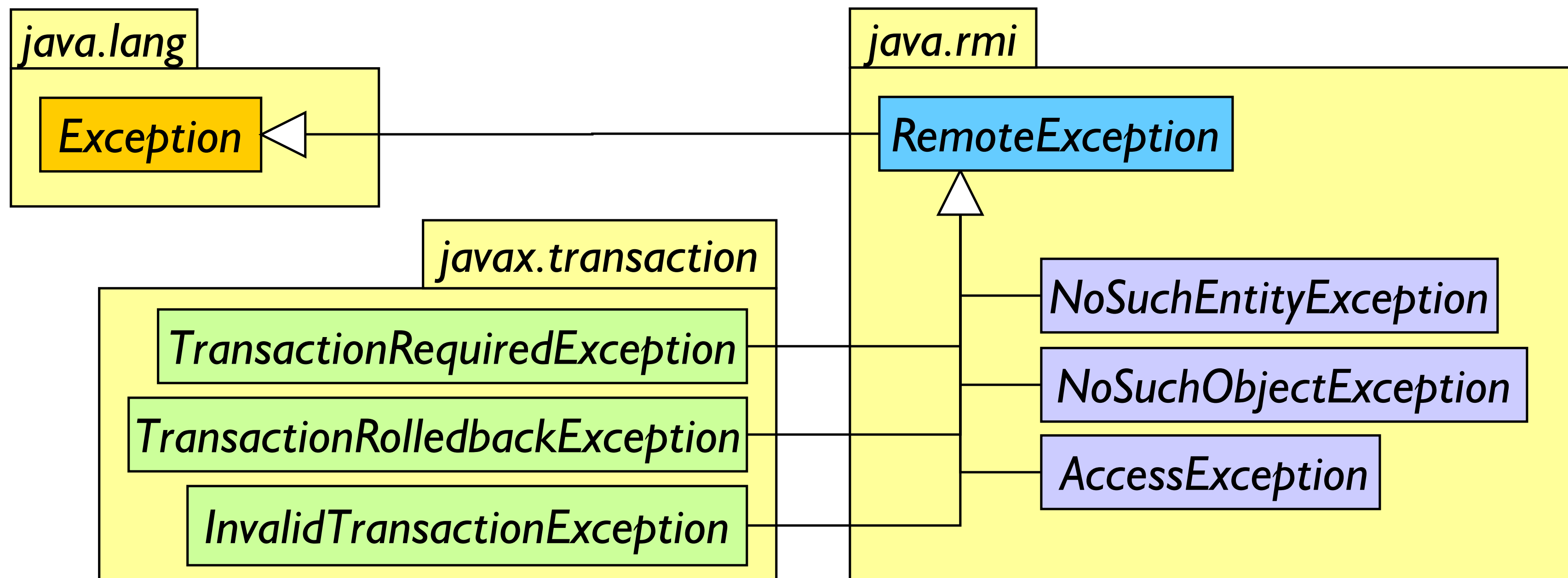
RuntimeException

- Qualquer RuntimeException (NullPointer, IndexOutOfBounds, EJBException, etc.) faz com que o container
  - Realize o rollback da transação
  - Grave um log da exceção
  - Destrua a instância EJB
- Podem ser lançadas de qualquer método, de qualquer tipo de bean (inclusive Message-driven beans)



RemoteException

- *RemoteException nunca é provocado pelo bean, mas apenas pelo container*
  - *É declarado nas interfaces para o objeto remoto, mas não nos métodos do bean*
- *O efeito de um RemoteException em um EJB é o mesmo do RuntimeException*
  - *Rollback da transação, log e destruição do objeto*



Application exceptions

- São sempre devolvidas para o cliente (em vez de serem empacotadas em um Remote ou EJBException
- Uma Application Exception é
  - Qualquer descendente de Exception menos RuntimeException e RemoteException
  - Qualquer exceção específica do domínio da aplicação
  - Exemplos: MinhaException, SuaException, SQLException, JMSException, FileNotFoundException
- Caso uma Application Exception cause um erro grave, deve-se condenar a transação usando ctx.setRollbackOnly()

Application exceptions da API EJB

Os métodos abaixo requerem rollback explícito da transação

- CreateException
  - Provocado pelo método create() na interface remota
- FinderException
  - Indica erro de lógica (argumentos errados, etc.) em métodos findBy()
  - Não use para indicar que entidades não foram achadas[3]. Retorne um Collection vazio, em finders de múltiplas instâncias, ou use ...
- ObjectNotFoundException
  - Usado em finders que localizam única instância para indicar que objeto não foi encontrado
- DuplicateKeyException
  - Subtipo de CreateException usado para indicar que um bean com a mesma chave primária já existe no banco de dados
- RemoveException
  - Indica erro enquanto bean está sendo removido.

Níveis de isolamento

- Isolamento é uma das características ACID
  - Isolamento é alcançado bloqueando acesso simultâneo de usuários a todo ou parte de um recurso
  - 100% de isolamento geralmente é ineficiente
  - É possível flexibilizar o isolamento em certas tarefas, sem riscos de perder dados e causar dados aos dados
  - Um bom projeto precisa determinar o nível mínimo de isolamento necessário para garantir a integridade dos dados sem reduzir demasiadamente a performance
- O gerenciamento de níveis de isolamento não é definido na especificação EJB 2.0
  - A definição desses níveis depende da API do gerenciador do recurso (pode ser controlado através de JDBC)

Problemas de falta de isolamento

- Dependendo do nível de flexibilização do isolamento entre transações, vários problemas podem surgir
- Dirty read
  - É o mais grave dos problemas de isolamento. Ocorre quanto uma transação lê valores não cometidos, gravados no banco por uma transação que ainda não terminou. Se a primeira transação falhar, a segunda estará com dados incorretos.
- Unrepeatable read
  - Ocorre quando duas leituras consecutivas do mesmo registro retorna valores diferentes, devido à alteração de dados por outra transação.
- Phantom read
  - Ocorre quando duas leituras consecutivas da mesma tabela retorna valores diferentes, devido à inserção de dados por outra transação.

Niveis de isolamento

- *Há quatro níveis de isolamento de transações*
- *Read Uncommitted:*
  - *Não oferece garantia de isolamento (aceita dirty reads - cliente pode ler dados ainda não cometidos) mas oferece a maior performance.*
- *Read Committed*
  - *Resolve o os dirty reads mas não garante que leituras consecutivas retornem os mesmos resultados (aceita unrepeatable reads)*
- *Repeatable Read*
  - *Bloqueia o acesso aos dados utilizados durante a consulta impedindo unrepeatable reads, mas não impede que novos dados apareçam no banco durante a operação (aceita phantom reads)*
- *Serializable*
  - *Bloqueia o acesso concorrente aos dados (transações ocorrem em série) durante o uso mas baixa a performance significativamente.*

Como definir o nível de isolamento

- *Use as ferramentas do seu servidor ou banco de dados*
  - *Não há como especificar níveis de isolamento no DD*
  - *Para Entity Beans CMP, a única forma de definir o nível de isolamento é usando configuração proprietária do container*
- *Em session, message-driven e entity beans com BMP, pode-se utilizar as constantes de java.sql.Connection*

  TRANSACTION_READ_UNCOMMITTED,
  TRANSACTION_COMMITTED,
  TRANSACTION_REPEATABLE_READ *ou*
  TRANSACTION_SERIALIZABLE

  como argumento do método setTransactionIsolation()
  - *Mudar o nível de isolamento de conexões individuais pode ser problemático já que as conexões geralmente ficam em pools onde são utilizadas por várias aplicações*
  - *Nem todos os níveis de isolamento são suportados por todos os fabricantes. Alguns suportam apenas um ou dois.*

Nível de isolamento no JBoss

- No JBoss, o nível de isolamento default pode ser configurado no arquivo *-ds.xml de cada data source instalada.
  - Esse recurso depende de suporte por parte do gerenciador de banco de dados e a sintaxe varia. Exemplo:

```
<transaction-isolation>
    TRANSACTION_COMMITTED
</transaction-isolation>
```

  - O banco deve suportar o nível de isolamento escolhido
  - O nível de isolamento por conexão também pode ser alterado via JDBC usando setTransactionIsolation() (se o banco suportar)
- O banco nativo HSQLDB, não suporta bloqueio de registros. Seu nível de isolamento é sempre Read Uncommitted
  - O valor retornado por getTransactionIsolation() (de Connection) é sempre TRANSACTION_READ_UNCOMMITTED e qualquer outro valor passado para setTransactionIsolation() causa exceção.

Controle de concorrência

- *Há duas estratégias para buscar um equilíbrio razoável entre isolamento e performance*
  - *Controle de concorrência pessimista*
  - *Controle de concorrência otimista*
- *Na estratégia pessimista, o EJB tem acesso exclusivo aos dados durante toda a duração da transação*
  - *Garante acesso confiável aos dados*
  - *Razoável em sistemas onde acesso simultâneo é raro*
  - *Pouco escalável*
- *Na estratégia otimista, O EJB aceita compartilhar os dados com outros objetos, e torce para que nada falhe*
  - *Se o banco detectar uma colisão, a transação é desfeita*
  - *Assume que vale mais a pena lidar com eventuais colisões que limitar a escalabilidade do sistema*
  - *Requer escrever código para lidar com gerenciamento de colisões*

Controle de concorrência otimista

- Estratégia mais flexível. Permite ajustar nível de bloqueiode acordo com a necessidade de performance/segurança
- Requer que algumas medidas sejam tomadas para que os dados não sejam corrompidos: detecção de colisões
- Há várias formas de implementar detecção de colisões
  - Usando o banco: Se o banco implementa o nível de isolamento SERIALIZABLE, um acesso concorrente irá provocar uma exceção, que pode ser usada para lidar com o problema
  - Usando flags: Coluna extra em cada tabela (e como conseqüência um campo extra em cada bean). Valor é incrementado a cada uso. A cada leitura/gravação o número é conferido e se for diferente, outro cliente manipulou com os dados (houve colisão)
- A primeira alternativa é problemática e não portável
- A segunda alternativa é mais indicada (funciona com CMP e no HQSLDB) mas requer a alteração das tabelas

Two-phase commit (XA)

- Containers EJB suportam transações distribuídas. O suporte é implementado usando um protocolo chamado de Two-phase commit, que realiza a transação em duas fases
- Na primeira fase, o servidor envia uma mensagem para todos os recursos envolvidos na transação (before commit),
  - É uma oportunidade para que abortem a transação. Se qualquer recurso envolvido decidir abortar, a transação inteira é cancelada
  - Se ninguém abortar, a transação continua e não pode mais parar, a não ser que haja uma falha grave
  - Para evitar perda de dados no caso de uma falha grave, as atualizações são gravados em um log persistente que sobrevive a falhas, para que possa ser revertido
- A segunda fase só ocorre se a fase um completar, e é quando todos os gerenciadores de recursos envolvidos realizam as atualizações

Conclusões

- O uso de transações tem um custo
  - Impacto (positivo ou negativo) na performance
  - Risco de deadlock
  - Aumento da complexidade e possibilidade de ter que lidar com questões não portáveis como isolamento, concorrência, etc.
- Nem sempre um método precisa estar em uma transação
  - Utilize transações apenas onde realmente for necessário
- Para melhores resultados, use CMT sempre que possível
  - Riscos de deadlock e complexidade são muito menores
  - CMT suporta melhor a separação de papéis (deployer pode ajustar forma como transações serão aplicadas)
- Transações podem melhorar a performance de aplicações EJB
  - A sincronização de EJBs ocorre uma vez por transação: inicie a transação fora dos entity beans (no Session Façade).
  - Evite iniciar longas transações (principalmente se a partir de clientes fora do container)

Exercícios opcionais

1. O Session Bean CriaLivro possui três métodos necessários para criar um Entity Bean com os valores por eles retornados.
   - Eles são chamados pelo método criarLivro() do session bean Biblioteca. Cada um, localiza um Entity Bean, guarda os dados lidos, destrói o bean e devolve os dados (String).
   - O método criarLivro() cria um novo bean (Livro) com os dados
   - Há 20% de chance de qualquer um dos métodos de CriaLivro gerar uma exceção, o que deve reverter todo o processo.
     - a) Execute a aplicação algumas vezes e veja os resultados no banco
     - b) Defina um controle de transações declarativa para que a aplicação só remova os beans se tiver criado um Livro
2. (opcional) Inicie uma transação a partir do cliente
   - a) Chame os três métodos em uma UserTransaction no cliente
   - b) Chame criarLivro() do cliente e experimente alterar os atributos transacionais do método no DD

Fontes

[1] *Ed Roman et al. Mastering Enteprise JavaBeans, 2nd. Edition, Chapter 10: Transactions. John Wiley & Sons, 2002.*

[2] *Linda de Michiel et al. Enterprise JavaBeans 2.1 Specification, Chapter 17: Support for Transactions. Sun Microsystems, 2003.*

[3] *Richard Monson-Haefel. Enterprise JavaBeans, 3rd. Edition. O'Reilly and Associates, 2001*

[4] *Dale Green. J2EE Tutorial: Transactions. Sun J2EE Tutorial, Sun Microsystems, 2002*

[5] *Jim Farley et al. Java Enterprise in a Nutshell, 2nd. Edition. O'Reilly & Associates 2002.*

[6] *Susan Cheung. Java Transaction Service Specification. Sun Microsystems, 1999*

[7] *Susan Cheung et al. Java Transaction API Specification 1.01B. Sun Microsystems, 2002*

[8] *Oracle. Design Considerations when using Entity Beans with CMP. 2003. http://otn.oracle.com*

Curso J530: Enterprise JavaBeans

Revisão 2.0

Transações. Rev. 2.0 (junho 2003)

© 2001-2003, Helder da Rocha

(helder@acm.org)