



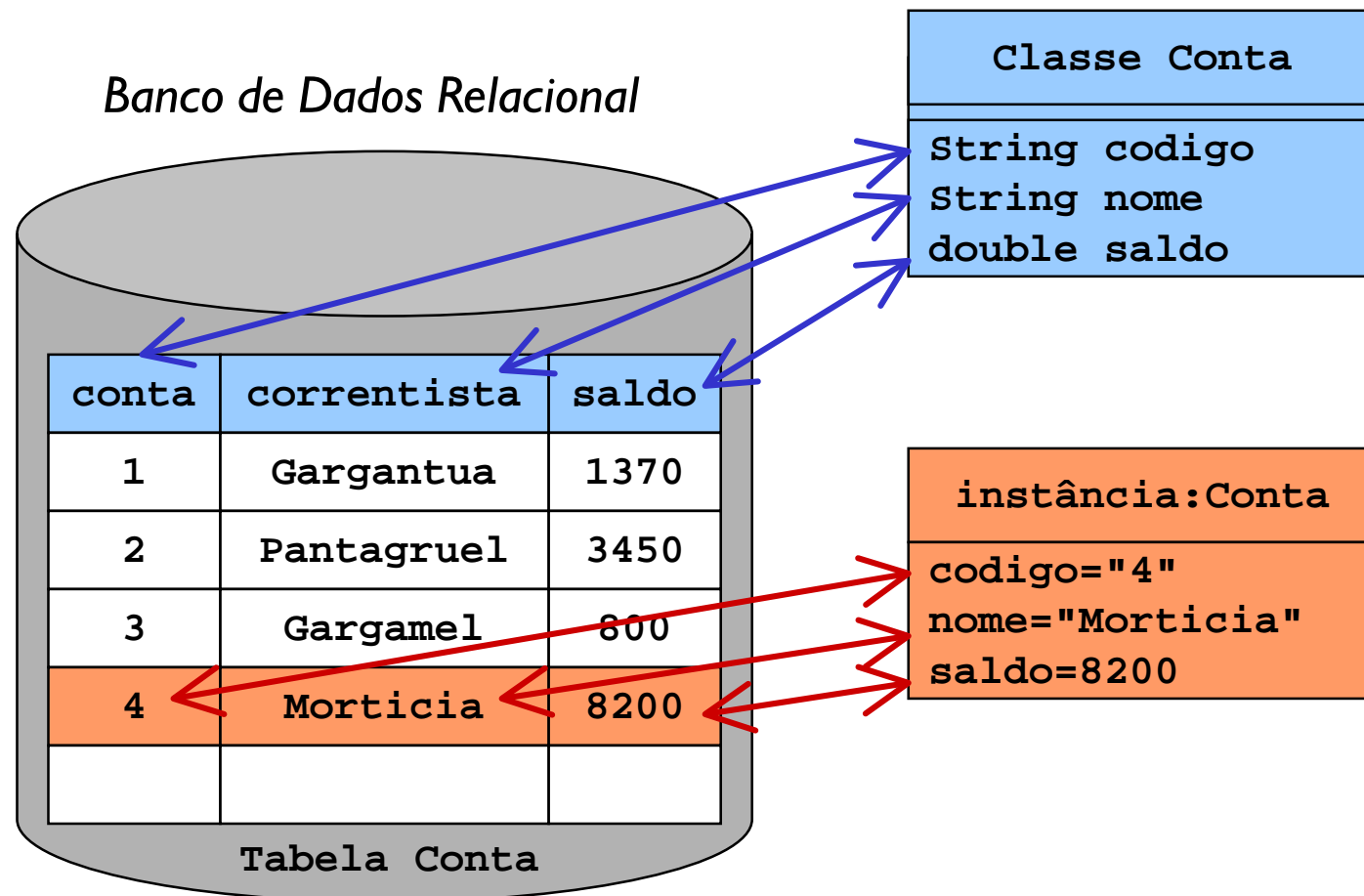
Entity Beans

Helder da Rocha
www.argonavis.com.br

- *Uma maneira de armazenar o estado de objetos Java é através da **serialização***
 - *Vantagem: facilidade de leitura e gravação: objeto pode ser enviado pela rede, armazenado em um banco ou árvore JNDI*
 - *Desvantagem: inviabilidade de realização de pesquisa sobre campos dos objetos: qualquer pesquisa necessitaria a desserialização de cada objeto para obter os campos pesquisados*
 - *Conclusão: para persistência de grande quantidade de dados que precisam de expor seus dados como atributos de pesquisa, serialização não é viável (solução seria serialização em XML!)*
- *Soluções alternativas*
 - *Bancos de dados relacionais: oferecem recursos poderosos e eficientes de busca, mas, não são orientados a objeto (SQLJ, JDBC)*
 - *Bancos de dados orientados a objeto*
 - *Mapeamento objeto-relacional (O/R Mapping)*

Object-Relational Mapping

- *Em vez de serializar os objetos, decompomos cada um em suas partes e guardamos suas partes no banco*



O que é um Entity Bean?

- *Session beans modelam a lógica relacionada a tarefas realizadas pela aplicação*
- *Entity beans modelam entidades*
- *São componentes que representam dados*
 - *Session beans podem guardar informações em bancos de dados mas não "representam" dados.*
 - *Entity beans são os dados*
- *Permitem que se manipule com objetos ignorando a forma de armazenamento*
 - *A especificação não define nenhum tipo obrigatório*
- *Duas partes*
 - *Entity bean instance: os dados na memória (representa uma instância da classe do Entity Bean)*
 - *Entity bean data: os dados fisicamente armazenados no banco*

Partes de um entity bean

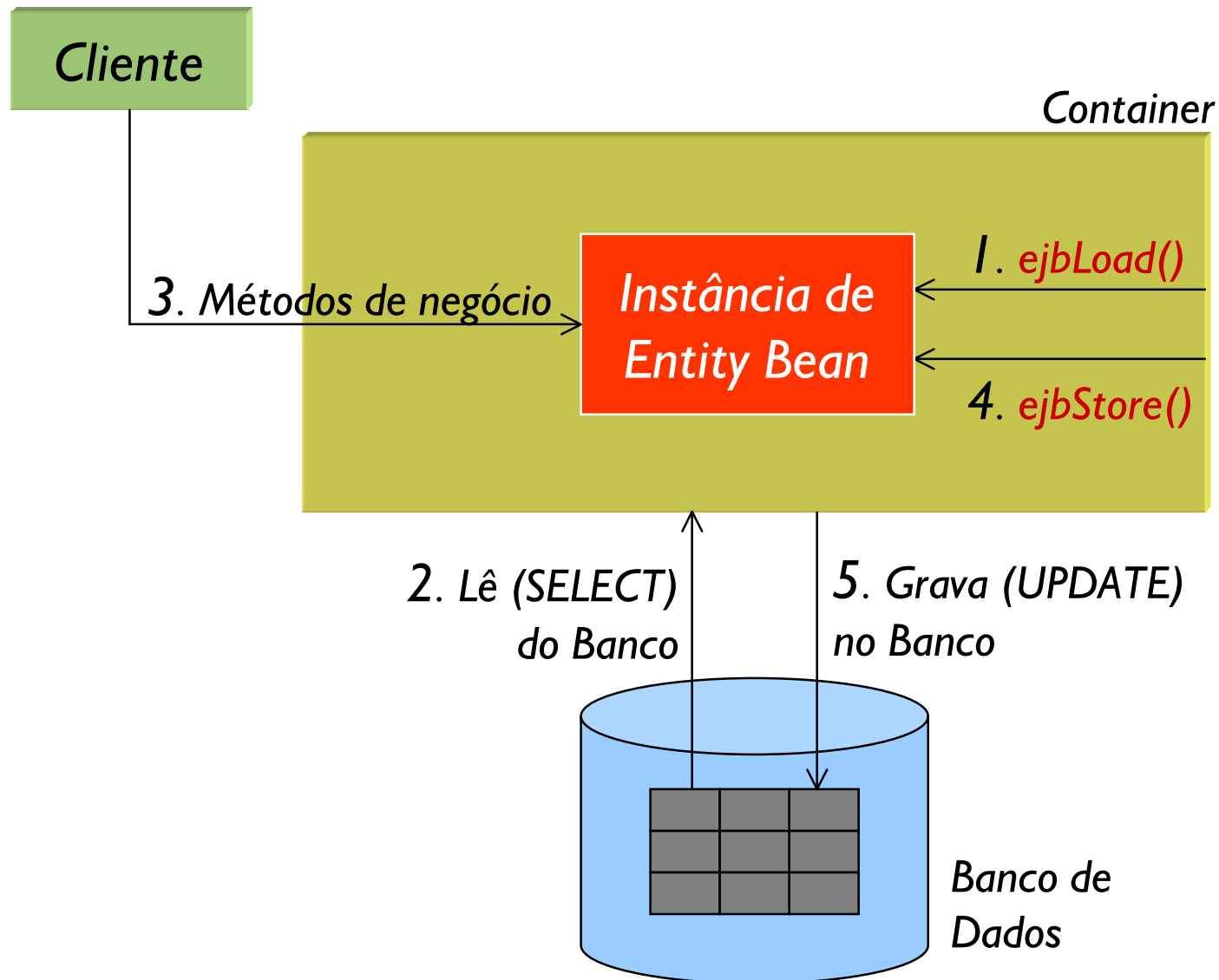
- Como qualquer EJB, um Entity Bean possui
 - Uma interface Home ou LocalHome
 - Uma interface Remote ou Local
 - Um deployment descriptor
 - Uma classe EJB
- As diferenças são
 - A classe do EntityBean está **mapeada** à uma definição de entidade de um esquema de banco de dados (uma tabela, por exemplo)
 - Há métodos especiais que lidam com a sincronização do EntityBean com o banco de dados
 - Um entity bean tem uma classe **Primary Key** que permite identificar univocamente sua instância

Entity Beans sobrevivem

- *Entity Beans são objetos persistentes*
 - *Sobrevivem a quedas do servidor ou banco de dados*
 - *Por serem apenas representação de dados de um meio de armazenamento, o bean pode ser reconstruído lendo seus dados do banco.*
- *Ciclo de vida muito longo*
 - *Existem enquanto existir a tabela ou entidade que representam seus dados no banco*
 - *Muitas vezes já existem antes da construção da aplicação*

- *Uma instância de um entity bean é uma visão do banco de dados*
- *A instância está sempre sincronizada com os dados do banco. Container possui mecanismo que faz a atualização após cada alteração*
 - *Todas as classes implementam métodos que realizam essa atualização*
 - *Container chama os métodos automaticamente*
- *void **ejbLoad()***
 - *lê os dados do banco para dentro do entity bean*
- *void **ejbStore()***
 - *grava os dados do entity bean no banco*

ejbLoad() e ejbStore()



Uso e reuso de Entity Beans

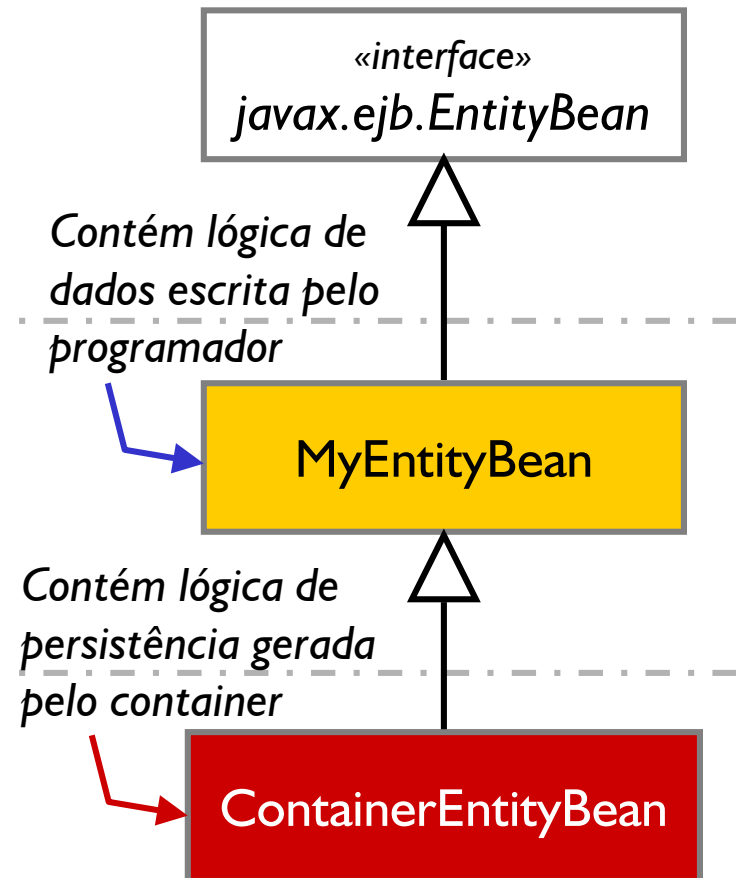
- *Instâncias diferentes de entity beans podem representar os mesmos dados*
 - *Transações garantem o isolamento necessário para evitar corrupção*
- *As mesmas instâncias podem ser reutilizadas para representar **dados diferentes** em momentos diferentes*
 - *Container gerencia ciclo de vida e ativação / passivação de Entity Beans, garantindo a sua disponibilidade*
 - *Programador pode controlar o que irá acontecer durante os métodos de sincronização ou pode deixar que o container se encarregue da tarefa.*

Duas formas de persistência

- *Entity beans podem ser associados ao meio de armazenamento de duas formas*
- **BMP** - *Bean Managed Persistence*
 - *O bean deve usar uma API (JDBC, SQLJ) para fazer as chamadas de UPDATE no `ejbStore()`, INSERT no `ejbCreate()`, SELECT no `ejbLoad()`, etc. (caso use banco relacional)*
- **CMP** - *Container Managed Persistence*
 - *O container faz um mapeamento dos objetos a entidades em um banco*
 - *Métodos de ciclo de vida, campos de dados e métodos de negócio são gerados automaticamente pelo container*

Características de EJB CMP

- Com *container-managed persistence* o programador do bean não implementa lógica de persistência
 - O bean não precisa usar APIs como JDBC
 - O container se encarrega de criar as tabelas (no deployment), criar os beans e os dados, removê-los e alterá-los
- O container gera automaticamente o JDBC ao estender a classe do bean criada pelo programador
 - O entity bean verdadeiro é o que foi gerado: a combinação da classe implementada pelo programador e a gerada pelo container



Entity Beans CMP não têm ...

- ... *campos de dados declarados*
 - São implementados na **subclasse*** (criada pelo container)
 - Programador declara seus campos no deployment descriptor (abstract persistence schema) usando `<cmp-field>`
- ... *implementação de métodos de acesso*
 - Programador declara todos os seus métodos de acesso (get/set) como **abstract**
 - Os nomes dos métodos devem ser compatíveis com os nomes dos campos declarados:

Se há `getCoisa()` no bean, e no deployment descriptor deve haver:
`<cmp-field><field-name>coisa</field-name></cmp-field>`
 - **Subclasse** gerada pelo container implementa os métodos (o container pode usar BMP, por exemplo)

Criação e remoção de Entity Beans

- Como Entity Beans *são* os dados, *criar* um bean é *inserir* dados em um banco; *remover* um bean é *apagar* esses dados
- Entity Beans são criados através do EJBObject (proxy) via sua interface Home.
- Se você tiver o seguinte *ejbCreate()* no seu bean

```
public AccountPK ejbCreate(String id, String who) {}
```
- Você deve ter o seguinte *create()* no seu Home

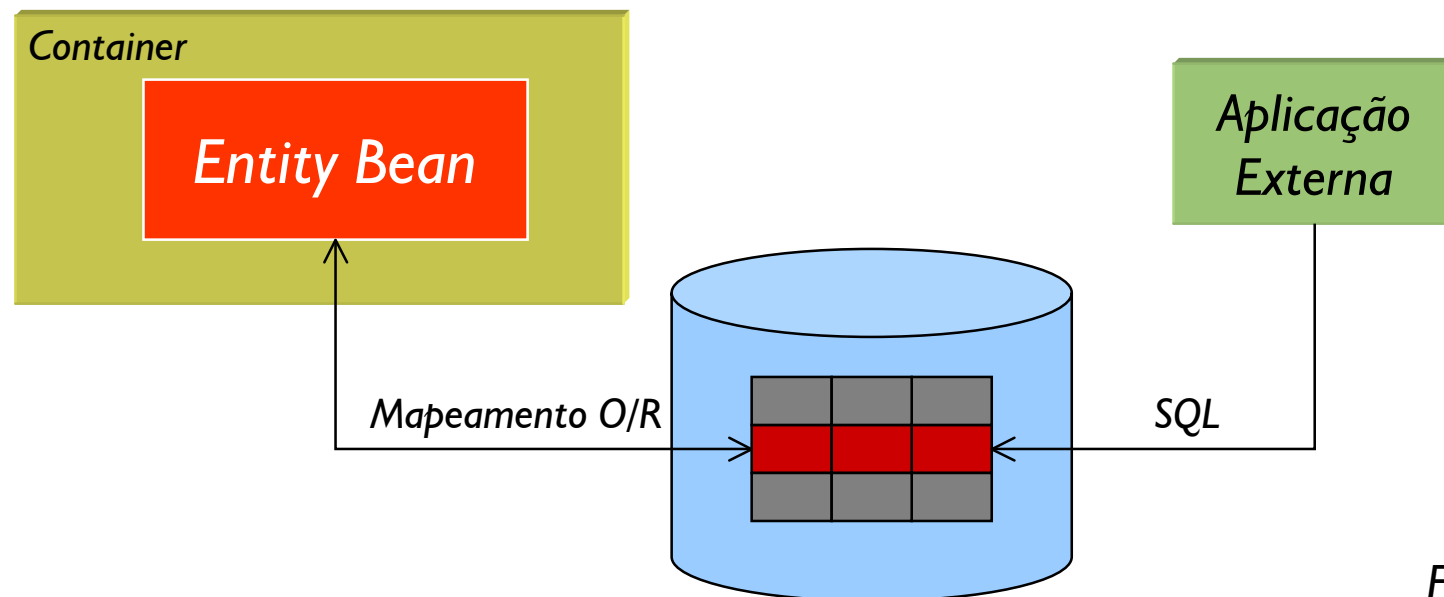
```
public Account create(String id, String who) {}
```
- Observe que os tipos retornados são diferentes
 - O bean retorna uma chave primária para o container (Home) que a utiliza para achar o bean para o cliente (que chamou create())
- Para destruir um bean (remover um registro), é preciso chamar *remove()* na interface Home

Entity Beans podem ser localizados

- Entity Beans nem sempre precisam ser criados.
Podem já existir no banco
 - Neste caso, devem ser *localizados*
 - Todo bean pode ser localizado via sua *chave primária*
 - Métodos *findBy()* adicionais podem ser criados para localizar beans de acordo com outros critérios
- *Finder methods*
 - Deve existir, no Home, pelo menos o *findByPrimaryKey()*
 - Métodos *create()* são opcionais
 - Se no home há um *findByXXX()* ...
... Se usa BMP, método é implementado na classe do bean e tem a forma: *ejbFindByXXX()*
 - ... Se usa CMP a sua lógica fica no *deployment descriptor*

Entity Beans podem ser modificados sem EJB

- Não é preciso usar EJB para modificar o conteúdo de um Entity Bean
 - Como Entity Beans representam dados externos, se esses dados puderem ser alterados externamente, sua mudança será refletida no Entity Bean
 - Riscos: **cache** de entity beans pode causar inconsistência (embora possível, a situação abaixo deve ser evitada)



Métodos da interface Home em Entity Beans

- Nos métodos abaixo, *Componente* pode ser a interface *Remote* (em *EJBHome*) ou *Local* (em *EJBLocalHome*) do bean
 - Nas interfaces *Remote*, todos provocam *RemoteException* também
- **Componente findByPrimaryKey(PK)**
throws **FinderException**
 - Único método obrigatório de *Home*
- **Componente create(...)** throws **CreateException**
 - Retorna uma referência para o bean criado
- **Componente findByXxx(...)** throws **FinderException**
- **Collection findByXxx(...)** throws **FinderException**
 - Retorna referência para o componente ou *Collection* de referências caso o *find* retorne mais de um objeto
- **Tipo xxx(...)**
 - Método que não se refere a nenhuma instância particular
 - Deve ter um correspondente **ejbHomeXxx()** no bean com mesmos argumentos e tipo de retorno

- Como não há como fazer queries JDBC com entity beans CMP, há uma linguagem para fazer queries nos próprios beans: **Enterprise JavaBean Query Language (EJB-QL)**
- EJB-QL é uma linguagem com sintaxe semelhante a SQL para pesquisar entity beans. Possui
 - cláusula **SELECT**
 - cláusula **FROM**
 - cláusula **WHERE**, opcional
- EJB-QL é colocado nos deployment descriptors no contexto da declaração do método
 - Todos os finders são implementados com EJB-QL
- O servidor transforma expressões EJB-QL em SQL e realiza as chamadas no banco de dados
 - Isto pode ser configurado no servidor usando recursos proprietários

Métodos da interface *EntityBean* (BMP/CMP)

Métodos do ciclo de vida das instâncias

- `void ejbActivate()`
 - Chamado logo após a ativação do bean
- `void ejbPassivate()`
 - Chamado antes da passivação do bean
- `void setEntityContext(EntityContext ctx)`
 - Chamado após a criação da instância no pool. O contexto passado deve ser gravado em variável de instância pois pode ser usado para obter a chave primária da instância atual
- `void unsetEntityContext()`
 - Destrói a instância (será recolhida pelo GC). Isto destrói o objeto, mas não o entity bean (que são os dados)

Métodos da especificação em BMP

Métodos que implementam a interface Home

- PK **ejbFindByXxx(...)** throws **FinderException**
 - Deve haver pelo menos **ejbFindByPrimaryKey()**, que devolve a própria **PrimaryKey** (e faz um query de identidade para verificar se a PK existe no banco: **SELECT pk where pk = pk**)
 - Cada **ejbFind()** devolve ou a **Primary Key** ou uma **Collection** de PKs
- PK **ejbCreate(...)** throws **CreateException**
 - Pode haver zero ou mais (diferindo pelo número e tipo de args)
 - Deve criar o objeto (**INSERT** ou equivalente) e retornar seu PK
- **void ejbPostCreate(...)**
 - Deve haver um para cada **ejbCreate()** com mesmos argumentos
 - Chamado após o create e pode ser vazio
- Tipo **ejbHomeXxx(...)**
 - Métodos de Home que não se referem a nenhuma instância em particular. **xxx()** é o nome do método na interface Home.

BMP: Relacionamento Home-Bean

- Os métodos de *EJBHome* à esquerda delegam chamadas aos métodos de *EntityBean* à direita
 - Ou seja, se você tiver os métodos à esquerda no seu Home, deve ter os métodos à direita no seu Bean

Interface Home	EJB
Objeto <code>findByPrimaryKey(ObjetoPK)</code>	ObjetoPK <code>ejbFindByPrimaryKey(ObjetoPK)</code>
Objeto <code>findByParams(Par1, Par2)</code>	ObjetoPK <code>ejbFindByParams(Par1, Par2)</code>
Collection <code>findByParams(Par1, Par2)</code>	Collection <code>ejbFindByParams(Par1, Par2)</code>
Objeto <code>create(Par1, Par2)</code>	ObjetoPK <code>ejbCreate(Par1, Par2)</code> void <code>ejbPostCreate(Par1, Par2)</code>
Tipo <code>operacao(Par1, Par2)</code>	Tipo <code>ejbHomeOperacao(Par1, Par2)</code>

- *Chave*
 - *Objeto*: interface do objeto remoto
 - *ObjetoPK*: chave primária do objeto remoto
 - *Par1*, *Par2* e *Tipo*: tipos primitivos, remotos ou serializáveis

Métodos da interface *EntityBean* em *BMP*

Métodos de persistência e sincronização com o banco

- `void ejbLoad()`
 - Deve conter query **SELECT** ou equivalente e em seguida atualizar os atributos do bean com os dados recuperados
 - Use `context.getPrimaryKey()` para saber qual a chave primária do bean a ser lido
- `void ejbStore()`
 - Deve conter query **UPDATE** ou equivalente e gravar no banco o estado atual dos atributos do objeto
- `void ejbRemove()`
 - Chamado antes que os dados sejam removidos do banco
 - Deve conter query **DELETE** ou equivalente
 - Use `context.getPrimaryKey()` para saber qual a chave primária do bean a ser removido

Componentes de um EJB-JAR que usa CMP

- Interfaces *Home* e *Component*
 - *Idênticas a um bean que usa BMP*
- Classe *Primary Key*
 - *Idênticas às interfaces e classe Primary Key de um entity bean equivalente que usa BMP*
- Classe *Enterprise bean* (muito menor que classe BMP)
 - *Métodos `ejbLoad()`, `ejbStore()` não possuem código de persistência*
 - *Classe é **abstrata**; métodos de acesso (`get/set`) são abstratos; métodos `ejbSelectXXX()` são abstratos*
 - *Métodos de negócio contém apenas lógica de negócio*
 - *Métodos `finder` não são declarados*
- *Deployment descriptor* (maior que versão BMP)
 - *Definição da lógica de métodos e relacionamentos*
- *Arquivos de configuração do fabricante (JBoss)*
 - *Pode ter mapeamento de tipos, mapeamento de esquema*

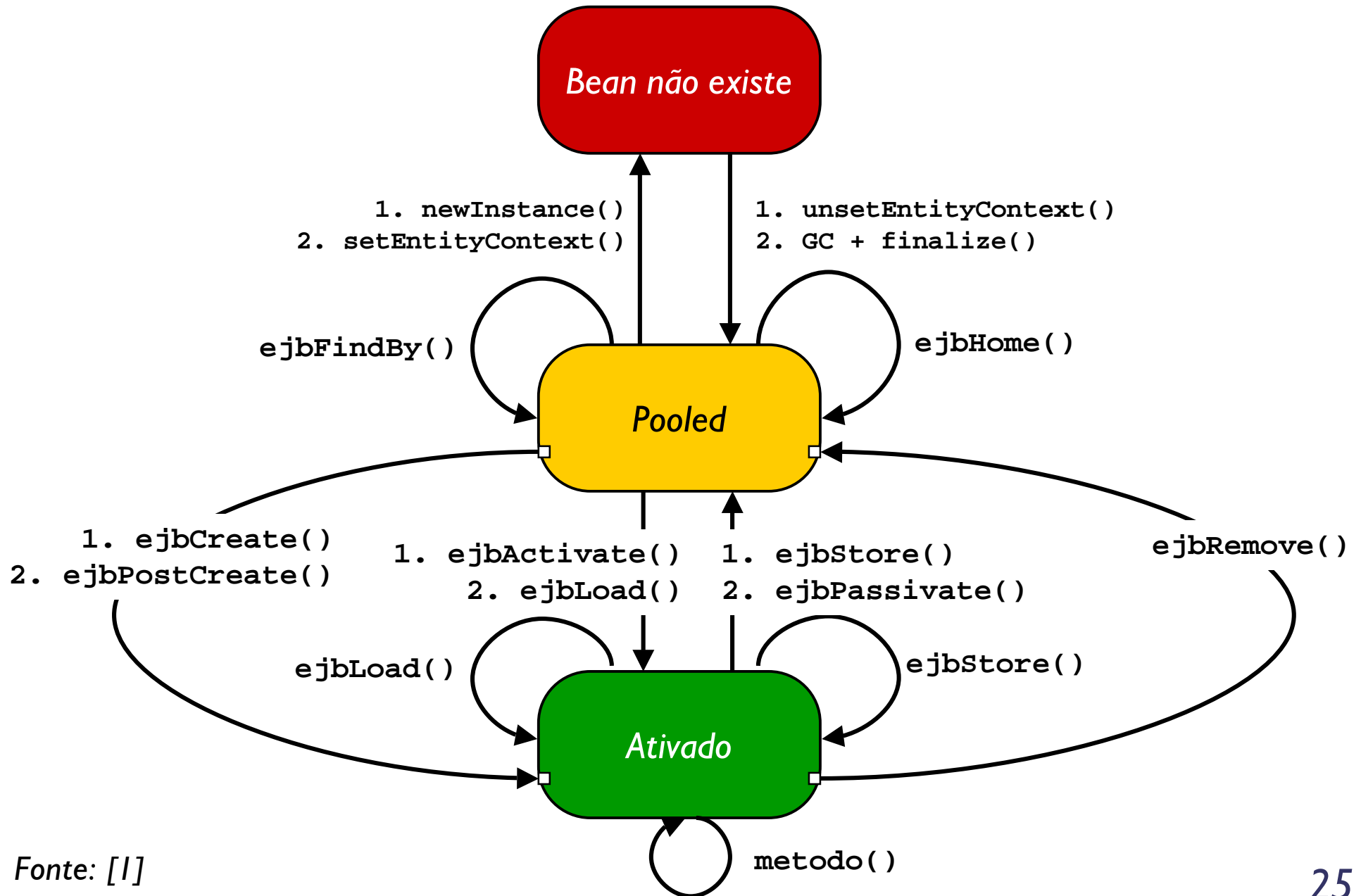
CMP: diferenças nos métodos padrão

- **void `ejbFindXXX(...)`**
 - *Não são implementados no bean em CMP*
 - *Finders são declarados no DD e têm lógica em EJB-QL*
- **abstract void `ejbSelectXXX(...)`;**
 - *Não existem em beans BMP*
 - *Usados para fazer queries genéricos para o bean*
 - *É método de negócio mas não pode ser chamado pelo cliente*
 - *Declarados como **abstract** com lógica descrita no DD em EJB-QL*
- **void `ejbHomeXXX(...)`**
 - *Usado para definir métodos não específicos a uma instância*
 - *Deve chamar um ou mais `ejbSelect()` para realizar as operações*
- **PK `ejbCreate(...)`**
 - *Chama os métodos `setXXX()` abstratos inicializando-os com os dados que serão usados pelo container para criar novo registro*
 - *Deve retornar **null***
- **`ejbLoad()` , `ejbRemove()` e `ejbStore()`**: vazios

Diferenças BMP-CMP

Diferença	Container-Managed Persistence	Bean-Managed Persistence
<i>Definição da classe</i>	<i>Abstrata</i>	<i>Concreta</i>
<i>Chamadas de acesso ao banco de dados</i>	<i>Gerada pelas ferramentas no deployment</i>	<i>Codificada pelo programador</i>
<i>Estado persistente</i>	<i>Representadas como campos persistentes virtuais</i>	<i>Codificadas como variáveis de instância</i>
<i>Métodos de acesso a campos persistentes e relacionamentos</i>	<i>Obrigatórios (abstract)</i>	<i>Não há</i>
<i>Método findByPrimaryKey</i>	<i>Gerado pelo container</i>	<i>Codificado pelo programador</i>
<i>Métodos finder customizados</i>	<i>Gerados pelo container mas programador escreve EJB-QL</i>	<i>Codificado pelo programador</i>
<i>Métodos select</i>	<i>Gerados pelo container</i>	<i>Não há</i>
<i>Valor de retorno de ejbCreate()</i>	<i>Deve ser null</i>	<i>Deve ser a chave primária</i>

Ciclo de vida de um Entity Bean



Exemplo: Interface do Componente Remote


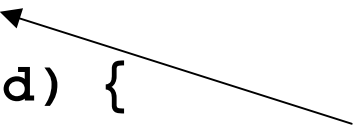
```
public interface Account extends EJBObject {
    public void deposit(double amt)
        throws AccountException, RemoteException;
    public void withdraw(double amt)
        throws AccountException, RemoteException;

    public double getBalance()
        throws RemoteException;
    public String getOwnerName()
        throws RemoteException;
    public void setOwnerName(String name)
        throws RemoteException;
    public String getAccountID()
        throws RemoteException;
    public void setAccountID(String id)
        throws RemoteException;
}
```

Exemplo: Interface Home

```
public interface AccountHome extends EJBHome {  
  
    Account create(String accountID, String ownerName)  
        throws CreateException, RemoteException;  
    ↘  
    public Account findByPrimaryKey(AccountPK key)  
        throws FinderException, RemoteException;  
    ↘  
    public Collection findByOwnerName(String name)  
        throws FinderException, RemoteException;  
  
    public double getTotalBankValue()  
        throws AccountException, RemoteException;  
  
}
```

Exemplo: Primary Key

```
public class AccountPK implements java.io.Serializable {  
     public String accountID;   
    public AccountPK(String id) {  
        this.accountID = id;  
    }  
    public AccountPK() { }  
    public String toString() {  
        return accountID;  
    }  
    public int hashCode() {  
        return accountID.hashCode();  
    }  
    public boolean equals(Object account) {  
        return  
            ( (AccountPK)account )  
                .accountID.equals(accountID);  
    }  
}
```

*Mesmo identificador
usado no Bean
(essencial, para CMP)*

Exemplo: Bean usando BMP

```
public class AccountBean implements EntityBean {
    protected EntityContext ctx;

    private String accountID;        // PK
    private String ownerName;
    private double balance;

    // Getters e Setters
    public String getAccountID() {...} ...
    // Métodos de negócio
    public void deposit(double amount) {...} ...

    // Métodos do ciclo de vida
    public void ejbActivate() {...} ...
    // Métodos de Home
    public AccountPK findByPrimaryKey(AccountPK pk) {}
    // Métodos de persistência
    public void ejbLoad() {...} ...
}
```

BMP: Acesso ao banco via DataSource

```
private Connection getConnection() throws Exception {
    try {
        Context ctx = new InitialContext();
        javax.sql.DataSource ds =
            (javax.sql.DataSource)
                ctx.lookup("java:comp/env/jdbc/ejbPool");
        return ds.getConnection();
    } catch (Exception e) {
        System.err.println("Could not locate datasource:");
        e.printStackTrace();
        throw e;
    }
}
```

Veja o nome que foi usado no lookup e compare com o nome declarado como <resource-ref> no ejb.jar.xml e jboss.xml

Métodos do ciclo de vida (BMP/CMP)

```
public void setEntityContext(EntityContext ctx) {  
    System.out.println("setEntityContext called");  
    this.ctx = ctx;  
}
```

```
public void unsetEntityContext() {  
    System.out.println("unsetEntityContext called");  
    this.ctx = null;  
}
```

```
public void ejbPassivate() {  
    System.out.println("ejbPassivate () called.");  
}
```

```
public void ejbActivate() {  
    System.out.println("ejbActivate() called.");  
}
```

BMP: Métodos de negócio e get/set

```
public void deposit(double amt) throws AccountException {
    balance += amt;
}

public void withdraw(double amt) throws AccountException {
    if (amt > balance) {
        throw new AccountException("Cannot withdraw "+amt+"!");
    }
    balance -= amt;
}

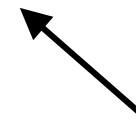
// Getter/setter methods on Entity Bean fields
public double getBalance() {
    return balance;
}
public void setOwnerName(String name) {
    ownerName = name;
}
public String getOwnerName() {
    return ownerName;
}
...
```

Estes métodos são a razão de existir do bean! E são os mais simples!

BMP - Métodos de Home: findByPrimaryKey



```
public AccountPK ejbFindByPrimaryKey(AccountPK key)
    throws FinderException {
    PreparedStatement pstmt = null;
    Connection conn = null;
    try {
        conn = getConnection();
        pstmt = conn.prepareStatement
            ("select id from accounts where id = ?");
        pstmt.setString(1, key.toString());
        ResultSet rs = pstmt.executeQuery();
        rs.next();
        return key;
    } catch (Exception e) {
        throw new FinderException(e.toString());
    } finally { conn.close(); ... }
}
```



*Query só para saber
se PK existe no banco!*

BMP - Métodos de Home: create()

```
public AccountPK ejbCreate(String accountID, String ownerName)
    throws CreateException {

    PreparedStatement pstmt = null;
    Connection conn = null;
    try {
        this.accountID = accountID;
        this.ownerName = ownerName;
        this.balance = 0;
        conn = getConnection();
        pstmt = conn.prepareStatement("insert into accounts "+
            "(id, ownerName, balance) values (?, ?, ?)");
        pstmt.setString(1, accountID);
        pstmt.setString(2, ownerName);
        pstmt.setDouble(3, balance);
        pstmt.executeUpdate();
        return new AccountPK(accountID);
    } catch (Exception e) {
        throw new CreateException(e.toString());
    } finally { conn.close(); ... }
}

public void ejbPostCreate(String accountID, String ownerName) {
}
```

*Não esqueça de implementar **ejbPostCreate()***

BMP - Métodos de Home: findByXXX()

```
public Collection ejbFindByOwnerName(String name)
    throws FinderException {
    PreparedStatement pstmt = null;
    Connection conn = null;
    Vector v = new Vector();

    try {
        conn = getConnection();
        pstmt = conn.prepareStatement
            ("select id from accounts where ownerName = ?");
        pstmt.setString(1, name);
        ResultSet rs = pstmt.executeQuery();

        while (rs.next()) {
            String id = rs.getString("id");
            v.addElement(new AccountPK(id));
        }
        return v;
    } catch (Exception e) {
        throw new FinderException(e.toString());
    } finally { con.close(); ... }
}
```

BMP - Métodos de Home: xxx()

```
public double ejbHomeGetTotalBankValue()
    throws AccountException {
    PreparedStatement pstmt = null;
    Connection conn = null;
    try {
        conn = getConnection();
        pstmt = conn.prepareStatement
            ("select sum(balance) as total from accounts");
        ResultSet rs = pstmt.executeQuery();
        if (rs.next()) {
            return rs.getDouble("total");
        }
    } catch (Exception e) {
        e.printStackTrace();
        throw new AccountException(e);
    } finally {...}
    throw new AccountException("Error!");
}
```

BMP: `ejbRemove()`


```
public void ejbRemove() throws RemoveException
    AccountPK pk = (AccountPK) ctx.getPrimaryKey();
    String id = pk.accountID;

    PreparedStatement pstmt = null;
    Connection conn = null;
    try {
        conn = getConnection();
        pstmt = conn.prepareStatement
            ("delete from accounts where id = ?");
        pstmt.setString(1, id);
        if (pstmt.executeUpdate() == 0) {
            throw new RemoveException("...");
        }
    } catch (Exception ex) {
        throw new EJBException("...", ex);
    } finally {...conn.close() ... }
}
```

BMP: ejbStore()

```
public void ejbStore() {
    PreparedStatement pstmt = null;
    Connection conn = null;
    try {
        conn = getConnection();
        pstmt = conn.prepareStatement
            ("update accounts set ownerName = ?,
            balance = ? where id = ?");
        pstmt.setString(1, ownerName);
        pstmt.setDouble(2, balance);
        pstmt.setString(3, accountID);
        pstmt.executeUpdate();
    } catch (Exception ex) {
        throw new EJBException("...", ex);
    } finally {
        ...conn.close() ...
    }
}
```

*Sempre provoque EJBException
quando algo sair errado em
ejbStore() e ejbLoad()*



BMP: ejbLoad()

```
public void ejbLoad() {
    AccountPK pk = (AccountPK) ctx.getPrimaryKey();
    accountID = pk.accountID;
    PreparedStatement pstmt = null;
    Connection conn = null;
    try {
        conn = getConnection();
        pstmt = conn.prepareStatement
            ("select ownerName, balance from accounts where id = ?");
        pstmt.setString(1, accountID );
        ResultSet rs = pstmt.executeQuery();
        rs.next();
        ownerName = rs.getString("ownerName");
        balance = rs.getDouble("balance");
    } catch (Exception ex) {
        throw new EJBException("...", ex);
    } finally {...conn.close() ... }
}
```

Em BMP, use um DAO

- *Nos exemplos anteriores, o código JDBC foi usado dentro dos métodos de persistência do bean*
 - *Esta prática é razoável como exemplo, mas não deve ser usada em produção*
 - *Dificulta a manutenção e mistura lógica do banco com a lógica de negócios do bean*
- **Use um DAO - *Data Access Object***
 - *Com um DAO, cada chamada a método de persistência é delegada a um método de um DAO, que realiza as operações desejadas*
 - *O DAO pode encapsular toda a lógica de acesso e pode até usar outro mecanismo de persistência sem quebrar a aplicação.*

Exemplo: Deployment Descriptor

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>Account</ejb-name>
      <home>examples.AccountHome</home>
      <remote>examples.Account</remote>
      <local-home>examples.AccountLocalHome</local-home>
      <local>examples.AccountLocal</local>
      <ejb-class>examples.AccountBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>examples.AccountPK</prim-key-class>
      <reentrant>False</reentrant>
      <resource-ref>
        <res-ref-name>jdbc/ejbPool</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
      </resource-ref>
    </entity>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>Account</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

Este é o nome usado no ENC deste bean para a fonte de dados JDBC. O *jboss.xml* faz o mapeamento a uma fonte de dados local

Fonte de dados onde bean está armazenado

Declaração da política de transações é obrigatória em Entity Beans pois SEMPRE usam transações controladas pelo container

Exemplo: jboss.xml

```
<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>Account</ejb-name>
      <jndi-name>account/AccountHome</jndi-name>
    </entity>
  </enterprise-beans>
  <resource-managers>
    <resource-manager>
      <res-name>jdbc/ejbPool</res-name>
      <res-jndi-name>java:/DefaultDS</res-jndi-name>
    </resource-manager>
  </resource-managers>
</jboss>
```

Chamado por clientes dentro e fora do ENC

Veja chamada no bean

Execução do Exemplo

- *Para executar a aplicação exemplo, use o Ant no diretório `cap06/exemplos/mejb2`*
 - *Configure o `build.properties` se necessário*
- *Etapas*
 - *Povoar banco de dados: ant create.table*
 - *Deployment: ant jboss.deploy*
 - *Rodar o cliente: ant run.jboss.client*
- *Manhêna a janela do JBoss visível para ver simultaneamente as chamadas no cliente e servidor*
- *Exemplo do Tutorial da Sun (semelhante)*
 - *`cap07/exemplos/sun/bmp`*

- *I. Implementar EntityBean que represente uma tabela com o seguinte esquema*
 - *a) Implemente, no pacote loja, as classes **Produto**, **ProdutoPK**, **ProdutoHome**, **ProdutoBean** (algumas classes já estão parcialmente prontas). Use o **ProdutoDAO** e **ProdutoVO** fornecidos para implementar métodos de **ProdutoBean** que acessam o banco*
 - *b) Implemente o método **create()** e os **finders** definidos na interface **Home** (use os métodos do **DAO**)*
 - *c) Configure a chamada JNDI no **DAO** para que localize o **DataSource** usando o nome **jdbc/produtos** no **ENC***
 - *d) Preencha os tags em **jboss.xml** e **ejb-jar.xml***
 - *e) Depois do deploy, termine o cliente e teste a aplicação*

```
create table produtos (  
    id varchar(16) primary key,  
    nome varchar(32),  
    preco numeric(18),  
    qte integer  
);
```

Exemplo de CMP: Como executar

- Diretório *cap06/exemplos/mejb/*
- Configuração
 - Configure o arquivo *build.properties* com seu ambiente
 - Inicie o JBoss
- Inicialização
 - > *ant drop.table*
 - > *ant clean*
- Deployment
 - > *ant jboss.deploy* (*observe as mensagens de criação das tabelas no JBoss*)
- Execução do cliente
 - > *ant run jboss.client*
 - > *ant select.all* (*faz SELECT na tabela*)

Deployment Descriptor (I)

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>ProdutoEJB</ejb-name>
      <home>loja.ProdutoHome</home>
      <remote>loja.Produto</remote>
      <ejb-class>loja.ProdutoBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>loja.ProdutoPK</prim-key-class>

      <reentrant>False</reentrant>

      <cmp-version>2.x</cmp-version>
      <abstract-schema-name>
        Produto
      </abstract-schema-name>

      ...
```

CMP: Enterprise Bean (é só isto!)

```
public abstract class ProdutoBean implements EntityBean {
    protected EntityContext ctx;

    public abstract String getName();
    public abstract void setName(String name);
    public abstract String getCodigo();
    public abstract void setCodigo(String codigo);
    (...)
    public void ejbStore() {}
    public void ejbLoad() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void setEntityContext(EntityContext c) {ctx = c;}
    public void unsetEntityContext() {ctx = null;}
    public void ejbPostCreate(String codigo, String name,...) {}
    public void ejbCreate(String codigo, String name, ...) {
        setCodigo(codigo);
        setName(name);
        (...)
    }
}
```

Métodos de negócio abstratos (lógica no deployment descriptor)

ejbLoad e *ejbStore* são implementados pelo container

ejbFindBy não são declarados (lógica está no deployment descriptor)

ejbCreate chama métodos abstratos para passar dados recebidos

Deployment Descriptor (2)

```
...
<cmp-field>
  <field-name>codigo</field-name>
</cmp-field>
<cmp-field><field-name>name</field-name></cmp-field>
<cmp-field>
  <field-name>preco</field-name>
</cmp-field>
<cmp-field>
  <field-name>quantidade</field-name>
</cmp-field>
<!--
<primkey-field>codigo</primkey-field>
-->
<query>
  ... declarações EJB-QL
</query>
</entity>
</enterprise-beans>
```

Se PK for um campo *unidimensional* em vez de uma classe (String, por exemplo) use este elemento e indique o seu tipo no <prim-key-class> (java.lang.String, por exemplo)

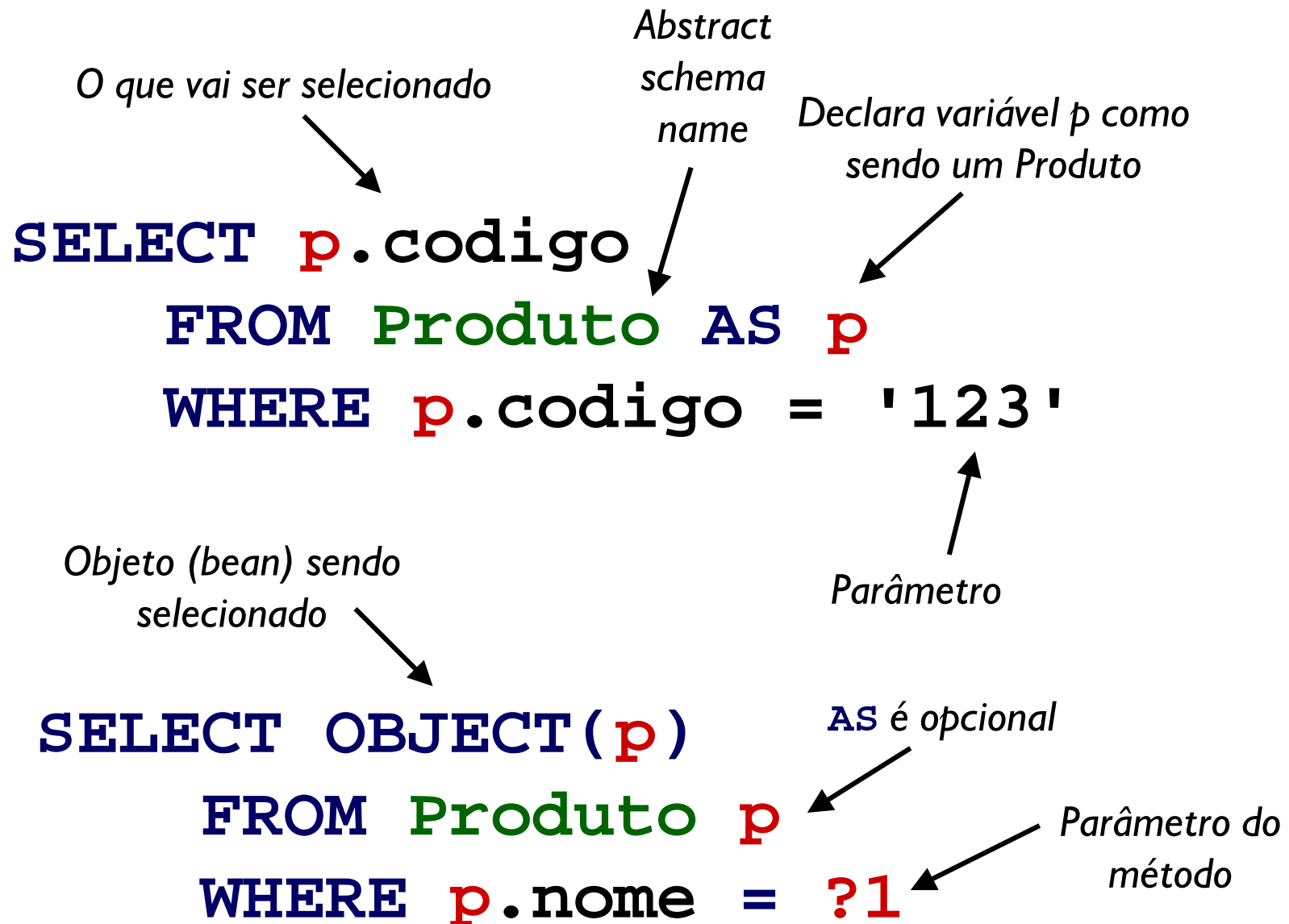
...

Primary Key em CMP

- A Primary Key de um bean CMP, se implementada com uma classe, deve ter campos de dados públicos com os **mesmos identificadores** que os campos persistentes que a compõem
 - Por exemplo, se uma **PK simples** é representada pelo campo **codigo**, deve ter uma variável de instância **codigo** e ela deve ser **pública**
 - Se for uma **PK composta**, cada um dos campos que a representa deve ter o mesmo identificador que seus atributos públicos
 - Se esta regra não for observada, o container não terá como descobrir quem é a Primary Key do objeto
- Os métodos que alteram atributos pertencentes a uma PK **não devem ser expostos** através das interfaces e só devem ser chamados uma vez dentro do bean
 - Bean deve ter par get/set para cada atributo, mas interface Local ou Remote só deve ter o método get().

- *Toda pesquisa no banco é feita através de Enterprise JavaBeans Query Language (EJB-QL)*
- *O servidor mapeia expressões EJB-QL a expressões SQL para realizar as pesquisas no banco*
- *EJB-QL é usado no deployment descriptor*
 - *Não há SQL ou EJB-QL na classe do bean. O bean lida apenas com lógica de negócio*
 - *O bean pode usar métodos que retornam queries customizados, previamente definidos no deployment descriptor em EJB-QL*

Exemplos de EJB-QL



Palavras-chave EJB-QL

- Usadas na cláusula **SELECT**
 - **DISTINCT, OBJECT**
- Usadas na cláusula **FROM**
 - **AS, IN**
- Usadas na cláusula **WHERE**
 - **AND, OR, NOT**
 - **BETWEEN**
 - **LIKE, IS, MEMBER, OF**
 - **TRUE, FALSE**
 - **EMPTY, NULL**
 - **UNKNOWN**

Cláusula SELECT

- A cláusula **SELECT** informa o que se deseja obter com a pesquisa. Pode retornar
 - Objetos (interfaces locais ou remotas)
 - Atributos dos objetos (tipos dos campos <cmp-field>)
- A palavra **SELECT** pode ser seguida de **DISTINCT** para eliminar valores duplicados nos resultados
- **SELECT** utiliza uma variável declarada na cláusula **FROM**
- Se retorna um objeto, é preciso usar **OBJECT(variavel)**
- Exemplos

```
SELECT OBJECT(p) FROM Produto p
```

Retorna
objetos

```
SELECT DISTINCT OBJECT(p) FROM Produto p
```

```
SELECT p.codigo FROM Produto p
```

Retorna
campos

```
SELECT DISTINCT p.codigo FROM Produto p
```

Cláusula WHERE

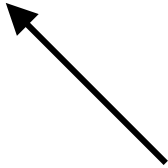
- A cláusula **WHERE** é opcional e restringe os resultados da pesquisa com base em uma ou mais expressões condicionais concatenadas
- As expressões podem usar
 - Literais (strings, booleanos ou números)
 - Identificadores (declarados no FROM)
 - Operadores
 - Funções
 - Parâmetros do método que utiliza a pesquisa (**?1**, **?2**,...)
- Literais
 - Strings são representados entre apóstrofes: **'nome'**
 - Números têm mesmas regras de literais Java **long** e **double**
 - Booleanos são **TRUE** e **FALSE** (case-insensitive)

- *Expressões matemáticas*
 - *+, -, *, /*
- *Expressões de comparação*
 - *=, >, >=, <, <=, <>*
- *Operadores lógicos*
 - **NOT, AND, OR**
- *Outros operadores*
 - **BETWEEN, NOT BETWEEN**
 - **IN, NOT IN**
 - **LIKE, NOT LIKE**
 - **NULL, NOT NULL**
 - **IS EMPTY, IS NOT EMPTY**
 - **MEMBER, NOT MEMBER**
- *Operadores do LIKE*
 - *_ representa um único caractere*
 - *% representa uma seqüência de zero ou mais caracteres*
 - *\ caractere de escape (necessário para usar _ ou %) literalmente*

- **CONCAT** (String, String)
 - *Retorna String. Concatena dois strings*
- **SUBSTRING**(String, int start, int length)
 - *Retorna String. Parte um string em um string menor*
- **LOCATE**(String, String, [start])
 - *Retorna um inteiro mostrando onde um string está localizado dentro de outro String*
- **LENGTH**(String)
 - *Retorna um inteiro com o comprimento do String*
- **ABS** (numero)
 - *Retorna o valor absoluto de um número (int, double)*
- **SQRT**(double numero)
 - *Retorna a raiz quadrada de um número (double)*
- **MOD**(int, int)
 - *Retorna inteiro com o resto da divisão*

Não existem em EJB-QL 2.0

- *Várias funções e tipos comuns em SQL não existem em EJB-QL 2.0. Para obter o mesmo efeito, é preciso às vezes recorrer à combinação de métodos `ejbSelect()` ou mesmo a BMP*
 - *Em último caso, pode-se recorrer a extensões proprietárias sacrificando-se a portabilidade*
- **Funções ausentes**
 - **ORDER BY**
 - **MIN, MAX**
 - **COUNT**
 - **SUM**
- **Tipos ausentes**
 - **DATE** (use **long** para guardar datas)



Se puder, use um servidor que suporte EJB 2.1 para não comprometer a independência de fabricante

Exemplos de EJB-QL (de [1])

- *Encontre todos os produtos que são chips e cuja margem de lucro é positiva*
 - **SELECT OBJECT(p)**
FROM Produto p
WHERE (p.descricao = 'chip'
AND (p.preco - p.custo > 0)
- *Encontre todos os produtos cujo preço é pelo menos 1000 e no máximo 2000*
 - **SELECT OBJECT(p)**
FROM Produto p
WHERE p.preco BETWEEN 1000 AND 2000
- *Encontre todos os produtos cujo fabricante é Sun ou Intel*
 - **SELECT OBJECT(p)**
FROM Produto p
WHERE p.fabricante IN ('Intel', 'Sun')

Exemplos de EJB-QL (2)

- *Encontre todos os produtos com IDs que começam com 12 e terminam em 3*
 - **SELECT OBJECT(p)**
FROM Produto p
WHERE p.id LIKE '12%3'
- *Encontre todos os produtos que têm descrições null*
 - **SELECT OBJECT(p)**
FROM Produto p
WHERE p.descricao IS NULL
- *Encontre todos os pedidos que não têm itens (coleção)*
 - **SELECT OBJECT(pedido) FROM Pedido pedido**
WHERE pedido.itens IS EMPTY
- *Encontre todos os itens ligados a pedidos em coleção*
 - **SELECT OBJECT(item)**
FROM Pedido pedido, Item item
WHERE item IS MEMBER pedido.itens

<query> EJB-QL

```
<query>
  <query-method>
    <method-name>findByName</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </query-method>
  <ejb-ql><![CDATA[ SELECT OBJECT(obj)
FROM ProdutoBean obj WHERE obj.name = ?1 ]]></ejb-ql>
</query>
  ...
<query>
  <query-method>
    <method-name>findAllProducts</method-name>
  </query-method>
  <ejb-ql><![CDATA[
    SELECT OBJECT(obj) FROM ProdutoBean AS obj
    WHERE obj.codigo IS NOT NULL
  ]]></ejb-ql>
</query>
```

Nome de método de Home ou Remote

Parâmetro recebido

abstract schema name

Nome de um cmp-field

AS é opcional

findByPrimaryKey não deve ser declarado!

JBoss deployment descriptor

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE jboss PUBLIC "-//JBoss//DTD JBOSS 3.0//EN"
           "http://localhost/dtd/jboss_3_0.dtd">

<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>ProdutoEJB</ejb-name>
      <jndi-name>loja/produtos</jndi-name>
    </entity>
  </enterprise-beans>
  <resource-managers>
    <resource-manager>
      <res-name>jdbc/LojaDB</res-name>
      <res-jndi-name>java:/DefaultDS</res-jndi-name>
    </resource-manager>
  </resource-managers>
</jboss>
```

Nome JNDI Global

*Nome do ENC **java:comp/env***

*Nome JNDI global domínio **java:** local*

JBoss CMP: jbosscmp-jdbc.xml

- Este arquivo é necessário se a aplicação irá se comunicar com fonte de dados que tem esquema próprio

```
<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>ProdutoEJB</ejb-name>
      <table-name>produtos</table-name>
      <cmp-field>
        <field-name>nome</field-name>
        <column-name>nome</column-name>
      </cmp-field>
      <cmp-field>
        <field-name>codigo</field-name>
        <column-name>id</column-name>
      </cmp-field>
      <cmp-field>
        <field-name>quantidade</field-name>
        <column-name>qte</column-name>
      </cmp-field>
      ...
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

Campos CMP →

Tabela que será usada (ou criada) no banco

Campos do banco de dados

Pode-se ainda redefinir o mapeamento de *tipos* para todos os beans no arquivo de configuração *standardjbosscmp-jdbc.xml*

- Método de pesquisa, como um finder, mas que não é exposto ao cliente através da interface Home
 - É **privativo** ao bean: usado internamente como um método auxiliar na realização de pesquisas genéricas
 - É declarado no bean como **abstract** (para que possa ser usado pelos outros métodos) e provoca *FinderException*
- Com **ejbSelect()** é possível realizar pesquisas e devolver objetos e também valores não relacionados com objetos
 - Finders são limitados à seleção de objetos (beans) e coleções de objetos
 - Seletores podem retornar qualquer coisa

- Para implementar métodos `ejbHome()` em CMP é preciso recorrer a métodos `ejbSelect()`
 - Use métodos `ejbSelect()` dentro de `ejbHome()` para selecionar dados e devolver os resultados desejados
- Exemplo: `getProximoCodigo()` em `Home`, é implementado no bean como

```
public String.ejbHomeGetProximoCodigo() {
    Collection codigos = this.ejbSelectCodigos();
    Iterator it = codigos.iterator();
    int max = 0;
    while(it.hasNext()) {
        int cod = Integer.parseInt((String)it.next());
        if (cod > max) max = cod;
    }
    return "" + (max+1);
}
```

Exemplo de ejbSelect()

- No bean, ejbSelect() são declarados para que métodos ejbHome() possam usá-los

```
public abstract ejbSelectCodigos();
```

- No deployment descriptor, são definidos em EJB-QL

```
<query>
  <query-method>
    <method-name>ejbSelectCodigos</method-name>
  </query-method>
  <ejb-ql><![CDATA[
    SELECT obj.codigo FROM ProdutoBean obj
    WHERE obj.codigo IS NOT NULL
  ]]></ejb-ql>
</query>
```

- Beans CMP são **classes abstratas**
- Atributos de dados não são declarados na classe do bean mas no DD como parte do **esquema de dados** do bean
- Métodos **get/set** têm que combinar com atributos declarados e devem ser **abstract** no bean
- Nenhum **finder** é implementado no bean. Todos são declarados no DD com parâmetros e lógica descritos em **EJB-QL** que usa os componentes do esquema de dados
- **findByPrimaryKey()** só é declarado na interface **Home**
- Métodos **ejbHome()** não são declarados no DD, mas usam métodos privados **ejbSelect()**, **abstratos** no bean, que não aparecem no **Home** mas são especificados em **EJB-QL** para realizar suas tarefas.

Exercícios Teóricos

- *1. Escreva queries CMP para as seguintes situações (invente referências e nomes significativos)*
 - *a) Localizar todas as contas com saldo maior que zero*
 - *b) Localizar todas as contas cujos identificadores começam com '0000-'*
 - *c) Localizar as contas cujos saldos sejam menores que 100 ou maiores que 10000*
 - *d) Localizar produtos que custam menos que 1000 e que tenham pelo menos duas unidades em estoque*
- *Queries de relacionamento (extras)*
 - *e) Localize todos os clientes que têm pelo menos um pedido na sua coleção de pedidos (não está vazio)*
 - *f) Localize todos os pedidos que façam parte da coleção de pedidos do cliente cujo userid é 'genghis'*

- 2. *Implemente o exemplo BMP mostrado (Account) em CMP*
 - *Faça o deployment*
 - *Execute usando o mesmo cliente*
- 3. *(opcional) Adapte o exercício BMP (Produto) para CMP*
 - *Faça o deployment*
 - *Execute usando o mesmo cliente*
 - *Implemente queries sofisticados para pesquisar produtos (por preço, por nome, por quantidade, etc.)*

- [1] *Ed Roman et al. Mastering Enterprise JavaBeans, 2nd. Edition, Chapter 11: CMP & BMP Relationships. John Wiley & Sons, 2002.*
- [2] *Linda de Michiel et al. Enterprise JavaBeans 2.1 Specification. Sun Microsystems, 2003.*
- [3] *Richard Monson-Haefel. Enterprise JavaBeans, 3rd. Edition. O'Reilly and Associates, 2001*
- [4] *J2EE Tutorial. Sun J2EE Tutorial, Sun Microsystems, 2002*
- [5] *Emanuel Proulx. EJB Inheritance, Parts 1-4. O'Reilly Network, 2002-2003. www.oreillynet.com/pub/a/onjava/2002/09/04/ejbinherit.html (sobre herança em EJB - não abordado nesta versão - veja exemplo do cap. 15 para exemplo de herança entre Entity Beans)*

helder@argonavis.com.br

argonavis.com.br

*J500 - Aplicações Distribuídas com J2EE e JBoss
Revisão 1.5 (Junho de 2003)*

© 1999-2003, Helder da Rocha