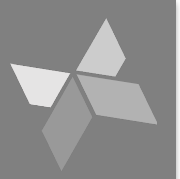
EJB
enterprise javabeans

Helder da Rocha (helder@acm.org)

Atualizado em maio de 2015

argonavis
tecnologia e arte

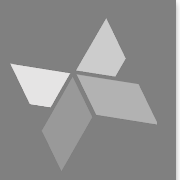
JAVA EE 7



Sobre este tutorial

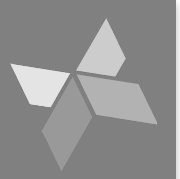
Este é um tutorial sobre tecnologia EJB (de acordo com a especificação Java EE 7) criado para cursos presenciais e práticos de programação por Helder da Rocha

O código dos exemplos está disponível em repositório público no GitHub e é software livre sob os termos da licença Apache 2.0.



Conteúdo

1. Introdução
2. Session Beans
3. Acesso local e remoto
4. Message-Driven Beans
5. Concorrência
6. Transações
7. Segurança

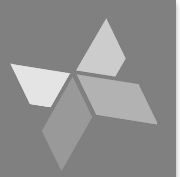


JAVAEE 7

EJB
enterprise javabeans

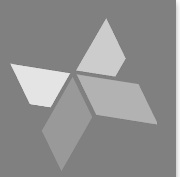
1

introdução



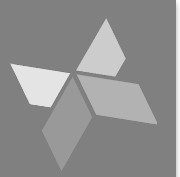
Enterprise JavaBeans

- **Enterprise JavaBeans** são **componentes** que encapsulam a lógica de negócios de uma aplicação.
- Implementam **serviços síncronos** e **assíncronos**, podem exportar uma interface para clientes **locais**, mas também para clientes **remotos** usando protocolos de objetos remotos (RMI/IIOP) ou Web Services (SOAP)
- Instâncias de EJBs têm o **ciclo de vida** controlado em tempo de execução pelo container no qual ele está instalado, que também disponibiliza o acesso a **serviços configuráveis** de segurança, transações, agendamento, concorrência, sincronização, distribuição e outros de forma **transparente**



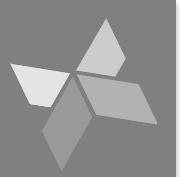
Enterprise JavaBeans

- Enterprise JavaBeans são **JavaBeans**, ou **POJOs**: objeto Java com construtor default, atributos encapsulados acessíveis via getters/setters
- Existem dois tipos de EJBs:
  - **Session beans**
  - **Message-driven beans**
- Em aplicações Java EE, EJBs são parte da **camada de negócios** e geralmente acessados por clientes locais ou remotos de uma mesma aplicação distribuída.
- EJBs podem ser **fachadas de serviços** para a **camada de dados**



Tipos de enteprise beans

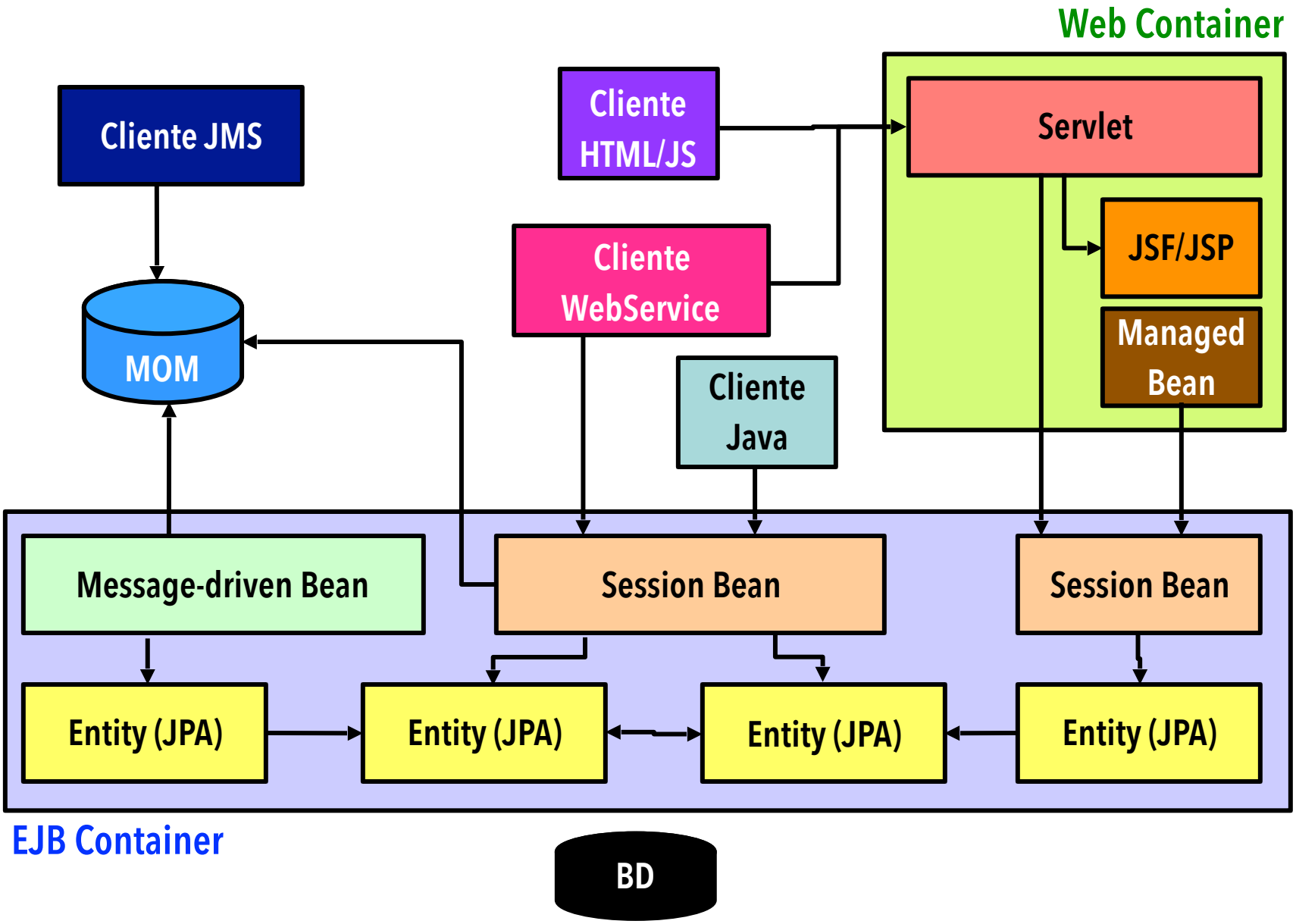
- **Session Beans**
  - Modelam **processos de negócio**. São **ações**, **verbos**.
  - Fazem coisas: acessam um banco, fazem contas,
  - Podem manter ou não **estado não-persistente**
  - Processar informação, comprar produto, validar cartão
- **Message-driven beans**
  - Modelam **processos assíncronos**. Respondem a **eventos**.
  - Agem somente quando recebem uma mensagem
  - **Não mantêm estado**

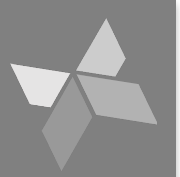


Persistent entities

- **Persistent entities**
  - Não são Enterprise JavaBeans
  - Modelam **dados** de negócio. São **coisas**, **substantivos**.
  - Um produto, um empregado, um pedido
  - Mantêm **estado persistente**
  - São mapeados a tabelas usando **ORM** (JPA em Java EE)
  - Geralmente **inacessíveis** a clientes externos: o acesso ocorre através de **fachadas** de serviços síncronos (Session Beans) e assíncronos (filas JMS e Message-Driven Beans)

Uso típico de componentes Java EE





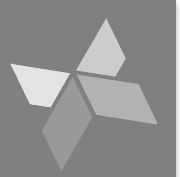
JAVAEE 7

EJB

enterprise javabeans

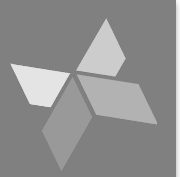
2

session beans



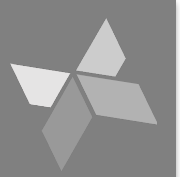
Session Beans

- Componentes que exportam uma **interface de serviços**
  - **Interface local** (acessível dentro da mesma aplicação WAR/EJB/EAR)
  - **Interface remota** (acessível de outras aplicações ou JVMs) através de protocolo RMI/IIOP ou SOAP
- Beans têm o ciclo de vida controlado pelo container (que cria e destrói os beans) e são acessíveis a clientes apenas via **proxies** que são **injetados** nos clientes pelo container ou **localizados** via registros JNDI
- Modelam uma sessão (**diálogo**) de comunicação com um **cliente** através da chamada de seus métodos



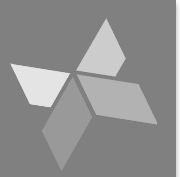
Tipos de Session Beans

- **Stateful Session Beans**
  - Modelam diálogos com um ou mais requisições que podem depender de requisições anteriores. **Preservam estado** do cliente **entre chamadas**.
- **Stateless Session Beans**
  - Modelam diálogos com apenas uma requisição ou várias requisições independentes. **Não preservam estado** do cliente **entre chamadas**.
- **Singleton Session Beans**
  - Modelam diálogos entre vários clientes e a aplicação. São criados uma vez por aplicação. **Compartilham estado** da aplicação **entre clientes**.



Interfaces para Session Beans

- Um bean é acessível apenas através de sua **interface**. Há 4 tipos:
  - **No-Interface View**: acessível localmente a clientes da mesma aplicação. A interface de serviços inclui todos os seus métodos (inclusive herdados).
  - **Interface local usando @Local**: bean implementa uma ou mais interfaces com métodos que são acessíveis por clientes na mesma aplicação e JVM.
  - **Interface remota (para clientes IIOP) usando @Remote**: bean implementa uma ou mais interfaces com métodos acessíveis por clientes locais ou de outras aplicações, servidores ou JVMs.
  - **Interface remota (para clientes SOAP) usando @WebService**: bean implementa interface que declara métodos acessíveis via WSDL/SOAP ou declara no bean (com @WebMethod) cada método da interface WSDL/SOAP



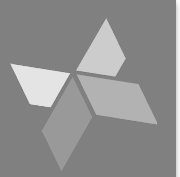
Stateless Session Beans (SLSB)

- A anotação **@Stateless** cria um Stateless Session Bean com uma **no-interface view** (interface local contendo **todos** os métodos do bean)

```
@Stateless
public class VitrineVirtual {
    public List<Produto> listarTodos() { ... }
    public CestaDeCompras getCesta() { ... }
}
```

- Um componente que tiver o bean injetado pode chamar seus métodos.

```
@EJB VitrineVirtual vitrine;
...
public void metodo() {
    for(Produto p : vitrine.listarTodos())
        ...
}
```



Stateful Session Beans (SFSB)

- A anotação **@Stateful** cria um Stateful Session Bean com uma **no-interface view**:

```
@Stateful
public class CestaDeCompras {
    public void adicionar(Produto p) { ... }
    public void remover(Produto p) { ... }
}
```

- Ele pode ser injetado em um cliente da mesma forma que o SLSB:

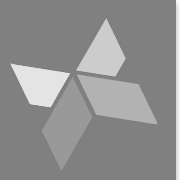
```
@EJB CestaDeCompras cesta;
...
public void metodo(Produto produto, int quantidade) {
    for(int i = 0; i < quantidade; i++)
        cesta.adicionar(produto);
    System.out.println(cesta.toString()); // herdado
}
```

Pool de Session Beans

- Session Beans podem ser guardados em um **pool**, onde são **pré-instanciados** pelo container, e reutilizados por diferentes clientes
- Em um pool de **Stateless Session Beans** todas as instâncias da mesma classe são **equivalentes e indistinguíveis**.
  - Qualquer instância pode servir a qualquer cliente
- Como **Stateful Session Beans** mantém estado para cada cliente, o container não pode fazer o mesmo tipo de pooling já que **beans não são equivalentes**.
  - Se um pool tem mais clientes que beans, o container otimiza o uso de recursos através de um mecanismo de **ativação e passivação**

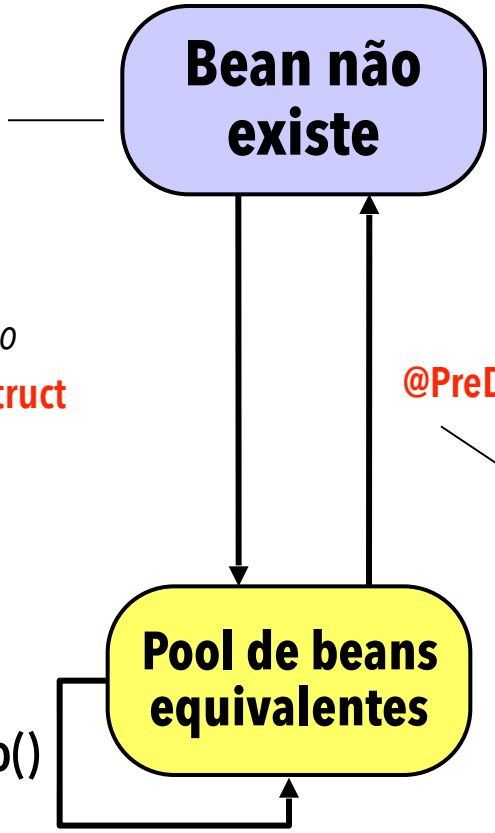
Passivação e ativação

- Um **pool** de Stateful Session Beans mantém várias instâncias ativas
- **Passivação**: se chegar um novo cliente e não houver mais beans no pool, o bean **que foi acessado há mais tempo** terá o estado da sua sessão gravado em meio persistente, e a instância irá para o cliente novo.
- **Ativação**: se esse cliente continua um **diálogo iniciado anteriormente** e que foi passivado, o estado da sessão é obtido do meio persistente e restaurado na instância.
- Antes da passivação, o container chama qualquer método anotado com **@PrePassivate**. Logo após a ativação, ele chama **@PostActivate**. Esses métodos podem ser usados para lidar com estados não serializáveis



Ciclo de vida: Stateless Session Bean

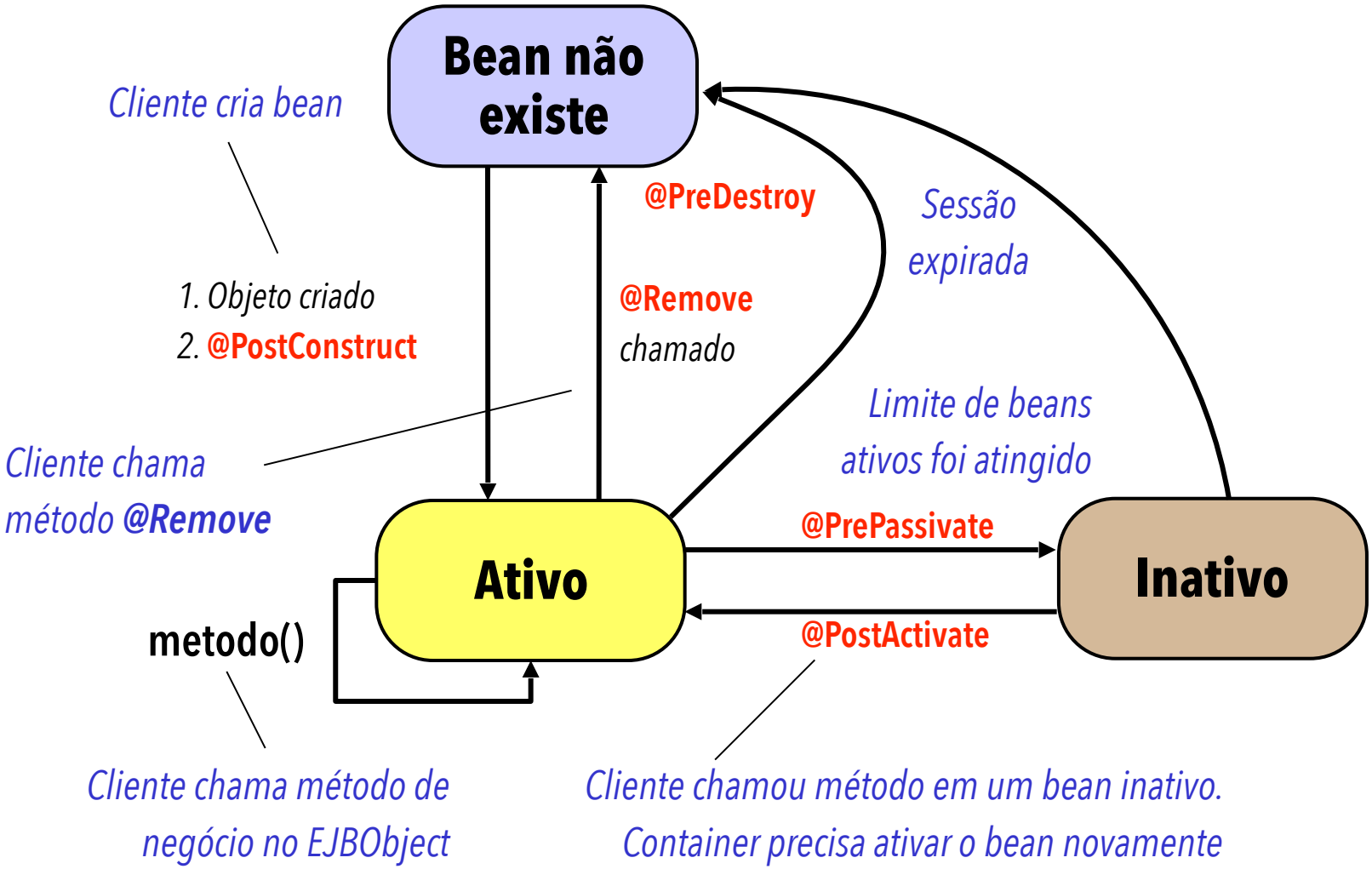
Container cria um novo bean quando ele acha que precisa de mais beans no pool para servir à demanda dos clientes

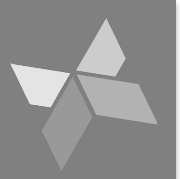


*Quando o container decidir que não precisa mais de tantas instâncias, chama método marcado com **@PreDestroy** e remove a instância*

Qualquer cliente pode chamar um método de negócio em qualquer EJBObject

Ciclo de vida: Stateful Session Bean



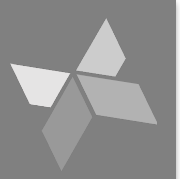


Encerramento da sessão (diálogo)

- Em um **Stateless Bean** ela inicia com a chamada de **um método** e termina quando esse método termina
- Em um **Stateful Bean** ela tem tempo indeterminado, já que o estado do cliente é mantido por **varias chamadas**
  - Se o cliente decidir que diálogo terminou, poderá remover o bean. Para isto é necessário anotar um método do bean com **@Remove**

```
@Remove public void remover() {}
```

  - O método pode ser vazio. O bean é removido quando o método termina

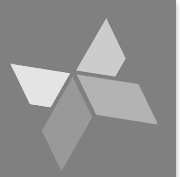


Singleton Session Beans

- A anotação **@Singleton** declara um Singleton Session Bean com uma **no-interface view**:

```
@Singleton
public class ServicosGlobais { ... }
```

- É um tipo especial de Stateless Session Bean que não mantém estado por cliente (é stateless) mas pode ter estado **compartilhado** por toda a aplicação
- É instanciado **uma única vez por aplicação** (mas se uma aplicação for distribuída por várias JVMs, haverá uma instância em cada)
- Singletons tem o **escopo da aplicação**. Seu estado é compartilhado mas seus métodos são **thread-safe** por default (acesso exclusivo)



Startup Singleton Beans

- É comum instruir o container a carregar o bean durante a inicialização da aplicação anotando-o com **@Startup**:

```
@Startup @Singleton
public class ServicosGlobais { ... }
```

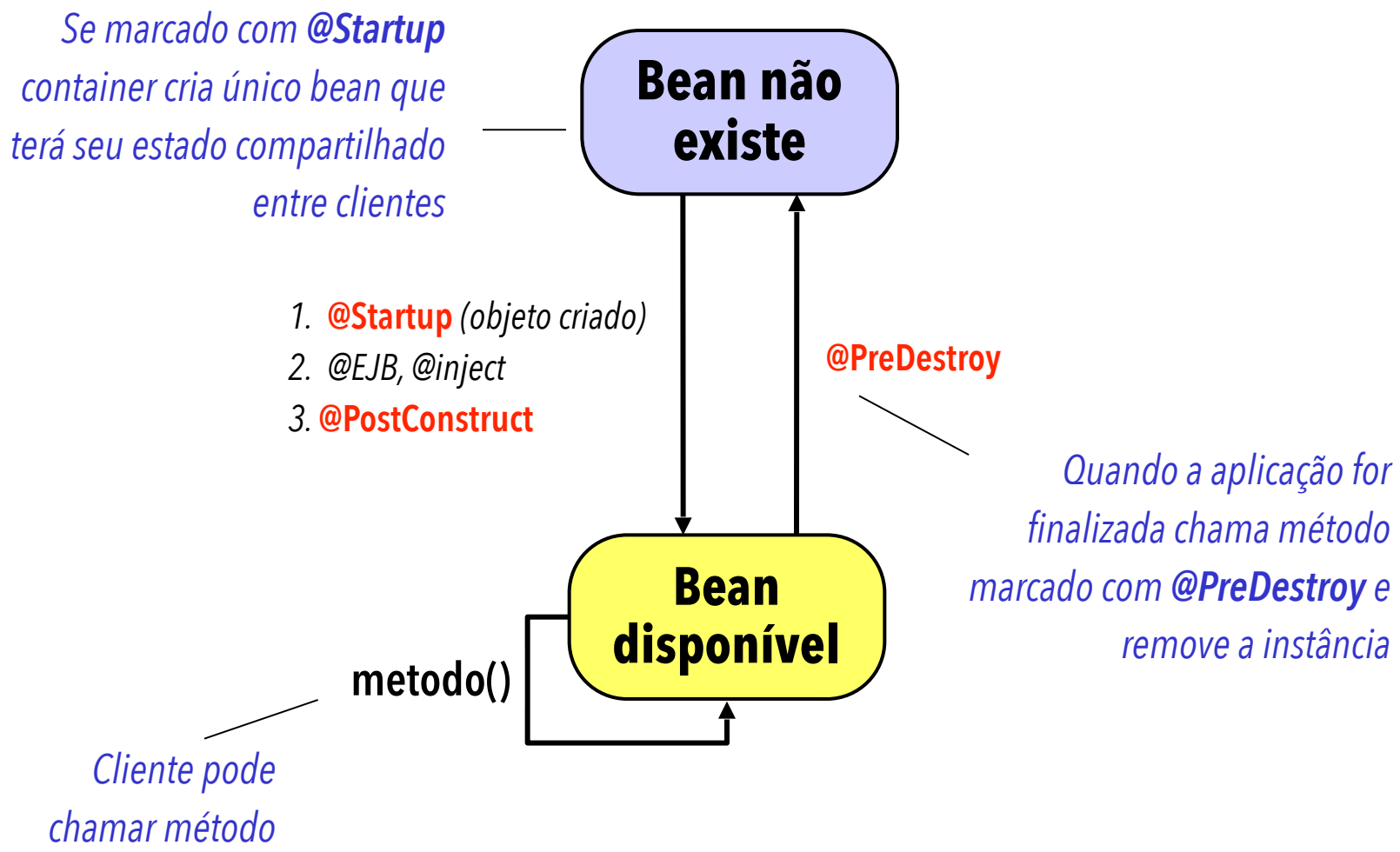
- Às vezes um singleton depende da inicialização prévia de outros singletons. Isto pode ser configurado usando **@DependsOn**:

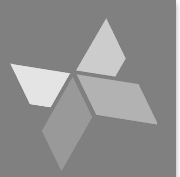
```
@Singleton @DependsOn("ServicosGlobais")
public class EstatisticasDeAcesso { ... }
```

- Pode-se fornecer uma lista de beans a inicializar previamente

```
@Singleton @DependsOn({"Configuracao", "Inicializacao"})
public class Aplicacao { ... }
```

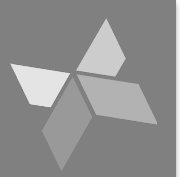
Ciclo de vida: Singleton Session Bean





Callbacks do ciclo de vida

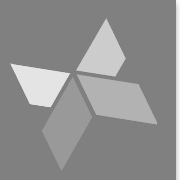
- Há quatro anotações EJB para **métodos de callback**, chamados **automaticamente** em eventos do ciclo de vida:
  - **@PostConstruct** (todos os beans: SB, MDB)
  - **@PreDestroy** (todos os beans: SB, MDB)
  - **@PrePassivate** (apenas Stateful Session Beans)
  - **@PostActivate** (apenas Stateful Session Beans)
- Outras anotações para métodos que influenciam o ciclo de vida (somente Session Beans)
  - **@Remove** (em Stateful Session Beans: permite remoção pelo cliente)
  - **@AroundConstruct** usada em interceptadores



@PostConstruct

- Usada para anotar um método que deve ser chamado
  - **depois** da execução do construtor default e injeção de dependências
  - **antes** de qualquer método da interface pública do bean

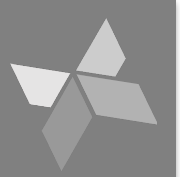
```
@Stateless
public class Biblioteca {

    @Inject ServicoIsbn servico;

    @PostConstruct
    private void inicializar() {
        servico.login();
    }
    ...
}
```



@PreDestroy

- Anota método que deve ser chamado **antes** do bean ser removido
- O bean pode ser removido explicitamente pelo **cliente** (via método anotado com **@Remove** em SFSB) ou pelo **container** (a qualquer momento para SLSB), ou na **finalização** da aplicação (singletons).

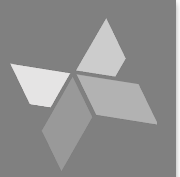
```
@Stateless
public class Biblioteca {
    ...
    @PreDestroy
    private void finalizar() {
        servico.logout();
    }
    ...
}
```



@PrePassivate

- Usada apenas em stateful session beans e chamado **antes** que o bean entre no estado passivo

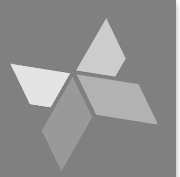
```
@Stateful
public class CestaDeCompras implements Serializable {
    private transient String codigoAcesso;
    @Inject private Cliente cliente;

    @PrePassivate
    private void passivar(InvocationContext ctx) {
        cliente.liberarCodigoAcesso();
    }
    ...
}
```



@PostActivate

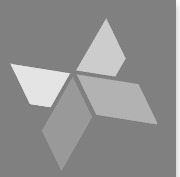
- Usada apenas em stateful session beans e chamado **depois** que o bean é reativado do seu estado passivo

```
@Stateful
public class CestaDeCompras implements Serializable {
    private transient String codigoAcesso;
    @Inject private Cliente cliente;

    @PostActivate
    private void ativar(InvocationContext ctx) {
        codigoAcesso = cliente.obterNovoCodigoAcesso();
    }
    ...
}
```



Exercícios

- 1. Crie um **Stateless bean** simples com os seguintes métodos:
  - **String caixaAlta(String texto)** - que põe o texto recebido em caixa-alta
  - **List<Integer> contagem(int limite)** - retorna sequência de números até o limite
- 2. Crie um cliente Web na mesma aplicação (um servlet, um JSP, um managed bean + JSF) que chame esses métodos
- 3. Crie um **Stateful bean** simples com os seguintes métodos:
  - **void incrementar()** - adiciona um
  - **void decrementar()** - remove um
  - **int resultado()** - retorna o total
- 4. Crie um cliente Web que injete três diferentes instâncias do bean, e chame vários desses métodos imprimindo o resultado no final
- 5. Transforme o bean do exercício 3 em um **Singleton** e execute o cliente novamente



JAVAEE 7

EJB

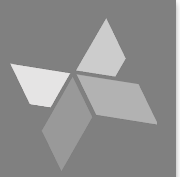
enterprise javabeans

3

interfaces locais e remotas

Localização de recursos

- O instanciamento de recursos (conexões, beans e outros componentes) é realizada automaticamente pelo container que gerencia seu ciclo de vida.
- Clientes precisam de **proxies** e **interfaces** para acessar os recursos
- Clientes locais **declaram dependências** que são **injetadas** automaticamente pelo container (com **DI** ou **CDI**), e clientes remotos **localizam explicitamente** dependências via **JNDI**
- Um nome **JNDI** pode ser especificado com escopo **global** ou no contexto de uma **aplicação** (EAR), **módulo** (WAR, JAR) ou **componente** (Bean)
- **DI** é acionado através de anotações **@EJB** e **@Resource**. **CDI** utiliza a anotação **@Inject**



JNDI Global

- Um nome JNDI **global** tem a forma **mínima**:

  java:global/aplicacao/bean

- Para beans armazenados em EJB JARs ou WARs. A **aplicação** é (por default) o nome do arquivo sem a extensão (aplicacao.war ou aplicacao.jar)

- Se o WAR ou EJB JAR estiver dentro de um EAR, ele é um **módulo** da aplicação EAR. Nesse caso, o EAR é incluído no caminho global:

  java:global/**aplicacao**/**modulo**/bean

- O nome da aplicação é o nome do EAR (ex: aplicacao.ear) e os módulos podem ser o nome do arquivo (default) ou declarados em **/META-INF/application.xml**.

- Se o bean tiver mais de uma interface e for necessário identificar uma delas, o nome (qualificado) da interface pode ser incluído depois do bean:

  java:global/aplicacao/modulo/bean**!pacote.Interface**

JNDI local a um contexto

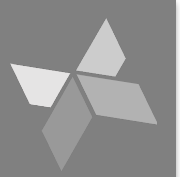
- JNDI relativo pode ter seu escopo limitado no escopo do componente, módulo / aplicação ou container
- Se o acesso JNDI ocorrer entre módulos da mesma **aplicação** (EAR), pode-se usar o acesso no escopo do container com prefixo java:app:

  java:app/modulo/bean

- Se a comunicação ocorrer no contexto do **módulo**, ou se a aplicação não estiver em um EAR, pode-se usar qualquer uma das duas formas:

  java:app/bean
  java:module/bean

- Pode-se também usar JNDI para acessar propriedades de um **componente** (declaradas em web.xml ou anotações) usando **java:comp/propriedade**



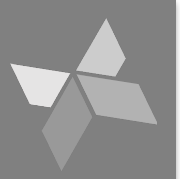
JNDI: exemplos

- Um bean **@Remote LivroBean** empacotado em **biblioteca.war** pode ser chamado localmente (ex: por um managed bean ou servlet no mesmo WAR) usando:

```
public class LivroClientServlet extends HttpServlet {
    public void init() {
        try {
            Context ctx = new InitialContext(); // inicialização do JNDI
            LivroBean bean = (LivroBean)ctx.lookup("java:app/LivroBean");
            bean.inicializar();
            ...
```

- E por um cliente remoto usando:

```
public class LivroRemoteTest extends TestCase {
    @Test
    public void testLookup() {
        Context ctx = new InitialContext();
        LivroBean bean =
            (LivroBean)ctx.lookup("java:global/biblioteca/LivroBean");
            ...
```



No-Interface View

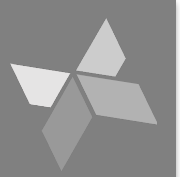
- Consiste no uso local de um bean sem declarar interfaces especiais
- A classe do bean pode ser referenciada via **injeção de dependências**

```
@EJB
BibliotecaBean bean; // classe do bean
```

- Também é possível referenciar o bean via JNDI

```
Context jndi = new InicialContext();
BibliotecaBean bean = (BibliotecaBean)
    jndi.lookup("java:app/BibliotecaBean");
```

- Os métodos podem ser chamados como se ele fosse uma referência Java comum (métodos herdados também fazem parte da interface)



Interface Local

- Uma interface **@Local** declara apenas **métodos acessíveis a clientes locais**:

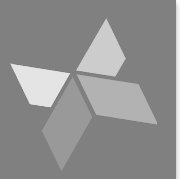
```
@Local
interface Biblioteca {
    void emprestar(Livro livro);
    void devolver(Livro livro);
}
```

- O bean só precisa **implementar** a interface:

```
@Stateless
class BibliotecaBean implements Biblioteca { ... }
```

- Se a interface **não puder ser anotada com @Local**, é possível usá-la se implementada no bean e passada como **parâmetro** de **@Local**:

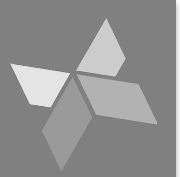
```
@Local(Biblioteca.class)
class BibliotecaBean implements Biblioteca {}
```



Usando uma interface @Local

- Os clientes de uma interface **@Local** devem pertencer à mesma aplicação e podem usar **injeção de dependências** com **@EJB** para referenciar a **interface** (não o bean):

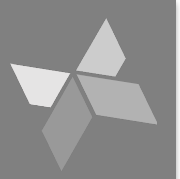
```
@SessionScoped
public class Administrador {
    @EJB Biblioteca biblioteca; // interface @Local

    public void recebidos(List<Livro>) {
        for(Livro m : livros) {
            biblioteca.devolver(m);
        }
    }
...
}
```



Usando uma interface @Local

- Pode-se também obter acesso ao bean através de lookup JNDI (também referenciando a interface):

```
@SessionScoped
public class Administrador {
    Biblioteca biblioteca;

    @PostConstruct
    public void init() {
        Context ctx = new InicialContext();
        biblioteca = (Biblioteca)
            ctx.lookup("java:app/Biblioteca");
    }
      ...
}
```



Acesso local a beans: resumo

- Beans com no-interface view: **@EJB** referencia **a classe do bean**

```
@EJB
ExampleBean exampleBean;
```

- Acesso via JNDI

```
ExampleBean exampleBean = (ExampleBean)
    initialContext.lookup("java:module/ExampleBean");
```

- Beans que implementam interface @Local: **@EJB** referencia **a interface**

```
@EJB
ExampleLocal example;
```

- Acesso via JNDI

```
ExampleLocal example = (ExampleLocal)
    initialContext.lookup("java:module/ExampleLocal");
```

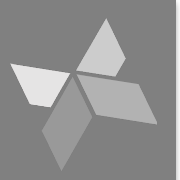
Interface Remota

- Interfaces **@Remote** têm métodos acessíveis por **clientes de outras aplicações ou JVMs**

```
@Remote
interface BibliotecaRemote {
    void consultar(Livro livro);
    void reservar(Livro livro);
}

@Stateless
class BibliotecaBean implements BibliotecaRemote { ... }
```

- Se a interface não puder ser anotada com **@Remote**, o bean ainda pode declará-la como parâmetro de **@Remote**:

```
@Remote(BibliotecaRemote.class)
class BibliotecaBean implements BibliotecaRemote, Intf2 {}
```



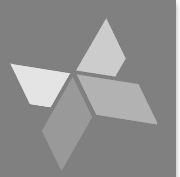
Usando uma interface remota

- Dentro do **mesmo servidor** pode-se obter a referência para um bean remoto via **injeção de dependências** (referenciando pela **interface**):

```
@EJB BibliotecaRemote biblioteca;
```

- Para acessar beans de containers localizados em **JVMs diferentes**, é necessário obter um **proxy** via **JNDI**:

```
Context ctx = new InicialContext();
BibliotecaRemote biblioteca = (BibliotecaRemote)
    ctx.lookup("java:global/libraryapp/Biblioteca");
```

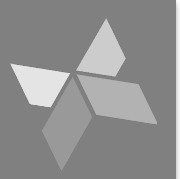


Clientes remotos RMI/IIOP

- **Clientes RMI/IIOP** que não estão em container Java EE podem obter um proxy para um EJB via JNDI global (proprietário) do servidor
- Em alguns servidores (ex: WebSphere 7) o proxy é um **objeto CORBA** que precisa ser **convertido** com **PortableRemoteObject.narrow()**:

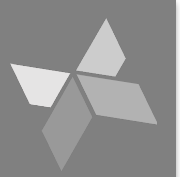
```
java.lang.Object stub = ctx.lookup("java:app/BibliotecaRemote ");
BibliotecaRemote biblioteca = (BibliotecaRemote)
    javax.rmi.PortableRemoteObject.narrow(stub,
                                          BibliotecaRemote.class);
```

- Chamadas remotas tem natureza diferente de chamadas locais
  - Parâmetros e tipos de retorno são passados **por valor** (são proxies de rede)
  - Não é possível fazer operações que alteram propriedades de um bean remoto por referência (é necessário alterar o bean localmente e depois **sincronizar**)



Web Services SOAP (JAX-WS)

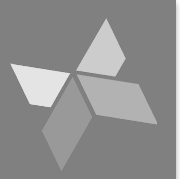
- Session Beans podem exportar interface como Web Service SOAP
  - Declara **@WebService(endpointInterface="nome da interface")** ou implementa a interface
- Com deploy, servidor gera classes, WSDL, e estabelece endpoint
  - **URI?wsdl** retorna WSDL que pode ser usado para gerar clientes
- **@WebMethod, @WebParam, @WebContext** e outras anotações configuram serviço e alteram defaults



Interface e implementação

```
@WebService
interface WSBeanInterface {
     int[] numerosDaSorte();
}
```

```
@Stateless @WebService
public class WSBean implements WSBeanInterface {
    @Override
    public int[] numerosDaSorte() {
        int[] numeros = new int[6];
        for(int i = 0; i < numeros.length; i++) {
            int numero = (int)Math.ceil(Math.random() * 60);
            numeros[i] = numero;
        }
        return numeros;
    }
}
```



Exemplo de cliente (GlassFish)

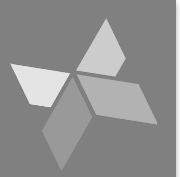
- Geração de stubs com JAX-WS

```
wscompile -s . http://servico/endpoint?wsdl
```

- Cliente

Classes geradas

```
public class App {
    public static void main( String[] args ) throws Exception {

        WSBeanService service = new WSBeanService();
        WSBeanInterface proxy = service.getWSBeanPort();
        java.util.List<Integer> numeros = proxy.numerosDaSorte();

        System.out.println("Numeros da sorte: ");
        for(int numero: numeros) {
            System.out.print(numero + " ");
        }
        System.out.println();
    }
}
```



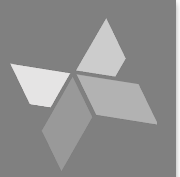
Embedable EJB

- Permite conectar sem depender de configurações proprietárias e obter um contexto JNDI global para localizar o bean

```
public void testEJB() throws NamingException {
    EJBContainer ejbC =
        EJBContainer.createEJBContainer(
           ResourceManager.getInitialEnvironment(new Properties()));
    Context ctx = ejbC.getContext();
    MyBean bean = (MyBean)
        ctx.lookup ("java:global/classes/org/sample/MyBean");
    // fazer alguma coisa com o bean
    ejbC.close();
}
```

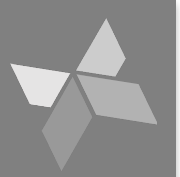
Configuração de Embeddable EJB

- É preciso ter no classpath os JARs requeridos pelo container (pode incluir JARs específicos do servidor usado)
- E necessário especificar as seguintes propriedades em um arquivo **jndi.properties** localizado no classpath:
  - **javax.ejb.embeddable.initial**
  - **javax.ejb.embeddable.appName**
- O valor das propriedades é dependente do fabricante e do ambiente do servidor. O fabricante também poderá requerer propriedades adicionais



Exercícios

- 5. Declare a seguinte interface **@Local**
  - **String dataHoje()** - retorne new Date()
  - **int[] numerosDaSorte(int quantidade)** - use exemplo fornecido
- 6. Crie um **Stateless bean** que implemente a interface do exercício 5 e um cliente Web que injete a interface e chame os métodos
- 7. Altere o bean do exercício 1 para que ele implemente uma interface **@Remote** e **@WebService** contendo seus dois métodos.
- 8. Escreva um cliente SOAP (use ferramentas WSDL do servidor) que rode em linha de comando e chame os métodos do bean através da interface desenvolvida no exercício anterior.



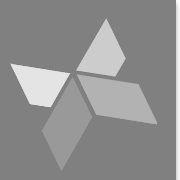
JAVAEE 7

EJB

enterprise javabeans

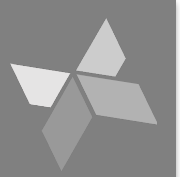
4

message-driven beans



Message-Driven Beans (MDB)

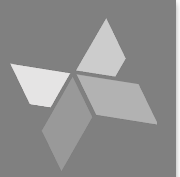
- **Message-driven beans** (MDB) são **processadores assíncronos**
- Clientes não acessam MDBs diretamente (MDBs não possuem interfaces) mas interagem através do envio de mensagens para um canal de mensageria (geralmente um destino JMS)
- MDBs atuam como **listeners de mensageria** registrados para consumir mensagens enviadas a um determinado canal
- Um MDB pode chamar outros beans, iniciar contextos transacionais, injetar e usar serviços disponíveis à aplicação



Message Driven Beans (MDB) com JMS

- Um MDB implementado com JMS é uma implementação de **MessageListener** anotada com **@MessageDriven**
- As mensagens que chegam no canal são recebidas por **onMessage()**
- **MessageDrivenContext** contém uma API com métodos para controle transacional e segurança

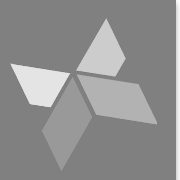
```
@MessageDriven( ... )
public class ProdutoMDB implements MessageListener {
    @Resource private MessageDrivenContext mdc;
    @EJB private ProdutoFacade fachada;

    public void onMessage(Message message) {
        // processa mensagem recebida
    }
}
```



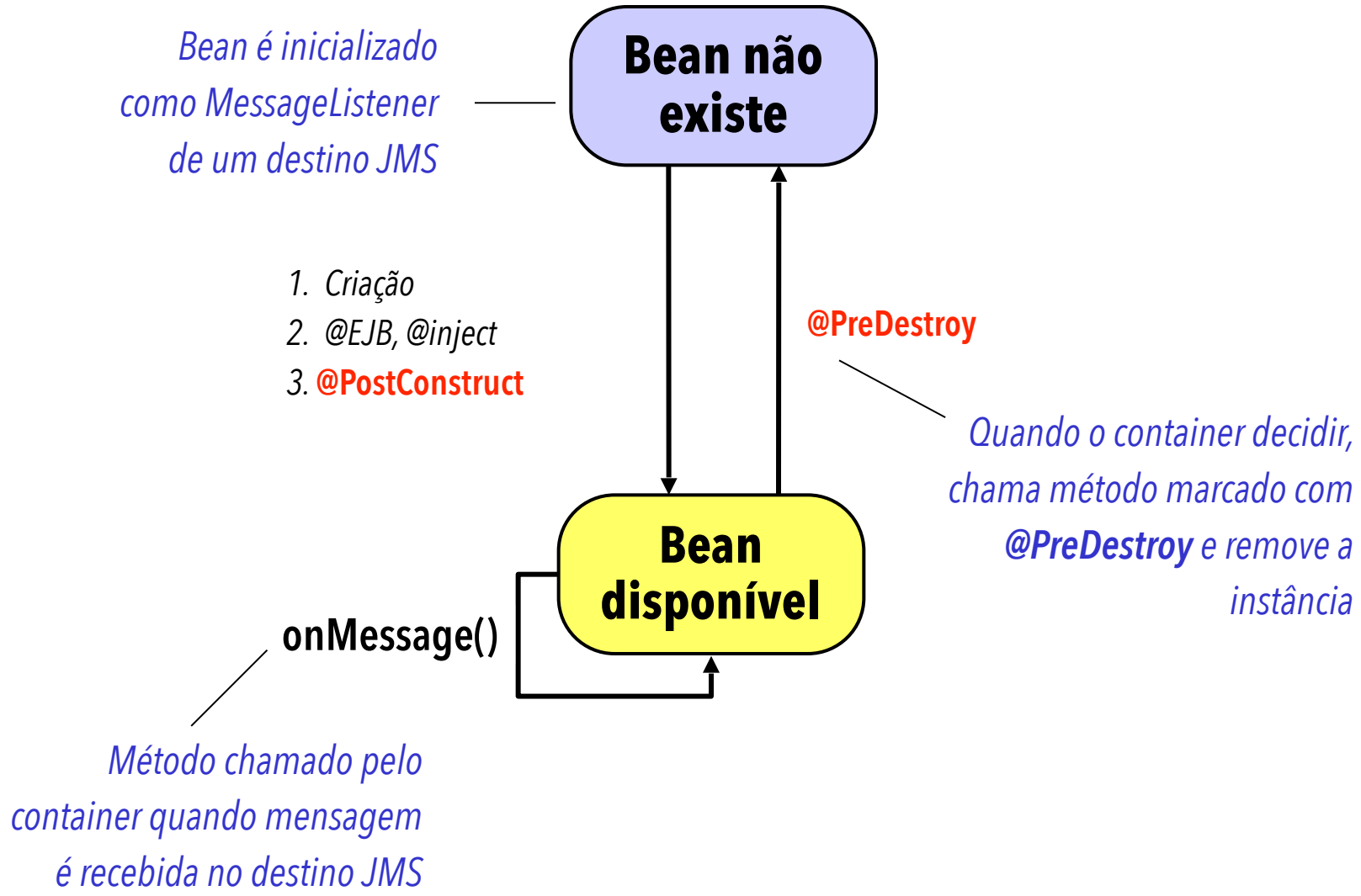
O método onMessage()

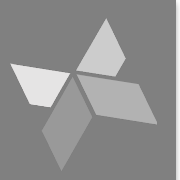
- Quando uma mensagem é recebida por um canal de mensageria, o container chama o método **onMessage()** dos MDBs registrados

```
public void onMessage(Message inMessage) {
    try {
        if (inMessage instanceof TextMessage) {
            String msg = (TextMessage) inMessage;
            fachada.processarPedido(msg);
            logger.info("Recebida: " + msg.getText());
        } else {...}
    } catch (JMSException e) {
        e.printStackTrace();
        mdc.setRollbackOnly();
    } catch (Throwable te) {...}
}
```



Ciclo de vida: Message-Driven Bean

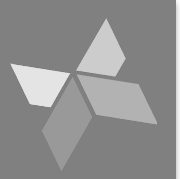




Mapeamento de um MDB a um canal

- O atributo **mappedName** é uma forma portável de informar o nome do canal, mas de acordo com a especificação é de implementação **opcional**.
  - Funciona no Glassfish mas em outros servidores onde pode ser necessário usar propriedades de ativação (sintaxe varia entre fabricantes).

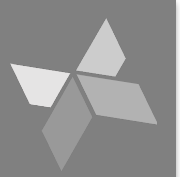
```
@MessageDriven(mappedName="pagamentos")
public class ProcessadorCartaoDeCredito implements MessageListener {
    @Override
    public void onMessage(Message message) {
        try {
            // processa a mensagem
        } catch (JMSException ex) { ... }
    }
}
```



Mapeamento de um MDB a um canal

- A propriedade **activationConfig** de **@MessageListener** recebe uma lista propriedades de configuração em anotações **@ActivationConfigProperty**. No WildFly/JBoss e WebSphere é obrigatório usar a propriedade **destination**, que informa a fila ao qual o bean deve ser registrado:

```
@MessageDriven(
  activationConfig={
    @ActivationConfigProperty(propertyName="destination",
                              propertyValue="pagamentos")}
)
public class ProcessadorCartaoDeCredito implements MessageListener {
    @Override
    public void onMessage(Message message) {
        try {
             // processa a mensagem
        } catch (JMSException ex) { ... }
    }
}
```



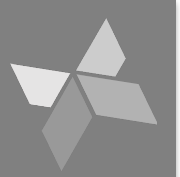
Filtro de mensagens

- A propriedade **messageSelector** filtra mensagens que serão recebidas
- O filtro atua sobre **cabeçalhos** e **propriedades** da mensagem, que incluem dados como data de validade, id e dados incluídos pelo remetente
- Um cliente JMS pode gravar uma **propriedade** em uma mensagem JMS com:

```
mensagem.setIntProperty("quantidade", 10);
```

- O MDB pode ser configurado para não receber mensagens que não tenha essa propriedade ou filtrar por valor:

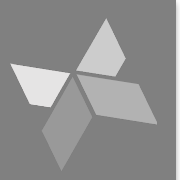
```
@MessageDriven(
  activationConfig={ ...,
    @ActivationConfigProperty(
        propertyName="messageSelector",
        propertyValue="quantidade is not null and quantidade >= 10")}
)
public class ProcessadorCartaoDeCredito implements MessageListener {…}
```



Produtor JMS

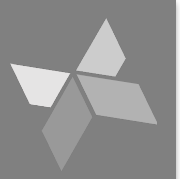
- Produtores são clientes que **enviam** mensagens a um destino
- Qualquer componente Java EE ou aplicação standalone que tenha acesso a conexões e canais usados pelo MDB
- Se estiver dentro do container, pode **injetar** a **conexão** ou o **contexto JMS 2.0 (via CDI)**, e o **destino (canal)** para onde enviará mensagens

```
@Resource(mappedName="ConnectionFactory")
private static ConnectionFactory connectionFactory;

@Inject @JMSConnectionFactory("jms/ConnectionFactory")
private static JMSContext context;

@Resource(mappedName="jms/Queue")
private static Destination queue;
```



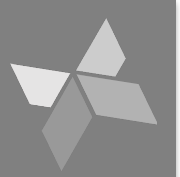
Um produtor JMS 2.0

```
@Stateless
public class ClienteEJB {
    @Resource(mappedName="topic/testTopic”)
    Topic topic;

    @Resource(mappedName="java:/JmsXA”)
    ConnectionFactory connectionFactory;

    public void metodo() throws Exception {
        JMSContext context =
            connectionFactory.createContext();
        context.createProducer().send(queue, "Texto");
    }
}
```



Um produtor JMS 2.0 (usando CDI)

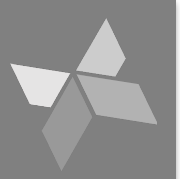
```
@Stateless
public class ClienteEJB {

    @Resource(mappedName="queue/testQueue”)
    Destination queue;

    @Inject
    @JMSConnectionFactory("jms/ConnectionFactory")
    JMSContext ctx;

    public void metodo() throws Exception {
        ctx.createProducer().send(queue, "Texto");
    }
}
```



Produtor JMS externo

- Clientes externos podem obter proxies através de registros JNDI globais (configuração e acesso dependentes de plataforma):

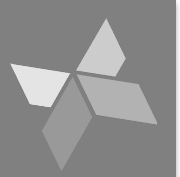
```
Context ctx = new InitialContext();

ConnectionFactory factory = (ConnectionFactory)
    ctx.lookup("global/ConnectionFactory");

Destination queue = (Destination)
    ctx.lookup("global/jms/queue/test");
```



Cliente JMS externo* ao container

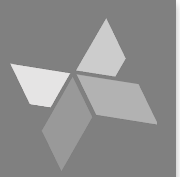
```
public class Client {
    public static void main(String[] args) throws Exception {
        Context jndi = new InitialContext();
        ConnectionFactory factory = (ConnectionFactory)
            jndi.lookup("ConnectionFactory");
        Destination topic = (Destination)
            jndi.lookup("topic/testTopic");

        Connection connection =
             factory.createConnection();
        JMSContext context =
             connectionFactory.createContext();
        context.createProducer().send(queue, "Mensagem");
    }
}
```

* Configuração e acesso ao JNDI depende da plataforma e pode ser mais complexo



Exercícios

- 9. Utilizando os exemplos como base, escreva um **produtor JMS** (pode ser um servlet) que envie mensagens para uma fila. As mensagens tem a seguinte estrutura:
  - "Codigo": X123 (StringProperty: gere uma sequência qualquer)
  - "Quantidade": 5 (IntProperty: varie entre 1 e 30)
  - Descrição do produto (Corpo da mensagem texto)
  - Use uma fila default. Ex: queue/A (WildFly)
- 10. Escreva um consumidor JMS **MessageDrivenBean** para ler e consumir as mensagens enviadas sincronamente, selecionando apenas as mensagens com menos de 10 produtos. Imprima os resultados no log ou na saída. Teste a aplicação rodando o cliente e observando a saída.



JAVAEE 7

EJB

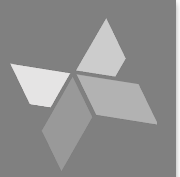
enterprise javabeans

5

concorrência

Conncorrência em EJB

- A especificação **desaconselha** em EJBs o uso explícito de threads, locks, sockets ou streams de I/O
  - Threads podem interferir no ciclo de vida dos componentes
  - Locks podem interferir no controle de transações e causar deadlocks
- Mas operações concorrentes, como notificações, agendamento e callbacks **podem** ser realizadas em EJB usando MDB, timers, métodos assíncronos e até mesmo executores que retornam objetos **Future**.
- A **exclusão mútua** de dados compartilhados é **default** em singletons, mas é possível configurar locks que permitem leituras simultâneas



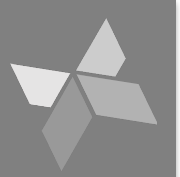
Concorrência controlada no container

- É opcional declarar explicitamente o controle de concorrência default:

  `@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)`

- O comportamento default garante que todo método de um Singleton é thread-safe, mas pode introduzir um gargalo desnecessário
  - Métodos que não mudam estado devem ser anotados **@Lock(LockType.READ)**, que permite acesso simultâneo
  - Métodos sem anotações **@Lock** ou anotados com **LockType.WRITE,** admitem apenas um cliente de cada vez, bloqueando os que esperam
  - A espera pode ser configurada com **@AccessTimeout** (na classe ou método). Acabando o tempo, o container lança **ConcurrentAccessTimeoutException**:

```
@Singleton
@AccessTimeout(value=60000)
public class SingletonBean {
    ...
}
```



Controle de concorrência

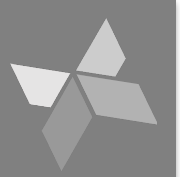
Todos os métodos são synchronized por default
(para mudar default é preciso usar @Lock)

```
@Singleton
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
public class ReadWriteSingleton {

    private String estado;

    @Lock(LockType.READ)
    public String getEstado() {
        return estado;
    }

    @Lock(LockType.WRITE)
    @AccessTimeout(value=300000)
    public void setEstado(String estado) {
        this.estado = estado;
    }
}
```

Múltiplos clientes podem acessar este método ao mesmo tempo

Apenas um cliente pode acessar este método ao mesmo tempo (este é o comportamento **default**)

O cliente espera no máximo 5 minutos (300 segundos) pelo acesso

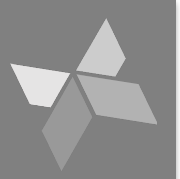


Controle de concorrência

```
@Singleton
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
public class ReadWriteSingleton {

    private String estado;

    @Lock(LockType.READ)
    public String getEstado() {
        return estado;
    }

    @Lock(LockType.WRITE)
    @AccessTimeout(value=300000)
    public void setEstado(String estado) {
        this.estado = estado;
    }
}
```

ConcurrencyManagementType .CONTAINER é default

@Lock(LockType.WRITE) é default



Concorrência controlada no bean

- É possível transferir a responsabilidade pelo controle da concorrência para o bean com a declaração:

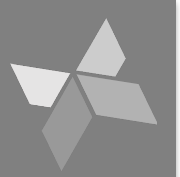
```
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
```

- Nesta configuração, os métodos do singleton não serão mais thread-safe e precisam usar Java para configurar restrições de acesso

- Pode-se usar mecanismos do Java para programação concorrente: blocos synchronized, variáveis volatile, sincronizadores e locks do pacote java.util.concurrent.

Chamadas assíncronas

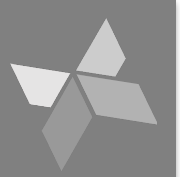
- Uma das maneiras de executar uma operação assíncrona usando session beans é implementá-lo como **cliente JMS** (padrão Service Activator):
  - O método cria uma mensagem, empacota os dados necessários para executar a operação e envia a mensagem para uma fila JMS mapeada a um MDB.
  - Recebida a mensagem, o MDB extrai os parâmetros e chama a operação.
- Outra forma (mais simples) de realizar operações assíncronas (desde Java EE 6) é usar métodos anotados com **@Asynchronous**
  - Método devolve um objeto **Future<V>** que permite a criação de **callbacks**
  - Suportado em qualquer session bean



Métodos assíncronos

- Retornam **Future<V>** (implementado por **javax.ejb.AsyncResult**) ou **void**
  - Um **Future** guarda o resultado que o cliente pode obter usando **get()**, quando estiver disponível (quando **isDone()** retornar true)
- **@Asynchronous** também pode ser declarado na classe

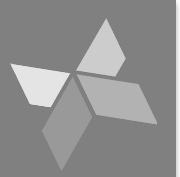
```
@Stateless
public class Tradutor {
    @Asynchronous
    public Future<String> traduzir(String texto) {
         StringBuilder textoTraduzido;
        // ... realiza a tradução (demorada)
        return new AsyncResult<String>(textoTraduzido.toString());
    }
}
```



Cliente de método assíncrono

```
@SessionScoped @Named
public class ManagedBean {
    @EJB
    private Tradutor tradutor;

    private Future<String> future;

    public void solicitarTraducao(String texto) {
       future =  tradutor.traduzir(texto); // assíncrono
    }
    public String receberTraducao() {
       while (!future.isDone()){
           Thread.sleep(1000);
           // fazer outras coisas
       }
       return (String)future.get();
    }
}
```

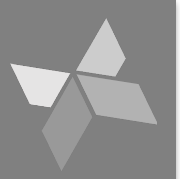
- O exemplo abaixo ilustra um **cliente** (CDI managed bean) que usa um método assíncrono



Timers

- Serviços de agendamento configuráveis para session beans
  - Apenas **Stateless** e **Singleton**
- Permitem que métodos sejam chamados **periodicamente**, ou uma única vez em **data e hora marcada**, ou depois de um **intervalo** especificado
- Configurados anotando métodos com **@Schedule** (ou **@Schedules** para múltiplos timers), que recebe **parâmetros** com regras de agendamento
  - hour, minute, second, dayOfWeek, dayOfMonth, timezone, year, etc.

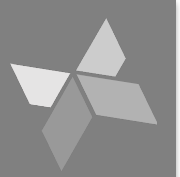
```
@Stateless
public class MyTimer {
   @Schedule(hour="*", minute="*", second="*/10")
   public void printTime() { ... }
...
}
```



Timers

- Exemplos de expressões:
  - **`hour="7,19,23", dayOfWeek="1-5"`**
    Executa 7, 19 e 23 h, segunda a sexta
  - **`minute="30", hour="5"`**
    Executa uma vez às 5 horas e 30 minutos
  - **`hour="*", minute="*", second="*/10"`**
    Toda hora e minuto a cada 10 s

- Também podem ser criados programaticamente usando métodos de **TimerService** em beans que implementam a interface **TimedObject** e seu método **ejbTimeout(Timer)**
  - Neste caso o timer é configurado na inicialização do bean



Timers

```
@Singleton @Startup
public class MyTimer implements TimedObject {

    @Resource TimerService timerService;

    @PostConstruct
    public void initTimer() {
        if (timerService.getTimers() != null) {
            for (Timer timer : timerService.getTimers()) {
                timer.cancel();
            }
        }
        timerService.createCalendarTimer(
          new ScheduleExpression().hour("*").minute("*").second("*/10"),
          new TimerConfig("myTimer", true)
        );
    }

    @Override
    public void ejbTimeout(Timer timer) {
        //código a executar quando o timer disparar
    }
}
```

Timers

- Além de **createCalendarTimer()**, que faz agendamento periódico, TimerService também permite criar timers que disparam uma única vez com **createSingleActionTimer()**

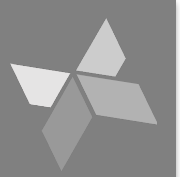
- Se o bean não implementa a interface **TimedObject**, ele pode definir um timeout usando um método anotado com **@Timeout**

  - O método precisa retornar void, e não precisa receber parâmetros (se receber deve ser um objeto Timer)

```
@Timeout
public void timeout(Timer timer) {
    //codigo a executar quando  timer disparar
}
```

Utilitários de concorrência

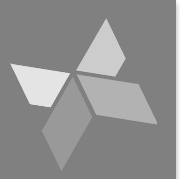
- Executores que permitem manipular threads dentro de aplicações Java EE, disponíveis no pacote **javax.enterprise.concurrent**
- Similares a classes de **java.util.concurrent** (do qual derivam) mas são gerenciadas pelo container e adequadas ao uso em EJBs e servlets
- As mais importantes são:
  - **ManagedExecutorService**
  - **ManagedScheduledExecutorService**
  - **ManagedThreadFactory**
  - **ContextFactory**



ManagedExecutorService

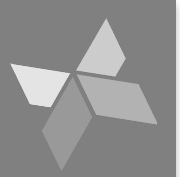
- Usando **@Resource** para injetar e usar um **ManagedExecutorService**

```
@Stateless
public class ConcurrentBean {
    @Resource
    ManagedExecutorService service;

    @Asynchronous
    public Future<String> tarefaDemorada() {
        return service.submit(new Callable<String>() {
            @Override
            public String call() throws Exception {
                StringBuffer buf = new StringBuffer();
                //... tarefa demorada
                return buf.toString();
            }
        });
    }
}
```



Exercícios

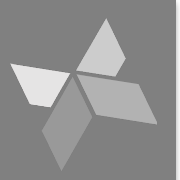
- 11. Escreva um **Singleton Timer** que execute uma operação que grave um String numa variável compartilhada depois de **1 minuto**
- 12. Escreva um bean que utilize o serviço do exercício anterior, lendo o String e retornando o resultado como um **Future**
- 13. Escreva um managed bean que acesse como cliente, o bean do exercício anterior, e uma página Web JSF que use Ajax para atualizar o valor na tela quando o string estiver pronto.



JAVAEE 7

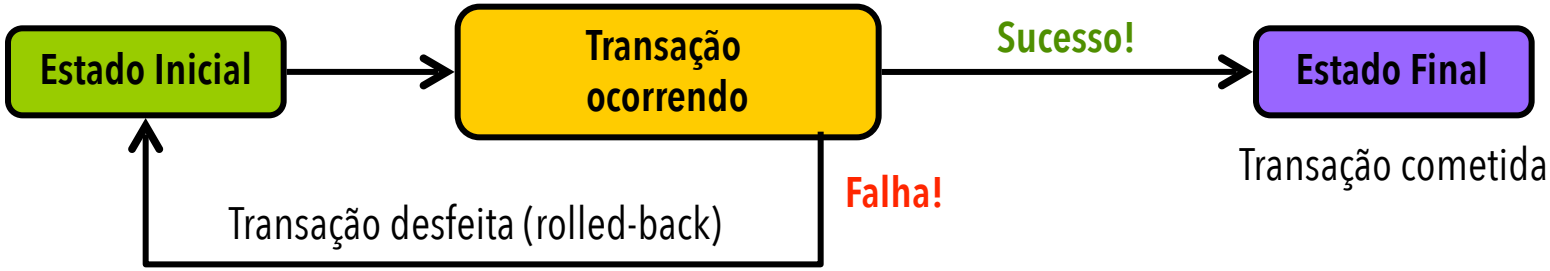
EJB
enterprise javabeans

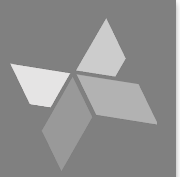
6
transações



Transações

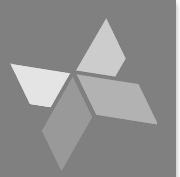
- Unidade **atômica** de trabalho
  - Ou todas as etapas terminam com sucesso ou processo é desfeito
- Commit e rollback
  - Sub-operações são realizadas e resultados temporários são obtidos
  - No final, a transação é cometida (**commit**) e o resultado final torna-se **durável** se cada sub-operação funcionar com sucesso
  - Se uma sub-operação falhar, as alterações realizadas são revertidas (**rollback**) e os dados voltam aos estados anteriores





Transações em EJB

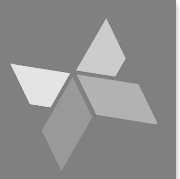
- A especificação EJB não suporta transações aninhadas. Se uma transação começa quando já existe um contexto transacional, ela pode:
  - **Continuar** o contexto transacional existente
  - **Interromper** o contexto transacional existente
  - **Iniciar** um novo contexto transacional
- O controle transacional em EJB resume-se a **demarcação**: determinar o início e o fim. Há duas formas de demarcar transações:
  - **Explícita**, ou programática (Bean-Managed Transactions – **BMT**)
  - **Implícita**, ou declarativa (Container-Managed Transactions- **CMT**) - **default**



Bean-Managed Transactions (BMT)

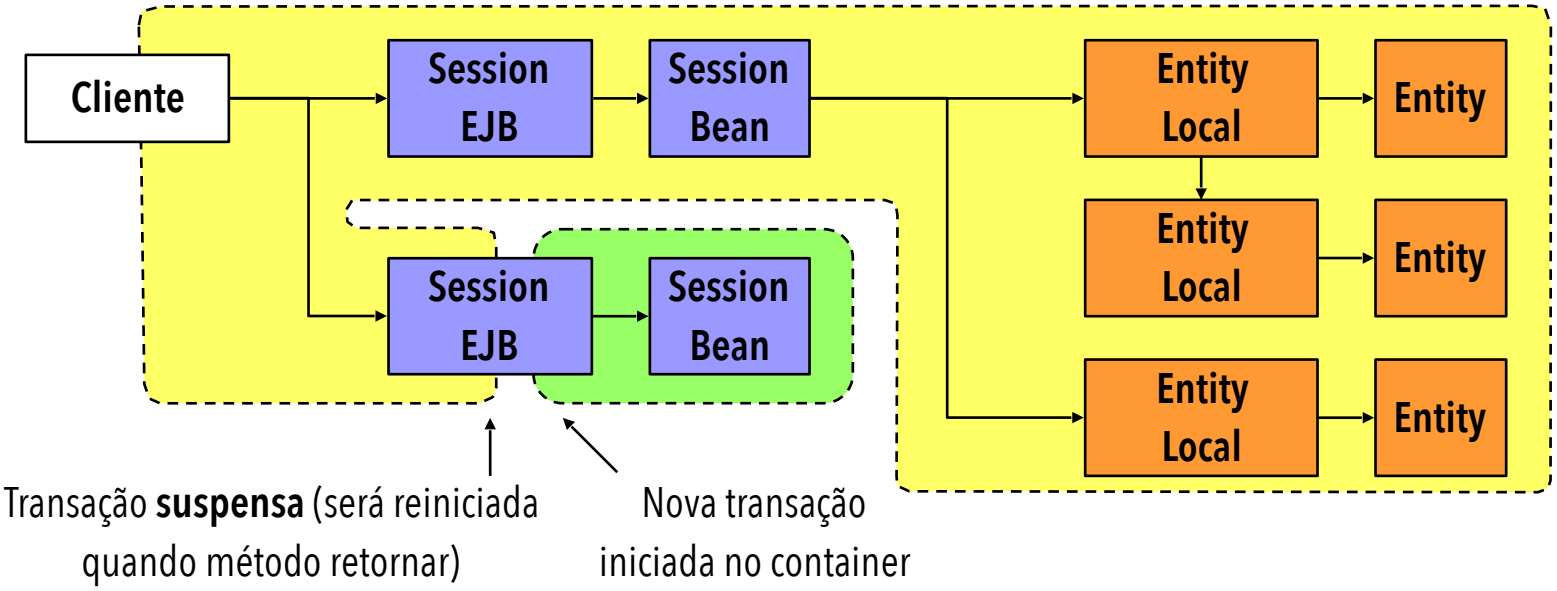
- Demarcação de transações através do uso explícito de APIs Java de controle transacional como
  - java.sql.Connection (JDBC)
  - javax.jms.Session (JMS)
  - EntityTransaction (JPA)
  - UserTransaction (JTA)
- O uso dessas APIs é proibido em CMT (Container-Managed Transactions)
- É preciso configurar BMT via XML ou @TransactionManagement:

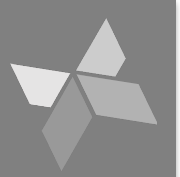
```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class Fachada { ... }
```



Propagação de transações

- Transações terminam no mesmo lugar onde começaram
- O contexto da transação será **propagado** para todos os métodos chamados (se eles não iniciarem nova transação)
- Se métodos chamados iniciarem nova transação, a transação principal será **suspensa** até que o método termine.





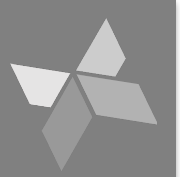
Bean-Managed Transactions (BMT)

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)

public class Bean {

  @Resource DataSource ds;

  public void processarPedido (String id, int quantidade) {
     Connection con = ds.getConnection();
     try {
        con.setAutoCommit(false);
        atualizarEstoque(id, quantidade);
        con.commit();
     } catch (Exception e) {
       try {
           con.rollback();
       } catch (SQLException sqx) {}
     }
  }
}
```

Bean usa transações do JDBC e precisa ser declarado como BMT

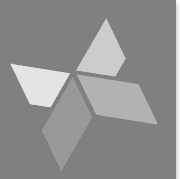
UserTransaction

- **javax.transaction.UserTransaction**, pode ser usada em qualquer componente EJB e clientes para demarcar transações distribuídas (two-phase commit)
- UserTransaction é usada internamente em transações declarativas (CMT) e sua API pode ser usada explicitamente em BMT
- Principais métodos de UserTransaction usados em BMT:
  - **begin()**: marca o início
  - **commit()**: marca o término
  - **rollback()**: condena a transação



UserTransaction em componente EJB

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class Bean {
    @Resource EJBContext ctx;
    private double saldo;

    public void sacar(double quantia) {
        UserTransaction ut = ctx.getUserTransaction();
        try {
            double temp = saldo;
            ut.begin();
            saldo -= quantia;
            atualizar(saldo); // afetado pela transaçao
            ut.commit();
        } catch (Exception ex) {
            try {
                ut.rollback();
                saldo = temp;
            } catch (SystemException e2) {}
        }
    }
}
```



Transações iniciadas no cliente

- Qualquer cliente pode iniciar transações JDBC e JMS
  - Cliente pode ser standalone, servlet, bean, etc.
- Clientes remotos precisam obter uma instância da classe **UserTransaction** através de JNDI
  - Pode necessitar de configuração proprietária
  - Geralmente disponibilizado em **java:comp/UserTransaction**

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

Container-Managed Transactions (CMT)

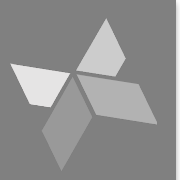
- É **default** (mas pode ser declarado em classes e métodos):

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class Fachada { … }
```

- Não demarca blocos de código. A granularidade mínima é o **método**.
- As transações são gerenciadas por default de acordo com a politica **REQUIRED**, que garante um contexto transacional (continua existente, ou cria novo).
- CMT não permite APIs transacionais (isto inclui métodos da API de transações de java.sql, javax.jms, EntityTransaction e UserTransaction).

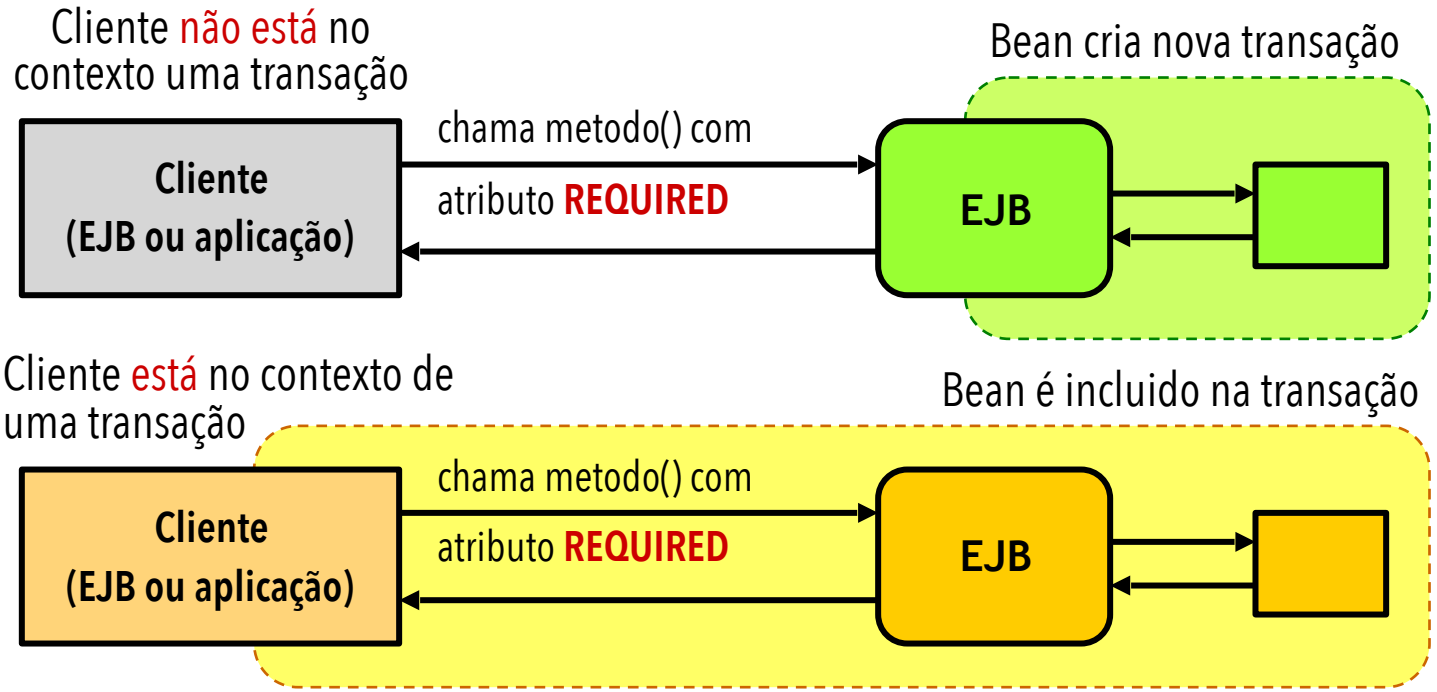
Atributos (políticas de propagação)

- O elemento **@TransactionAttribute** define a política de propagação das transações do componente ou método (seu comportamento ao ser chamado dentro ou fora de contexto transacional)
- Valores suportados (depende do tipo de bean)
  - `TransactionAttributeType.NOT_SUPPORTED`
  - `TransactionAttributeType.SUPPORTS`
  - `TransactionAttributeType.REQUIRED`
  - `TransactionAttributeType.REQUIRES_NEW`
  - `TransactionAttributeType.MANDATORY`
  - `TransactionAttributeType.NEVER`
- Declaração no nível da classe **não afeta** métodos **@PostConstruct**, etc. (callbacks) que não têm contexto transacional especificado



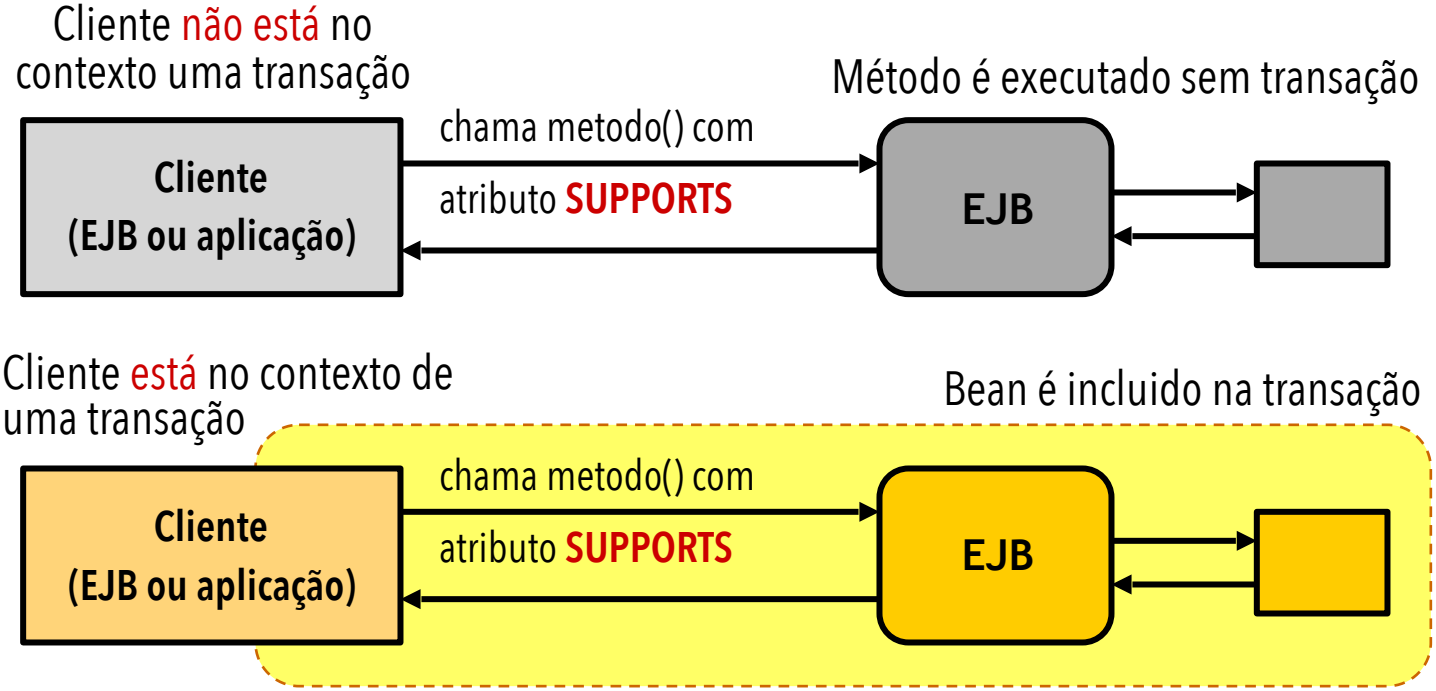
REQUIRED

- 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



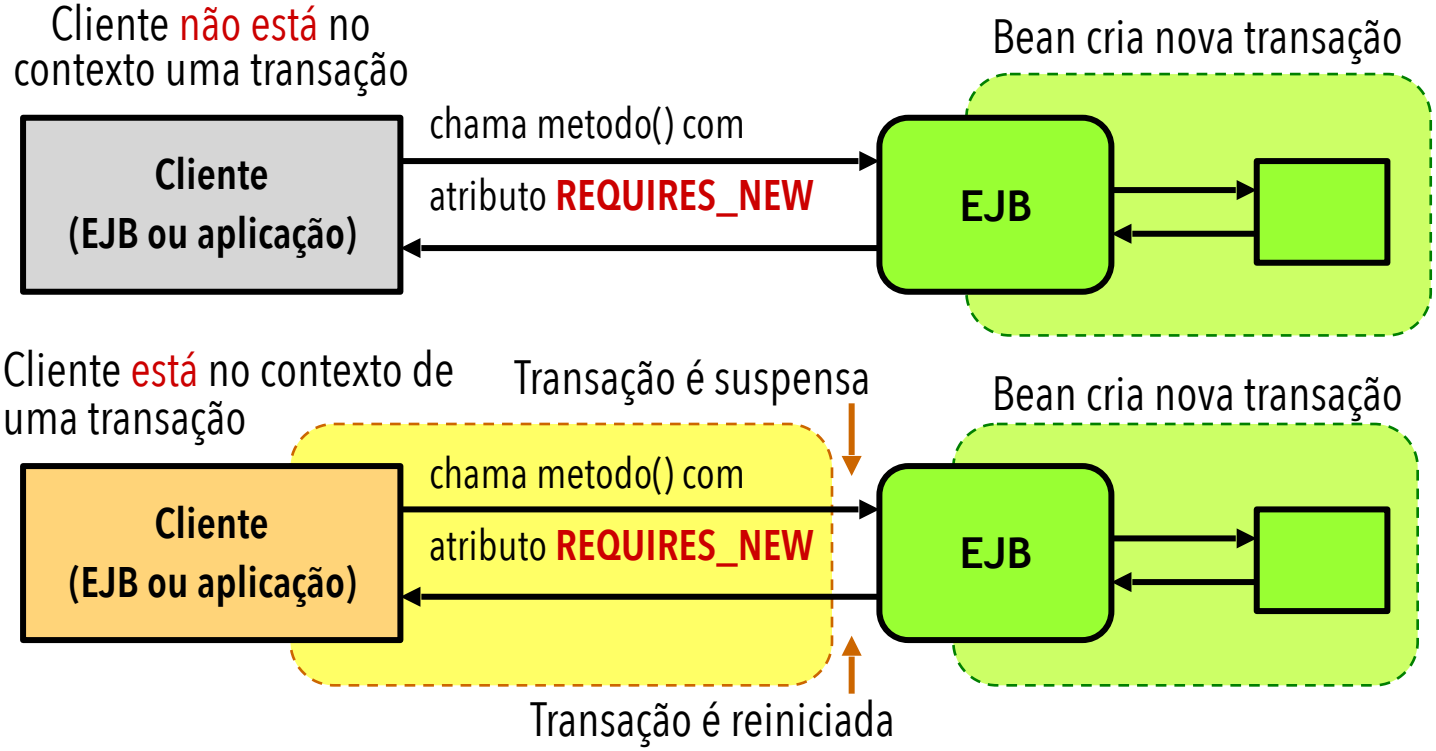
SUPPORTS

- 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



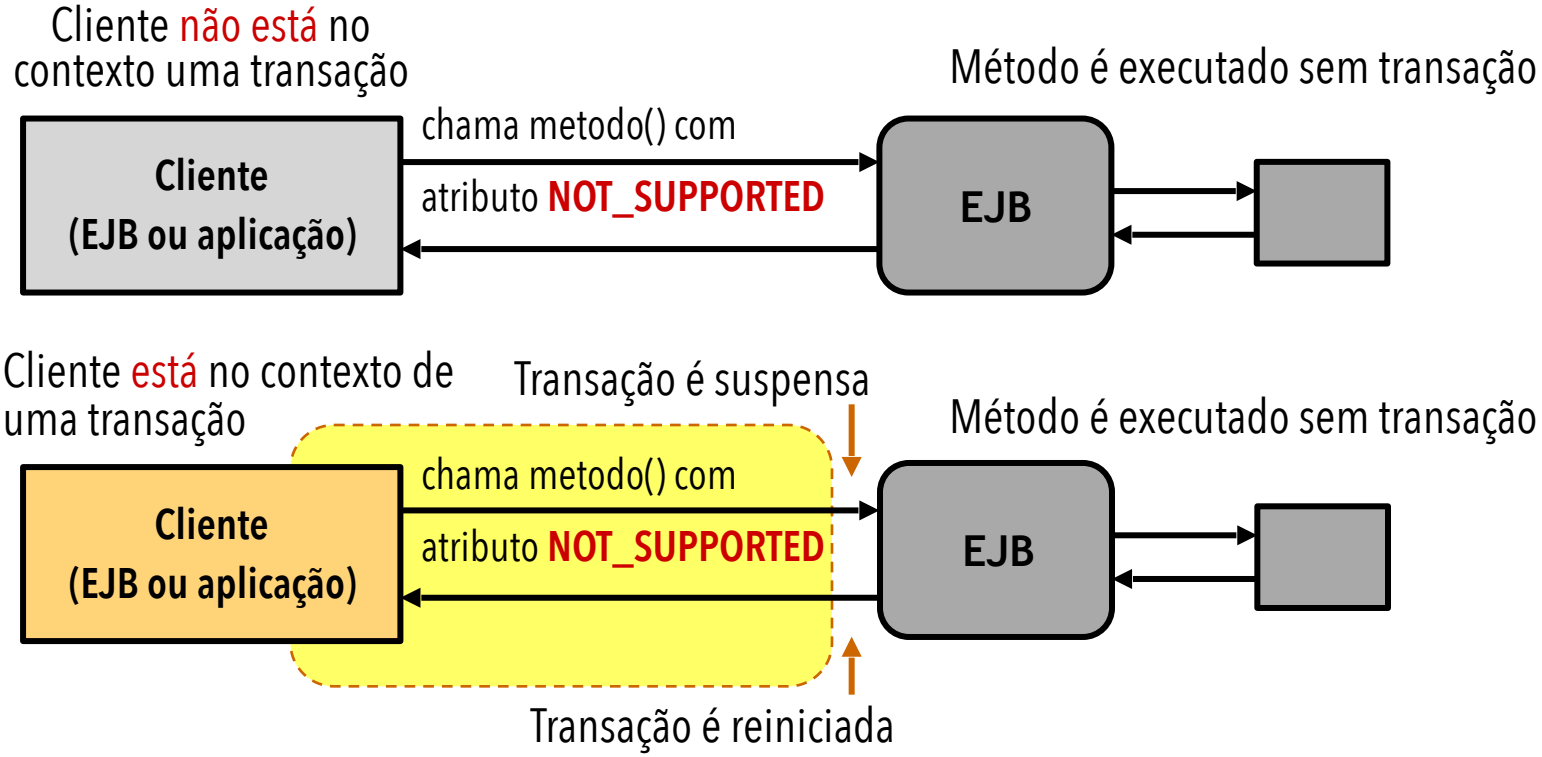
REQUIRES_NEW

- 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.



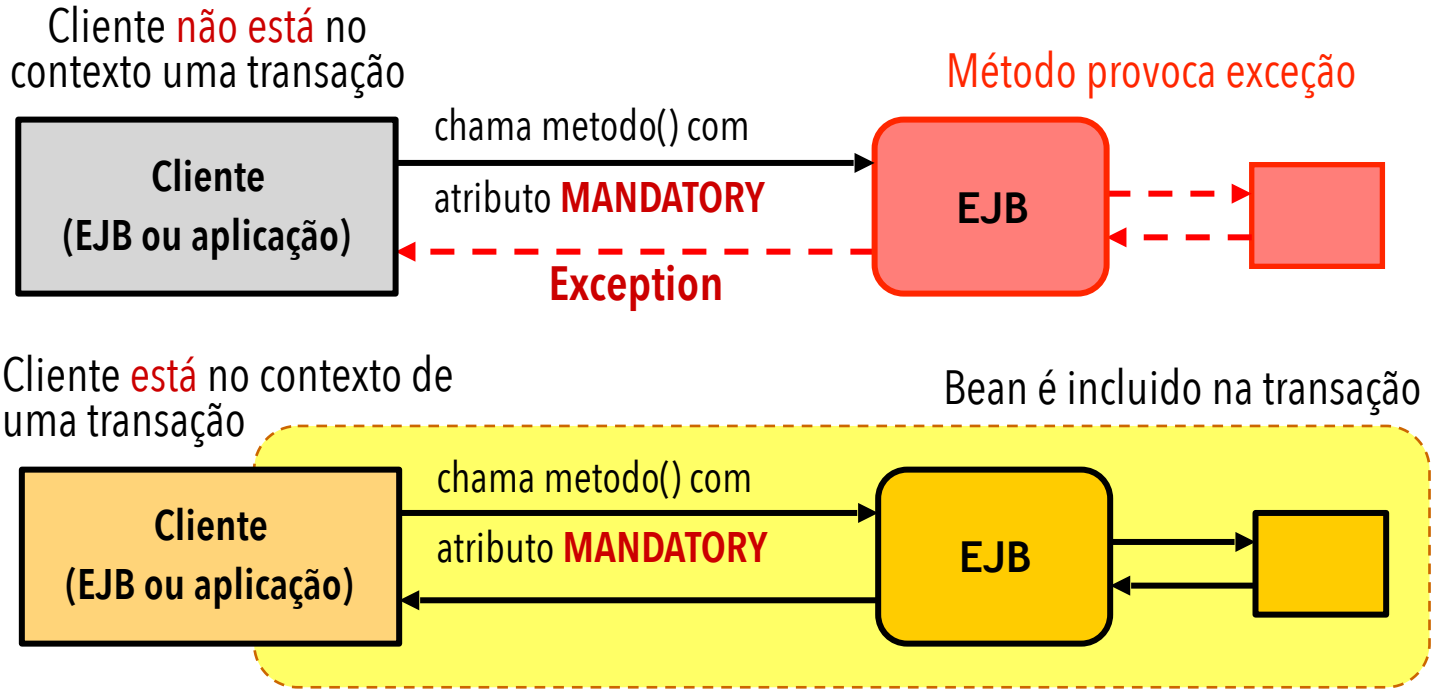
NOT_SUPPORTED

- 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)



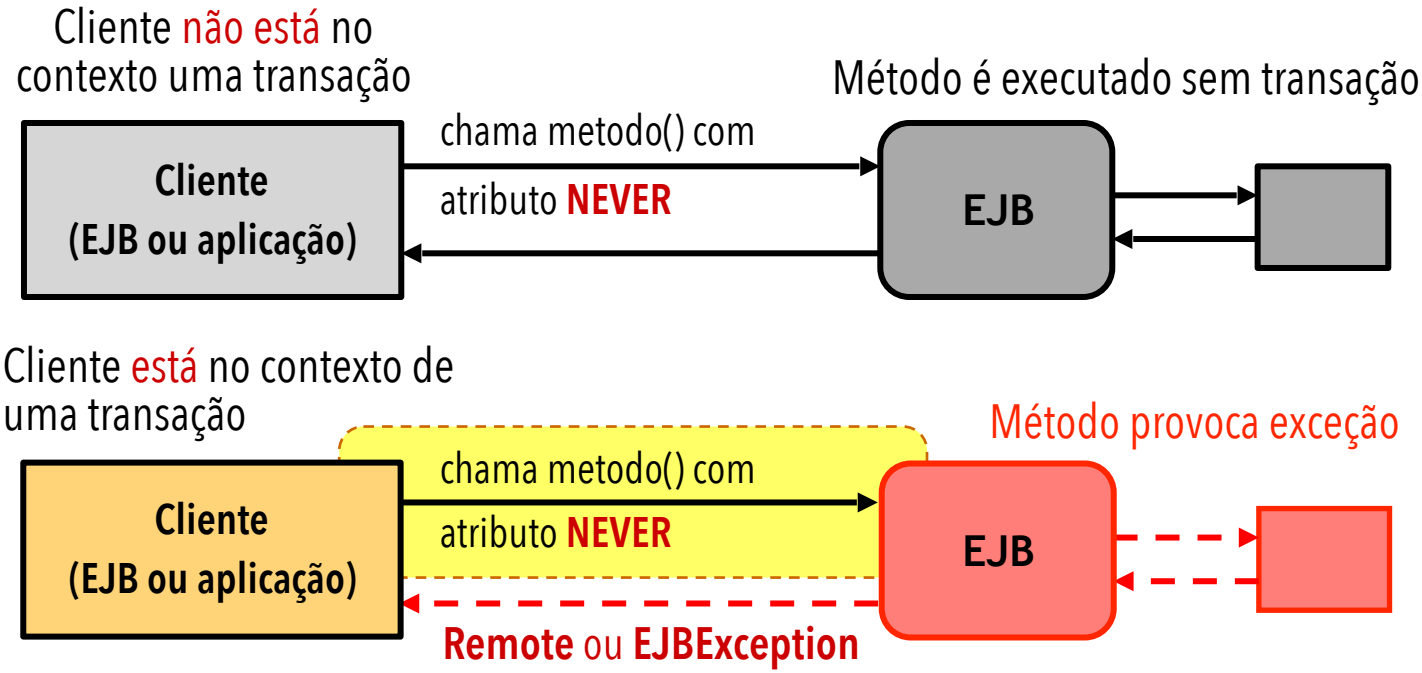
MANDATORY

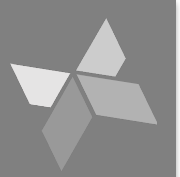
- Método só pode ser chamado no escopo de uma transação do cliente
  - Se o método for chamado fora de uma transação, ele causará javax.transaction.TransactionRequiredException (ou javax.ejb.TransactionRequiredLocalException)



NEVER

- 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)





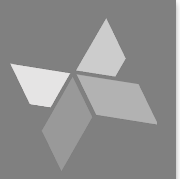
Destino de uma transação CMT

- Apenas exceções do sistema (Runtime, Remote, EJBException) provocam rollback automático da transação
- Chamar **rollback()** explicitamente não é permitido em CMT. Para que uma exceção dispare automaticamente um rollback há duas alternativas:
  - Anotar a exceção com **@ApplicationException**
  - **Condenar** a transação CMT usando **setRollbackOnly()** de EJBContext. O status está disponível via **getRollbackOnly()**:

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

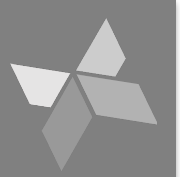
Transações em Message-driven beans

- Como um MDB não é chamado diretamente, não propaga contexto transacional, mas podem iniciar novas transações
  - A transação deve iniciar e terminar dentro do método **onMessage()**
- É mais simples usar 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
- Em CMT suportam apenas as políticas **NOT_SUPPORTED** e **REQUIRED**



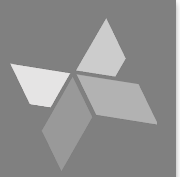
Transações em Stateful Session Beans

- O estado de Stateful Session Beans é mantido em **variáveis de instância** que não são recuperadas 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), pode-se guardar o estado antes do begin e recuperar após o rollback
  - Se o container iniciar a transação (CMT) é preciso implementar callbacks (**SessionSynchronization**)



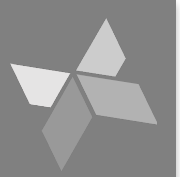
Interface SessionSynchronization

- Pode ser implementada por Stateful Session Beans para capturar eventos lançados nos pontos de demarcação
- void **afterBegin()**:
  - Chamado logo após o início da transação.
  - Guarde 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, state é true; caso contrário, é false deve-se restaurar o estado do bean aos valores guardados em afterBegin()



Exercícios

- 14. Os métodos do bean LoteriaBean falham aleatoriamente (a chance é de 25%). SorteioBean precisa ler o último valor do banco e substituir pelo novo valor apenas se o novo valor for maior, e permitir apenas 5 chamadas pelo mesmo cliente. Configure o mínimo de transações em SorteioBean para que a aplicação funcione corretamente (use SessionSynchronization se necessário).

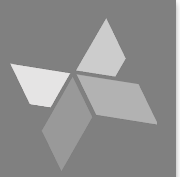


JAVAEE 7

EJB enterprise javabeans

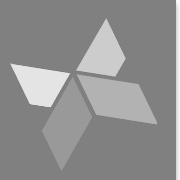
7

segurança



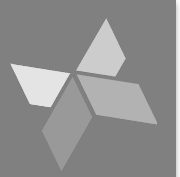
Segurança em EJB

- A segurança em EJB consiste do **controle de acesso** (**autorização**) a métodos públicos e trechos de código
- A configuração da autorização pode ser **declarativa** e/ou **programática**
- A configuração **declarativa** pode ser feita através de **anotações** antes de cada método ou via **web.xml**
- A configuração **programática** utiliza métodos de **EJBContext** e permite controlar acesso de trechos de métodos



Autorização em EJB: anotações

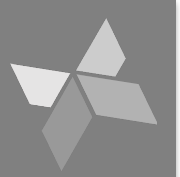
```
@DeclareRoles({"amigo", "administrador"})
@Stateless public class ImagemBean {
      @RolesAllowed("administrador")
      public void removerUsuario(Usuario u) {  ... }

      @RolesAllowed("amigo")
      public List<Usuario> listaDeUsuarios() { ... }

      @PermitAll
      public String getInfo() { ... }

      @RolesAllowed("**")
      public String getFreeStuff() { ... }

      @DenyAll
      public String loggingData() { ... }
}
```



Autorização em ejb-jar.xml

- **Raramente usado** por desenvolvedores (já que toda a segurança pode ser configurada por anotações)

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee" ... >
        <enterprise-beans>
                <session>
                        <ejb-name>testeBean</ejb-name>
                        <ejb-class>testebean.TesteBean</ejb-class>
                        <session-type>Stateless</session-type>
                        <transaction-type>Container</transaction-type>
                </session>
        </enterprise-beans>
        <assembly-descriptor>
                <security-role>
                        <role-name>administrador</role-name>
                </security-role>
                <method-permission>
                        <role-name>administrador</role-name>
                        <method>
                                <ejb-name>testeBean</ejb-name>
                                <method-name>*</method-name>
                        </method>
                </method-permission>          ...
        </assembly-descriptor>
</ejb-jar>
```



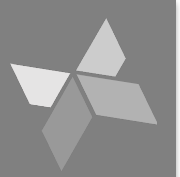
Propagação de identidade

- **@RunAs("role")**
  - Declarado antes da declaração de classe permite que o componente execute usando um role (permissão) específico
  - Permite que componente obtenha uma permissão mesmo que ela não tenha sido propagada pelo container cliente

```
@MessageDriven
@RunAs("administrador")
public class TestEJB {  ... }
```

O container **confia** na identidade propagada (não autentica de novo)

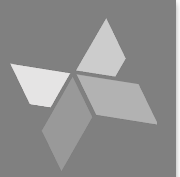
```
<message-driven>
      <ejb-name>TestEJB</ejb-name>
      <ejb-class>br.com.ejb.examples.TestEJB</ejb-class>
     <security-identity>
            <run-as>
                  <role-name>administrador</role-name>
            </run-as>
       </security-identity>
</message-driven>
```



Autorização com EJBContext

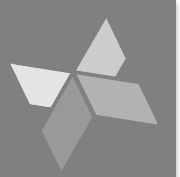
- Métodos de **javax.ejb.EJBContext**
  - Herdados por SessionContext e MessageDrivenContext
  - Permite controle de acesso em trechos de métodos
- Mesmos métodos, mas com **nomes diferentes**
  - Caller no contexto EJB é o User no contexto Web

```
String loggedUser = ctx.getCallerPrincipal().getName();

if( !ctx.isCallerInRole("administrador")) {
        if (!loggedUser.equals("cerebro")) {
                throw new EJBAccessException("...");
        }
}
```



Exercícios

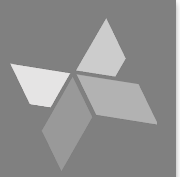
- 15. Configure o acesso aos métodos de LojaBean para que
  - a) O método **getProdutos()** possa ser acessado por qualquer usuário
  - b) O método **getProduto()** e **getCarrinho()** possam ser acessados por usuários que tenham o role "**cliente**"
  - c) O método **setProduto()** e **deleteProduto()** possam ser acessados por usuários que tenham o role "**admin**"
  - Use a aplicação Web fornecida que fornece os usuários, senhas, roles e mecanismo de autenticação (é possível que você tenha que configurar usuarios, senhas e roles no seu servidor)



Exercícios de revisão

- Faça os exercícios 5, 7, 10 e 14* da aplicação Biblioteca
  - 5) Aplicação Web com EJB e CDI (use aplicação parcialmente concluida em /projeto-parcial)
  - 7) Singleton Session Beans (exercício parcialmente resolvido)
  - 10) Stateful Session Beans (empréstimo de livros)
  - 14) Message-Driven Beans (processamento CSV assíncrono)

*Exercicios: http://www.argonavis.com.br/download/exercicios_javaee.html



Referências

- EJB 3.2 Expert Group. *JSR 345: Enterprise JavaBeans, Version 3.2*. Oracle. 2013. http://download.oracle.com/otndocs/jcp/ejb-3_2-fr-spec/index.html
- Eric Jendrock et al. *The Java EE 7 Tutorial*. Oracle. 2014. https://docs.oracle.com/javaee/7/JEETT.pdf
- Linda deMichiel and Bill Shannon. *JSR 342. The Java Enterprise Edition Specification. Version 7*. Oracle, 2013. http://download.oracle.com/otn-pub/jcp/java_ee-7-fr-spec/JavaEE_Platform_Spec.pdf
- Arun Gupta. *Java EE 7 Essentials*. O'Reilly and Associates. 2014.