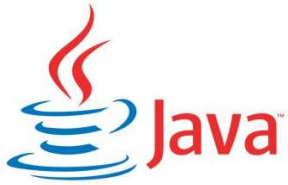
Java Enterprise Edition módulo IV

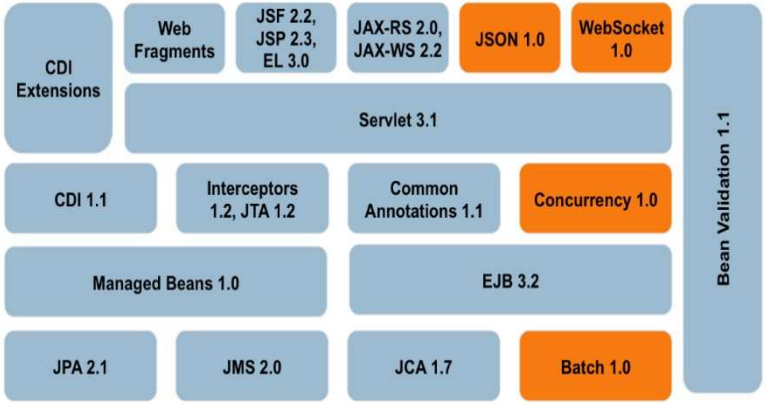
PROVAS DE TI
TUDO PARA VOCÊ PASSAR

Leonardo Marcelino
http://www.itnerante.com.br/profile/LeonardoMarcelino

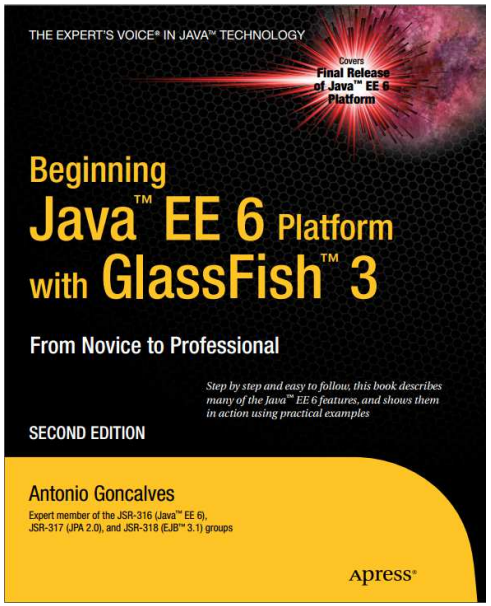
Java Enterprise Edition

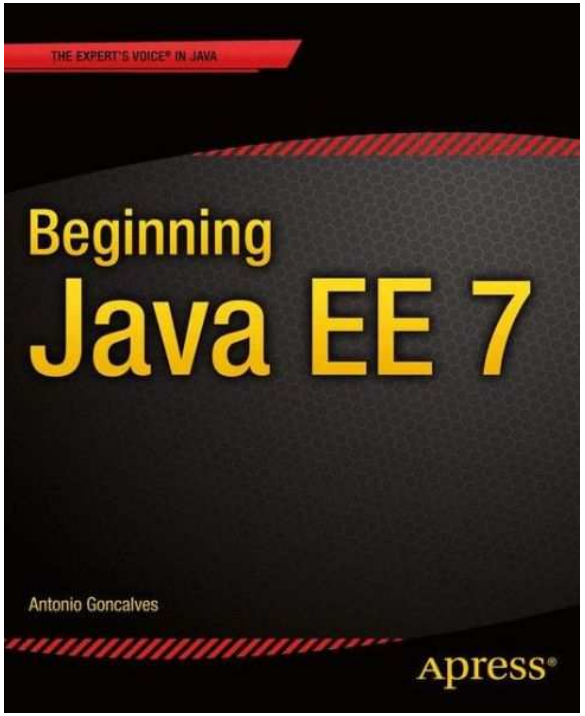


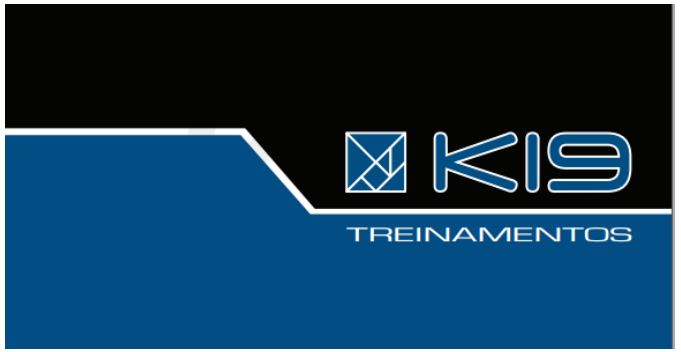
- Módulo 4
  - ✓Enterprise Application Technologies
    - Enterprise JavaBeans
    - Interceptors
    - Java Persistence
  - ✓Web Services Technologies
    - Java API for XML-Based Web Services
    - Java API for RESTful Web Services
  - ✓Java EE-related Specs in Java SE
    - Java Architecture for XML Binding
    - Java Database Connectivity



REFERÊNCIAS







Entreprise JavaBeans (EJB)

Entreprise JavaBeans (EJB)

- ✓ encapsulam a lógica de negócio
- ✓ rodam no servidor
  - Container EJB
    - ⇒ fornece serviços em nível de sistema
      - gerenciamento de transações e controle de concorrência
      - injeção de dependência
      - segurança (autenticação e autorização)
      - pooling de objetos
      - persistência

- ✓ versões x tipos
  - Entity Beans, Session Beans e Message-driven Beans
    - EJB 2.1 JSR 153 J2EE
    - EJB 3.0 JSR 220 JEE5 (inclui JPA)
  - Session Beans e Message-driven Beans
    - EJB 3.1 JSR 318 JEE6 (inclui interceptors)
    - EJB 3.2 JSR 345 JEE7

[01] AOCP - 2012 - BRDE

Sobre definições e características de Enterprise JavaBeans, analise as assertivas e assinale a alternativa que aponta as corretas.

I. A arquitetura Enterprise JavaBeans é uma arquitetura de componentes para o desenvolvimento e a implantação de aplicativos de negócios distribuídos baseados em componentes.

II. Aplicativos escritos utilizando a arquitetura Enterprise JavaBeans são escalonáveis, transacionais e seguros com multiusuários.

[01] AOCP - 2012 - BRDE

III. Aplicativos escritos utilizando a arquitetura Enterprise JavaBeans uma vez escritos e então implantados em qualquer plataforma de servidor, que suporta a especificação Enterpise JavaBeans.

IV. A arquitetura JavaBeans encontra-se presente em outras linguagens de programação além da linguagem de programação java, esta arquitetura encontra- se em Object Pascal, Objective-C, Python, SuperCollider, Ruby, Smalltalk, entre outras.

a) Apenas I e II.

b) Apenas I e III.

c) Apenas I, II e III.

d) Apenas II, III e IV.

e) I, II, III e IV.

[02] FCC - 2013 - TRT - 15 [adaptada]

O EJB é um modelo de componentes, especificado pela plataforma Java EE, elaborado para resolver problemas e desafios complexos de softwares corporativos. São componentes que atuam na camada servidor, classificados como componentes de negócio. Podem ser utilizados em diferentes situações como desenvolvimento distribuído, integração/conectividade com legado, processamento assíncrono baseado Fila / Mensagens, controle transacional e outros. Este componente é responsável pelas regras de negócio, ou seja, a persistência e o controle transacional.

[03] NUCEPE - 2015 - SEFAZ-PI

O Enterprise JavaBeans (EJB) é um componente da plataforma Java EE que executa em um container de um servidor de aplicação. Quais os seus 3 (três) tipos fundamentais de beans?

a) Entity Beans, Session Beans e Message Driven Beans.

b) Entity Beans, Session Beans e Work Beans;

c) Session Beans, Message Driven Beans e Work Beans.

d) Session Beans, Progress Beans e Work Beans.

e) Entity Beans, Progress Beans e Work Beans.

Tipos de EJB: Entity Beans

- **J2EE [**EJB 2.1 JSR 153**]**
- **JEE 5 [**JPA - EJB 3.0 JSR 220**]**

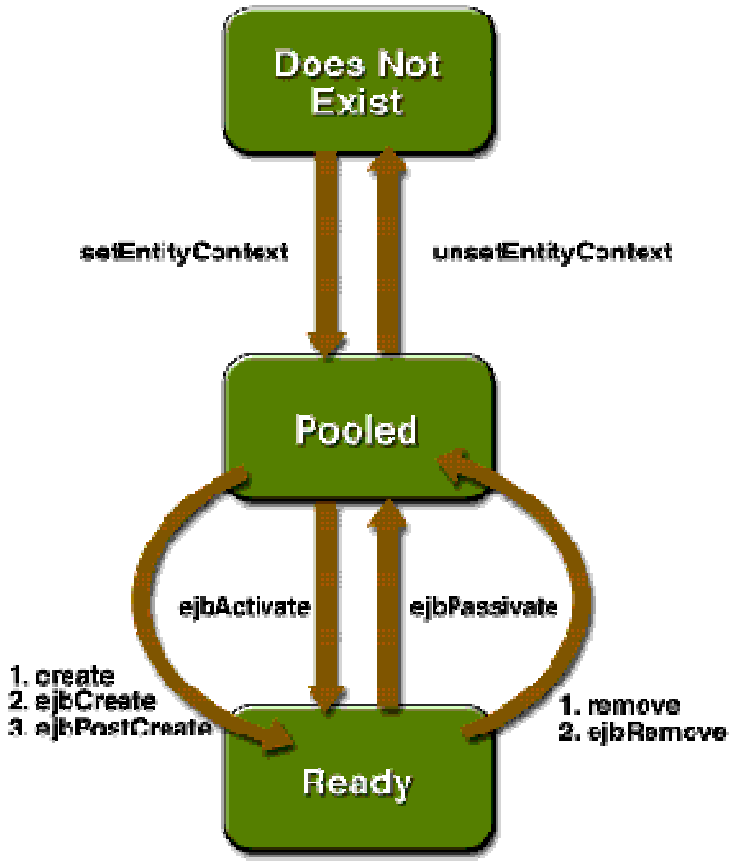
✓ persistência de objetos de negócio em bancos de dados
  - armazena o estado do objeto

✓ classe = tabela
  - possui atributo chave primária
  - participa de relacionamentos

✓ instância = linha

✓ registrado no ejb-jar.xml

- **Ciclo de vida**
  - não-existe
  - no-pool
    - ⇒ instâncias genéricas e idênticas
    - ⇒ sem ID de objeto associada
  - pronto
    - ⇒ pode invocar métodos de negócio



Tipos de EJB: Entity Beans

- **artefatos**
  - ✓ interface Home
    - extende EJBHome[remoto] ou EJBLocalHome[local]
    - define os métodos create() e finders
      - ⇒ findByPrimaryKey, findByName, findAll, etc
  - ✓ interface do componente
    - extende *javax.ejb.EJBObject* ou *javax.ejb.EJBLocalObject*
    - declara métodos que acessar atributos do componente
      - ⇒ getters e setters, etc

```
public interface EmployeeHome extends EJBHome {
    public Employee create(Integer empNo, String empName, Float salary)
            throws CreateException, RemoteException;
    public Employee findByPrimaryKey(EmployeePK pk) throws
            FinderException, RemoteException;

    public Collection findByName(String empName) throws FinderException,
            RemoteException;
    public Collection findAll() throws FinderException, RemoteException;
}
```

```
public interface Employee extends EJBObject {
    // getter remote methods
    public Integer getEmpNo() throws RemoteException;
    public String getEmpName() throws RemoteException;
    public Float getSalary() throws RemoteException;

    // setter remote methods
    public void setEmpNo(Integer empNo) throws RemoteException;
    public void setEmpName(String empName) throws RemoteException;
    public void setSalary(Float salary) throws RemoteException;
}
```

Tipos de EJB: Entity Beans

- **artefatos**
  - ✓ classe PrimaryKey
    - PK = atributo simples : declarado no ejb-jar.xml
    - PK = complexo : definir uma classe PK
      - ⇒ <*name*>PK
      - ⇒ implementa serializable + atributos public
      - ⇒ construtor para criar a instância PK
      - ⇒ implementar os métodos hashCode() e equals()

```
public class EmployeePK implements java.io.Serializable {
        public Integer empNo;
        public Integer empSSN;

        public EmployeePK() {
                this.empNo = null;
                this. empSSN = null;
        }

        public EmployeePK(Integer empNo, Integer empSSN) {
                this.empNo = empNo;
                this. empSSN = empSSN;
        }
}
```

```
<enterprise-beans>
        <entity>
                <prim-key-class>java.lang.Integer</prim-key-class>
                <cmp-version>2.x</cmp-version>
                <abstract-schema-name>Employee</abstract-schema-name>
                <cmp-field>
                        <field-name>empNo</field-name>
                </cmp-field>
                <primkey-field>empNo</primkey-field>
        </entity>
</enterprise-beans>
```

Tipos de EJB: Entity Beans

- **artefatos**
  - ✓ classe abstrata Entidade (entity bean)
    - dois tipos:
      - Container-Managed Persistence (CMP)
      - Bean-Managed Persistence (BMP)
        - ⇒ implementado pelo desenvolvedor
    - implementa a interface *javax.ejb.EntityBean*
      - ⇒ método ejbCreate >> create() da interface home
      - ⇒ métodos do ciclo de vida do EntityBean
        - ⇢ CMP > só declaração
        - ⇢ BMP > implementação na classe
    - métodos declarados na interface do componente
      - ⇒ getters e setters, etc
        - ⇢ CMP > só declaração
        - ⇢ BMP > implementação na classe

Tipos de EJB: Entity Beans

```
public abstract class EmployeeBean implements EntityBean {

    public EmployeePK ejbCreate(Integer empNo, Integer empSSN, String empName,
            Float salary) throws CreateException {  }

    public void setEntityContext(EntityContext ctx) { }

    public void unsetEntityContext() { }

    public void ejbPostCreate(Integer empNo) throws CreateException { }

    public void ejbStore() throws EJBException, RemoteException { }

    public void ejbLoad() throws EJBException, RemoteException{ }

    public void ejbRemove() throws EJBException, RemoteException{ }

    public void ejbActivate() throws EJBException, RemoteException{ }

    public void ejbPassivate() throws EJBException, RemoteException{ }

    public abstract Integer getEmpNo();

    public abstract void setEmpNo(Integer empNo);
}
```

```
public EmployeePK ejbCreate(Integer empNo, Integer empSSN, String empName, Float salary)
        throws CreateException {
    setEmpNo(empNo);
    setEmpSSN(empSSN);
    setEmpName(empName);
    setSalary(salary);
    return new EmployeePK(empNo);
}
```

```
private Connection conn = null;
private PreparedStatement ps = null;
private EmployeePK pk;
private static final String dsName = "jdbc/OracleDS";

private static final String insertStatement = "INSERT INTO EMP (EMPNO,
                ENAME, SAL) VALUES (?, ?, ?)";

private DataSource getDataSource(String dsName) throws NamingException {
        DataSource ds = null;
        Context ic = new InitialContext();
        ds = (DataSource) ic.lookup(dsName);
        return ds;
}

private Connection getConnection(String dsName) throws SQLException,
                NamingException {
        DataSource ds = getDataSource(dsName);
        return ds.getConnection();
}
```

```
public EmployeePK ejbCreate(Integer empNo, Integer empSSN, String empName,
                        Float salary) throws CreateException {
        try {
                pk = new EmployeePK(empNo, empSSN);
                conn = getConnection(dsName);
                ps = conn.prepareStatement(insertStatement);
                ps.setInt(1, empNo.intValue());
                ps.setString(2, empName);
                ps.setFloat(3, salary.floatValue());
                ps.executeUpdate();
                return pk;
        } catch (SQLException e) {
                System.out.println("Caught an exception 1 " + e.getMessage());
                throw new CreateException(e.getMessage());
        } catch (NamingException e) {
                System.out.println("Caught an exception 1 " + e.getMessage());
                throw new EJBException(e.getMessage());
        } finally {
                try {
                        ps.close();
                        conn.close();
                } catch (SQLException e) {
                        throw new EJBException(e.getMessage());
                }
        }
}
```

```
<enterprise-beans>
    <entity>
        <description>no description</description>
        <display-name>EmployeeBean</display-name>
        <ejb-name>EmployeeBean</ejb-name>
        <home>EmployeeHome</home>
        <remote>Employee</remote>
        <ejb-class>EmployeeBean</ejb-class>
        <persistence-type>Container</persistence-type>
        <cmp-version>2.x</cmp-version>
        <abstract-schema-name>EmployeeBean</abstract-schema-name>
        <prim-key-class>EmployeePK</prim-key-class>
        <reentrant>False</reentrant>

        <cmp-field><field-name>empNo</field-name></cmp-field>
        <cmp-field><field-name>empSSN</field-name></cmp-field>
        <cmp-field><field-name>empName</field-name>     </cmp-field>
        <cmp-field><field-name>salary</field-name></cmp-field>

        <query>
            <description></description>
            <query-method>
                <method-name>findAll</method-name>
                <method-params />
            </query-method>
            <ejb-ql>Select OBJECT(e) From EmployeeBean e</ejb-ql>
        </query>
        <query>
            <description></description>
            <query-method>
                <method-name>findByName</method-name>
                <method-params>
                    <method-param>java.lang.String</method-param>
                </method-params>
            </query-method>
            <ejb-ql>Select OBJECT(e) From EmployeeBean e where e.empName = ?1
            </ejb-ql>
        </query>
    </entity>
</enterprise-beans>
```

```
<enterprise-beans>
    <entity>
        <description>no description</description>
        <display-name>EmployeeBean</display-name>
        <ejb-name>EmployeeBean</ejb-name>
        <home>EmployeeHome</home>
        <remote>Employee</remote>
        <ejb-class>EmployeeBean</ejb-class>
        <persistence-type>Bean</persistence-type>
        <prim-key-class>EmployeePK</prim-key-class>
        <reentrant>False</reentrant>
        <resource-ref>
            <res-ref-name>jdbc/OracleDS</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Application</res-auth>
        </resource-ref>
    </entity>
</enterprise-beans>
```

[04]

CESPE - 2011 - TRE-ES

Nos *beans* de entidade cuja persistência é gerenciada por contêiner, o desenvolvedor tem a responsabilidade de escrever todo o código JDBC para a interação com o banco de dados.

ESAF - 2008 - CGU [adaptada]

Os *Entity Beans* foram substituídos pela "*Java Persistence API*" na versão EJB 3.0, porém, os *Entity Beans* de versões 2.x podem continuar utilizando o *"Container-Managed Persistence"* (CMP) por questões de compatibilidade.

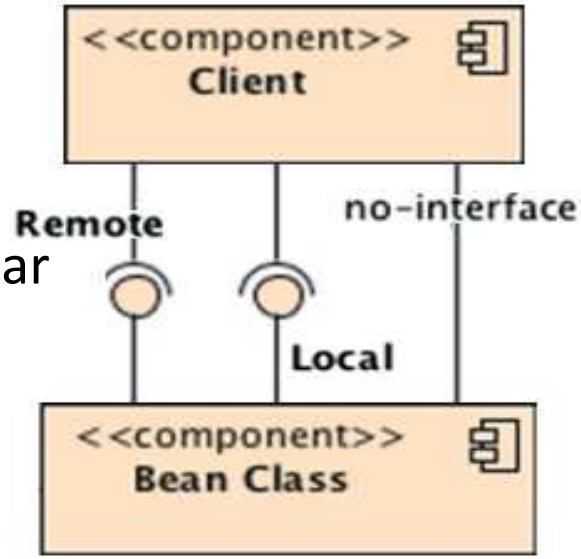
Tipos de EJB: Session Bean

- **J2EE [**EJB 2.1(153)**]**
- **JEE [**EJB 3.0(220), EJB 3.1(318) e EJB 3.2(345)**]**

- ✓ encapsula a lógica de negócio
- ✓ não é persistente
- ✓ roda no servidor
  - Container EJB
    - ⇒ fornece serviços em nível de sistema
      - gerenciamento de transações (CMT e BMT)
      - segurança (autenticação e autorização)
- ✓ tipos de Session bean
  - stateless, stateful e singleton (JEE6)
- ✓ composição
  - Business interfaces
  - Classe bean
  - Descritor de implantação (ejb-jar.xml)

Tipos de EJB: Session Bean

- **artefatos**
  - ✓ Business interface (bean interface)
    - declara métodos que o cliente pode invocar
    - anotada com
      - ⇒ @Remote (RMI)
        - serializable e parâmetros **por valor**
      - ⇒ @Local
        - parâmetros **por referência**
        - **EJB 3.1**: não precisa interface local
      - ⇒ no-interface : container trata como local



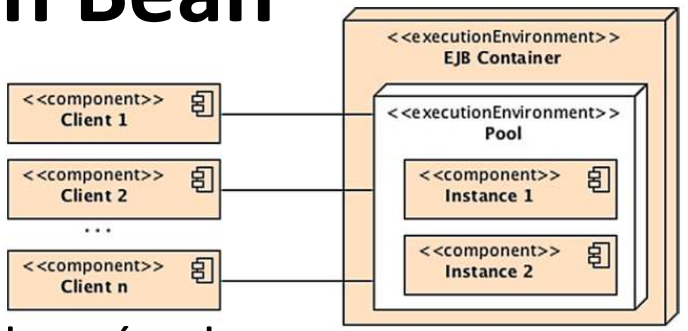
  - ✓ Classe Bean
    - implementa os métodos do negócio declarados nas interfaces
    - pode implementar métodos do ciclo de vida (callback)
    - anotada com *@Stateless*, *@Stateful* ou *@Singleton*
      - ⇒ J2EE implementa javax.ejb.SessionBean + ejb-jar.xml

Tipos de EJB: Session Bean

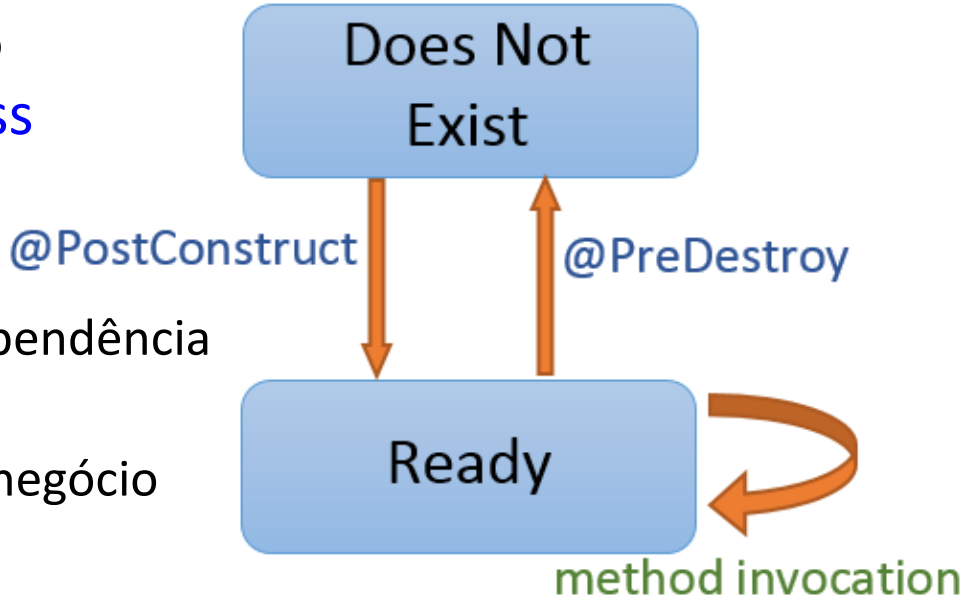
```
@Local
public interface ItemLocal {
    List<Book> findBooks();
    List<CD> findCDs();
}

@Remote
public interface ItemRemote {
    List<Book> findBooks();
    List<CD> findCDs();
    Book createBook(Book book);
    CD createCD(CD cd);
}

@WebService
public interface ItemSOAP {
    List<Book> findBooks();
    List<CD> findCDs();
}

@Path("/items")
public interface ItemRest {
    List<Book> findBooks();
}

@Stateless
public class ItemEJB implements ItemLocal, ItemRemote, ItemSOAP,
            ItemRest, Serializable {
    ...
}
```

```
public interface ItemLocal {
    List<Book> findBooks();
    List<CD> findCDs();
}

public interface ItemRemote {
    List<Book> findBooks();
    List<CD> findCDs();
    Book createBook(Book book);
    CD createCD(CD cd);
}

@Stateless
@Remote(ItemRemote.class)
@Local(ItemLocal.class)
@LocalBean
public class ItemEJB implements ItemLocal, ItemRemote, Serializable
{
    ...
}
```

Tipos de EJB: Stateless Session Bean



- ▷ **J2EE (**EJB 2.1 JSR 153)
- ✓ não mantem estado conversacional
  - não retêm informações entre requisições
  - tarefa concluída em uma única chamada de método
  - pode implementar web service
  - maior eficiência e escalabilidade
    - ⇒ pooling e compartilhamento
- ✓ Classe anotada com @Stateless
- ▷ **Ciclo de vida**
  - não-existe
    - ⇒instanciação, injeção de dependência
  - pronto
    - ⇒ pode invocar métodos de negócio
- ✓ Pooling (não é estado)



Tipos de EJB: Statefull Session Bean

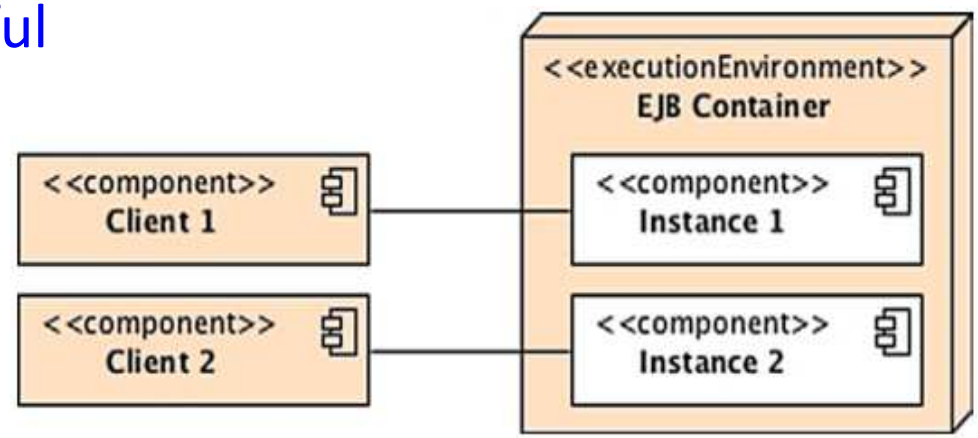
- **J2EE (**EJB 2.1 JSR 153)

✓ mantem estado conversacional

- retêm informações entre requisições
- variáveis de instância armazenam valores entre chamadas cliente
- tarefas feitas em várias etapas (interações)
- uma instância de stateful session bean por cliente

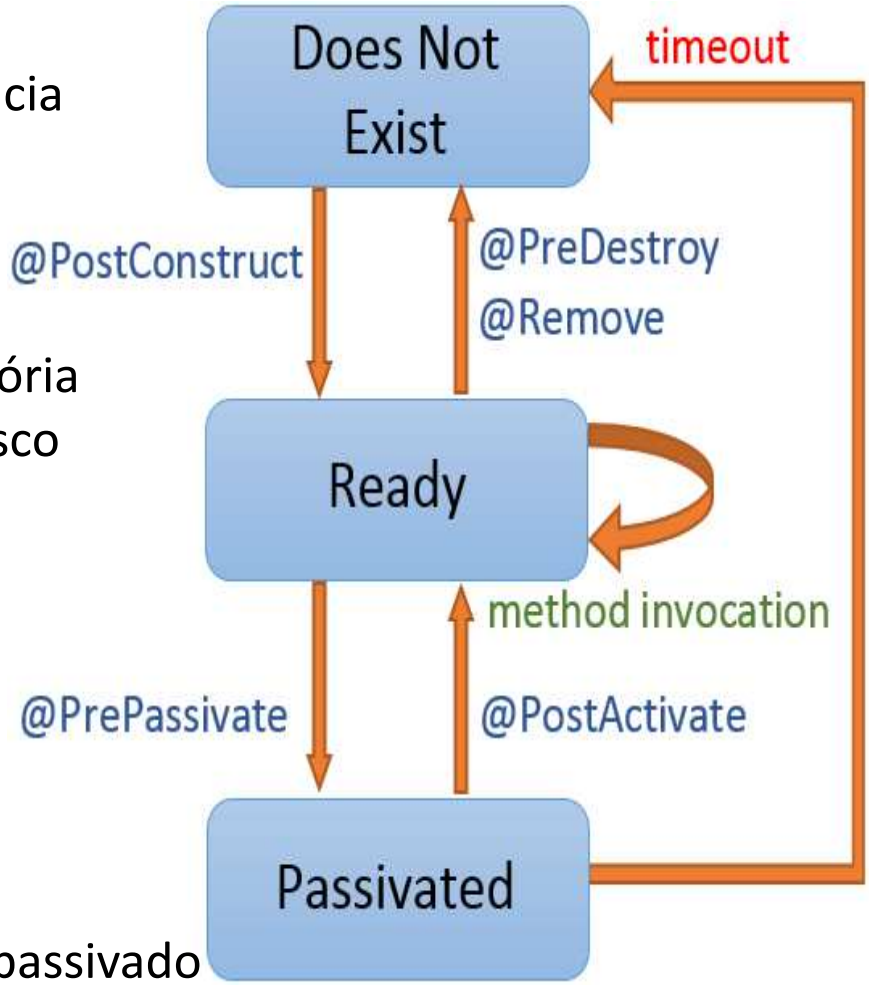
  ⇒ não há pooling e nem compartilhamento

✓ Classe anotada com @Stateful



Tipos de EJB: Statefull Session Bean

- **Ciclo de vida**
  - não-existe
    - ⇒instanciação, injeção de dependência
  - pronto
    - ⇒atende as requisições do cliente
  - passivado
    - ⇒evitar excessivo consumo de memória
    - ⇒armazena o estado do bean em disco
    - ⇒**passivação**
      - container invoca @PrePassivate
      - pode ser desabilitada (EJB 3.2)
    - ⇒**ativação**
      - nova requisição do cliente
      - container invoca @PostActivate
    - ⇒**timeout**
      - cliente não invoca session bean passivado



Tipos de EJB: Stateful

```
@Stateful
@StatefulTimeout(value = 10, unit = TimeUnit.MINUTES)
public class ShoppingCartEJB {

    @PersistenceContext
    private EntityManager entityManager;
    private List<Item> cartItems;

    public void addItem(Item item) {
        if (!cartItems.contains(item))
            cartItems.add(item);
    }

    public void removeItem(Item item) {
        if (cartItems.contains(item))
            cartItems.remove(item);
    }

    public Integer getNumberOfItems() {
        if (cartItems == null || cartItems.isEmpty())
            return 0;
        return cartItems.size();
    }

    public Float getTotal() {
        if (cartItems == null || cartItems.isEmpty())
            return 0f;
        Float total = 0f;
        for (Item cartItem : cartItems) {
            total += (cartItem.getPrice());
        }
        return total;
    }

    @Remove
    public void checkout() {
        for (Item item : cartItems) {
            entityManager.persist(item);
        }
    }

    @PostConstruct
    @PostActivate
    public void initialize() {
        cartItems = new ArrayList<>();
    }

    @PreDestroy
    @PrePassivate
    public void releaseResources() {
        cartItems.clear();
    }
}
```

Tipos de EJB: Stateless x Stateful

```java
@Remote
public interface iCalculatorStateless {
    public long add(long valueA, long valueB);
    public long subtract(long valueA, long valueB);
    public long multiply(long valueA, long valueB);
    public double divide(long valueA, long valueB);
}
```

```java
@Stateless
public class CalculatorStateless implements iCalculatorStateless {

    public long add(long valueA, long valueB) {
        return (valueA + valueB);

    public long subtract(long valueA, long valueB) {
        return (valueA - valueB);   }

    public long multiply(long valueA, long valueB) {
        return (valueA * valueB);   }

    public double divide(long valueA, long valueB) {
        return ((double) valueA / valueB);   }
}
```

```java
@Remote
public interface iCalculatorStateful {
    public void add(long value);
    public void subtract(long value);
    public void multiply(long value);
    public void divide(long value);

    public void clearIt();
    public double getValue();
}
```

```java
@Stateful
public class CalculatorStateful implements iCalculatorStateful {
    private double value = 0;

    public void add(long value) {
        this.value += value;    }

    public void subtract(long value) {
        this.value -= value;    }

    public void multiply(long value) {
        this.value *= value;    }

    public void divide(long value) {
        this.value /= value;    }

    @Remove
    public void clearIt() {
        value = 0;      }

    public double getValue() {
        return this.value; }
}
```

[05] FCC - 2015 - TRT-15

Em uma aplicação web que utiliza Enterprise JavaBeans (EJB) para implementar um carrinho de compras, utilizou-se um tipo de bean que mantém o estado durante uma sessão com o cliente. Nesta aplicação, para indicar ao servidor que a classe é um bean com estado de sessão deve-se utilizar, antes da declaração da classe, a anotação

a) @Session state="true"

b) @Stateful

c) @SessionState= "true"

d) @Stateless

e) @SessionRemote

[06] FCC - 2014 - TCE-RS

Em uma aplicação Java EE há:

- uma classe EJB chamada ExemploSessionBean.java que não permite manter um estado de conversação com o cliente,
- uma interface chamada ExemploSessionBeanRemote.java e
- uma classe cliente desktop chamada Main.java

todas criadas em seus respectivos projetos e em condições de execução ideais. Os fragmentos de código-fonte destas classes são apresentados a seguir, omitindo-se as declarações de pacotes e importação de classes:

ExemploSessionBean.java

```
I
.....
public class ExemploSessionBean implements ExemploSessionBeanRemote{
   II
   .....
   public String getMessageRemote() {
      return "Remote EJB";
   }
}
```

ExemploSessionBeanRemote.java

```
III
.....
public interface ExemploSessionBeanRemote {
   String getMessageRemote();
}
```

Main.java

```
public class Main {
   IV
   .....
   private static ExemploSessionBeanRemote ex;
   public static void main(String[] args) {
      System.err.print(ex.getMessageRemote());
   }
}
```

As lacunas I, II, III e IV são preenchidas correta e, respectivamente, pelas anotações

a) @Statefull, @Override, @Remote e @EJB.

b) @Stateless, @Override, @Remote e @EJB.

c) @EJB, @Stateless, @Remote e @EJBInjection.

d) @Stateless, @EntityManager, @RemoteInterface e @EJBInjection.

e) @Statefull, @EJBMethod, @RemoteInterface e @EJBInjection.

[07] CESGRANRIO - 2013 - BNDES

Cada tipo de enterprise bean passa por diferentes fases durante seu ciclo de vida. Um desses tipos possui um estado denominado Passivo. Quando um bean entra nesse estado, o container EJB o desloca da memória principal para a memória secundária.

Qual tipo de bean se comporta dessa maneira?

a) Stateless Session Bean

b) Stateful Session Bean

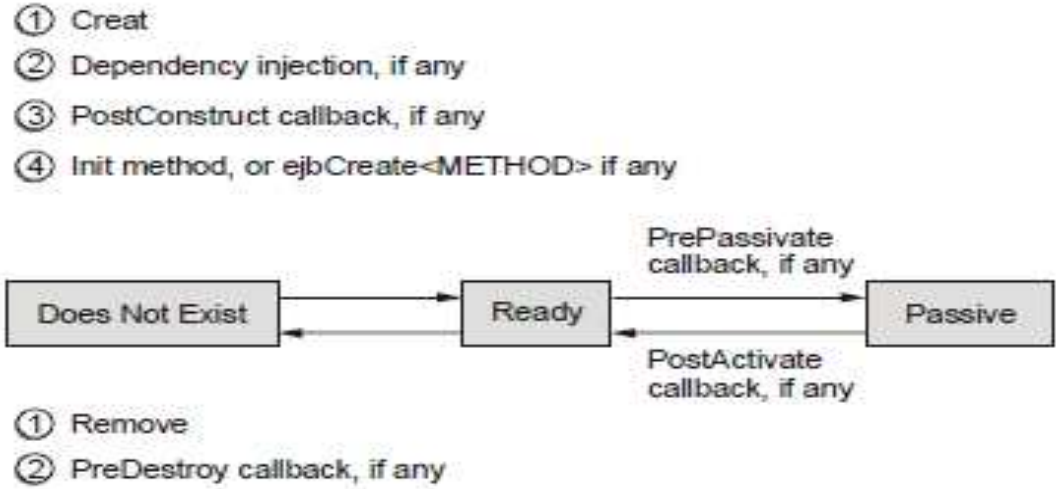
c) Web Service Bean

d) Singleton Session Bean

e) Message-Driven Bean

[08] FCC - 2012 - TRT-AM

Analise a figura:



Foi representado o ciclo de vida de um

a) Stateful Session Bean.
b) Stateless Session Bean.
c) Singleton Session Bean.
d) Message-Driven Bean.
e) Embedded Session Bean.

[09] FCC - 2011 - TRT-19

Tipo de *session bean EJB* 3.1 cujas instâncias não têm estado conversacional, isto é, todas as instâncias são equivalentes quando não estão envolvidas em atender um método invocado pelo cliente. Trata-se de

a) Stateful.

b) Stateless.

c) Singleton.

d) Message driven.

e) Entity.

[10]

ESAF - 2008 - CGU [adaptada]

e) os *Stateless Session Beans* são objetos distribuídos que não possuem estado, permitindo acesso concorrente aos mesmos. Assim, o conteúdo das variáveis de instância é preservado entre as chamadas de métodos.

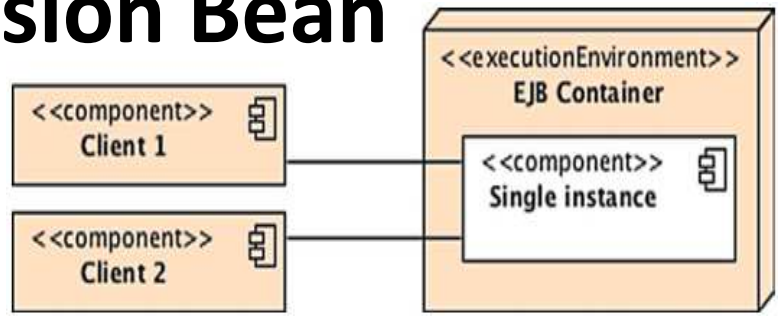
FCC - 2011 - TRT-24 [adaptada]

a) No Session Bean, toda vez que um método é invocado, o estado de suas variáveis se mantém apenas durante a invocação desse método.

NUCEPE - 2015 - SEFAZ-PI [adaptada]

e) o Stateful Session Bean não mantém estado, permitindo que diversas chamadas a métodos sejam feitas de forma que uma chamada dependa da outra.

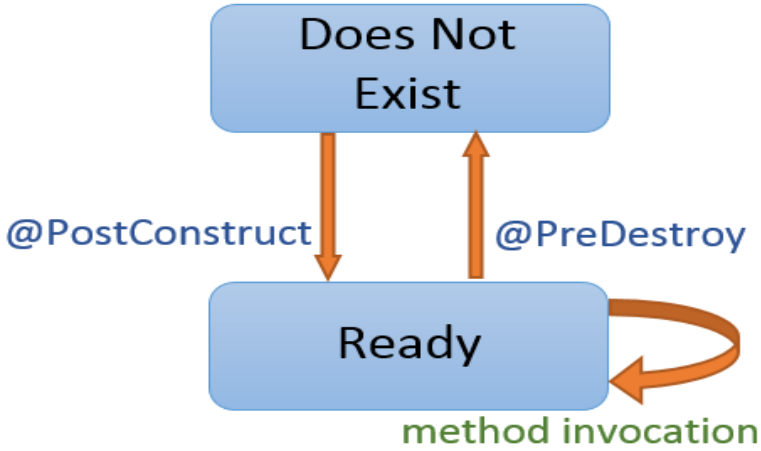
Tipos de EJB: Singleton Session Bean



- ▷ **JEE 6** (EJB 3.1 - JSR 318)
- ✓ não mantem estado conversacional
- ✓ existe uma única instância do EJB
  - implementação do Singleton (GoF)
  - compartilha seu estado com todos os clientes
    - ⇒ ponto único de acesso global
    - ⇒ concorrência entre requisições clientes
- ✓ não é cluster-aware
  - cada container tem seu Singleton
- ✓ usos (recursos compartilhados)
  - ⇒ cashing, spool de impressão, etc.
- ✓ Classe anotada com @Singleton

Tipos de EJB: Singleton Session Bean

- **Concorrência**
  - ✓ Tipos (*@ConcurrencyManagement*)
    - Container-Managed Concurrency (default)
      - ⇒ @Lock (classe ou método)
        - exclusivo: LockType.WRITE (default)
        - compartilhado: LockType.READ
    - Bean-Managed Concurrency
      - ⇒ synchronized (WRITE), volatile (READ), java.util.concurrent
  - ✓ @AccessTimeout
- **Ciclo de vida**
  - não-existe
    - ⇒1ª requisição cliente [new() ou DI]
    - ⇒deploy (@Startup)
  - pronto
    - ⇒ uma instância atende vários clientes



Tipos de EJB: Singleton Session Bean

```
@Singleton
@Startup
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
@Lock(LockType.READ)
@AccessTimeout(value = 20, unit = TimeUnit.SECONDS)
public class CacheEJB {
	private ConcurrentHashMap<Long, Object> cache;

	@Lock(LockType.WRITE)
	public void addToCache(Long id, Object object) {
		if (!cache.containsKey(id))
			cache.put(id, object);
	}

	public void removeFromCache(Long id) {
		if (cache.containsKey(id))
			cache.remove(id);
	}

	public Object getFromCache(Long id) {
		if (cache.containsKey(id))
			return cache.get(id);
		else
			return null;
	}

	@PostConstruct
	public void initCache() {
		this.cache = new ConcurrentHashMap<Long, Object>();
	}

	@PreDestroy
	void cleanup() {
		this.cache = null;
	}
}
```

Session Bean Timer Service

- ▷ **Agendador (Scheduler)**
  - ✓ Sintaxe baseada em calendário
    - J2EE (EJB 2.1 JSR 153)
    - JEE 5 (EJB 3.0 JSR 220) 
    - JEE 6 (EJB 3.1 JSR 318) 
  - ✓ são persistentes (default)
    - ⇒ container mantém registro e invoca o método

     não suportado por **Stateful** Session Bean

  - ❖ Declarativa @Schedule
  - ❖ Programática (API TimeService)
    - ⇒ obter instância de TimerService
    - ⇒ agendar com o objeto ScheduleExpression
    - ⇒ criar o timer - createCalendarTimer()
    - ⇒ método @Timeout

Session Bean Timer Service

```java
@Stateless
public class StatisticsEJB {
    @Schedule(dayOfMonth = "1", hour = "5", minute = "30")
    public void statisticsItemsSold() {
        // ...
    }

    @Schedules({ @Schedule(hour = "2"),
                 @Schedule(hour = "14", dayOfWeek = "Wed") })
    public void generateReport() {
        // ...
    }

    @Schedule(minute = "*/10", hour = "*", persistent = false)
    public void refreshCache() {
        // ...
    }
}
```

```java
@Stateless
public class CustomerEJB {

    @Resource
    TimerService timerService;

    public void createCustomer(Customer customer) {
        // ... customer creation
        ScheduleExpression birthDay = new ScheduleExpression().dayOfMonth(
                customer.getBirthDay()).month(customer.getBirthMonth());

        timerService.createCalendarTimer(birthDay, new TimerConfig(customer,
    true));
    }

    @Timeout
    public void sendBirthdayEmail(Timer timer) {
        Customer = (Customer) timer.getInfo();
        // ...
    }
}
```

Session Bean Assíncrono

- ▷ **JEE 6 (**EJB 3.1 JSR 318)
  - ⇒ antes processamento assíncrono via JMS + MDB
- ✓ processamento assíncrono
  - ⇒ container retorna o controle para o cliente
  - ⇒ método roda em outra thread
  - ⇒ usado em processamento longos
- ✓ método assíncrono anotado com @Asynchronous
  - retorna void ou Future<V>
    - ⇒ get(), cancel(Boolean), isDone(), isCancelled()
    - ⇒ javax.ejb.**AsyncResult<V>**
  - adornar a classe com @Asynchronous
    - ⇒ todos os métodos assíncronos

```
@Asynchronous
public Future<Integer> sendOrderToWorkflow(Order order)  {
    if (sessionContext.wasCancelCalled()) {
        return new AsyncResult<Integer>(0);
    } else {
        // process the payment
        return new AsyncResult<Integer>(result);
    }
}
```

```
Future<Integer> status = processPayment.sendOrderToWorkflow(order);
statusOrder = status.get();

status.cancel(true);
```

[11] NC-UFPR - 2015 - COPEL

Sobre Session Beans, conforme a especificação EJB (Enterprise JavaBeans) 3.1, identifique as afirmativas a seguir como verdadeiras (V) ou falsas (F):

( ) Não é possível utilizar Multithreading em EJBs do tipo Singleton.

( ) Stateless Session Beans não armazena nenhuma informação sobre o estado transacional (conversacional), ou seja, nenhuma informação é automaticamente mantida entre as diferentes requisições.

( ) Recomenda-se utilizar um Stateful Session Bean ao construir um carrinho de compras de um e-commerce, embora seja possível usar um Stateless Session Bean, tendo um pouco mais de trabalho.

( ) Existem apenas três tipos de Session Beans: Stateful, Stateless e Singleton.

Assinale a alternativa que apresenta a sequência correta, de cima para baixo.

a) F – V – F – F.

b) F – F – V – V.

c) V – V – V – F.

d) F – V – V – V.

e) V – F – F – V.

[12] CESPE - 2010 - TCU

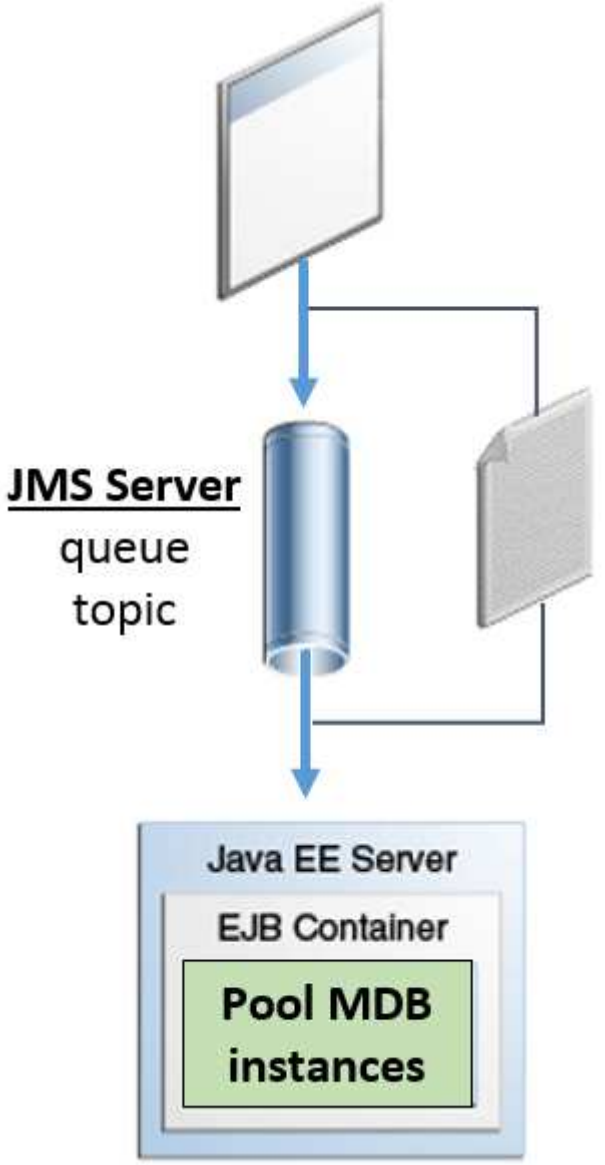
Uma equipe de desenvolvimento de *software* recebeu a incumbência de desenvolver um sistema com as características apresentadas a seguir.

- O sistema deverá ser embasado na plataforma JEE (Java *enterprise edition*) v.6, envolvendo *servlets*, JSP (Java *server pages*), Ajax, JSF (Java *server faces*) 2.0, Hibernate 3.5, SOA e *web services*.

A tecnologia EJB (*enterprise* Java *beans*) apresenta, na sua versão 3.1, melhorias que propiciam facilidades para o uso de *beans singleton* e que permitem o uso de *beans* de uma classe, sem necessidade de desenvolvimento de sua interface correspondente, e a invocação assíncrona de *beans* de sessão.

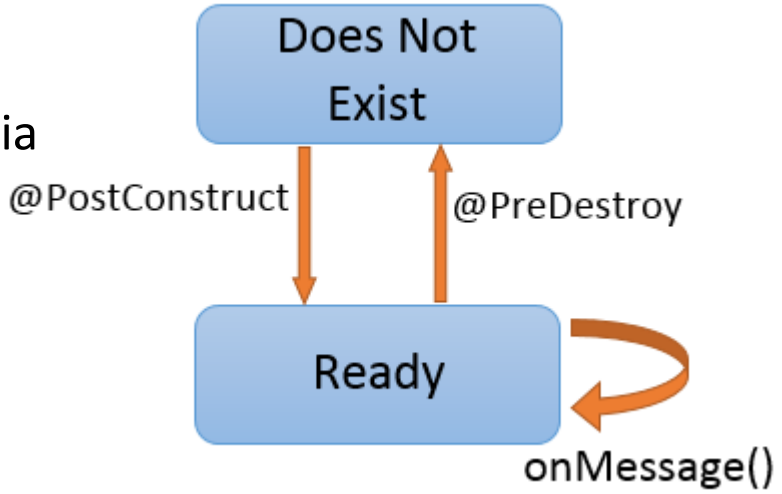
Tipos de EJB: Message-driven Bean

- ▷ **J2EE (**EJB 2.1 JSR 153)
- ✓ consumidor de mensagens assíncronas
  - ⇒ integração com sistemas externos
    - message-oriented middleware (MOM)
  - ⇒ Java Message Service (JMS)
  - ⇒ reduz acoplamento entre aplicações
- ✓ delegam lógica de negócio
  - ⇒ Stateless Session Bean
- ✓ semelhante ao Stateles
  - ‣ não mantém estado conversacional
  - ‣ instâncias carregadas em pool
  - ✗ cliente não acessa diretamente o MDB
    - ⇒ cliente envia para destinatário
    - ⇒ listener MDB
- ✓ pode ser transaction-aware
  - ⇒ operações em uma única transação



Tipos de EJB: Message-driven Bean

- **Ciclo de vida**
  - não-existe
    - ⇒instanciação, injeção de dependência
  - pronto
    - ⇒ onMessage() trata mensagens



✓Pooling (não é estado)

- **Diferenças do Stateless**
  - não implementa interfaces @Local ou @Remote
    - ⇒ interface *MessageListener* (javax.jms)
  - não é parte do EJB Lite (web profile)

✓Classe anotada com @MessageDriven

⇒ implementa MessageListener

⇒ método onMessage(*javax.jms.Message*)

⇒chamado pelo container

⇒lógica de negócio para tratar mensagens

Tipos de EJB: Message-driven Bean

```
@MessageDriven(mappedName = "jms/MyQueue", activationConfig = {
        @ActivationConfigProperty(propertyName = "destinationType",
                                  propertyValue = "javax.jms.Queue") })
public class SimpleMessageBean implements MessageListener {

    @Resource
    private MessageDrivenContext mDC;

    @Override
    public void onMessage(Message message) {
        try {
            if (message instanceof TextMessage) {
                System.out.println("Queue: I received a TextMessage at " + new Date());
                TextMessage msg = (TextMessage) message;
                System.out.println("Message is : " + msg.getText());
            } else if (message instanceof ObjectMessage) {
                System.out.println("Queue: I received an ObjectMessage at " + new Date());

                ObjectMessage msg = (ObjectMessage) message;
                Employee = (Employee) msg.getObject();
                System.out.println(employee.getSalary());

            } else {
                System.out.println("Not valid message for this Queue MDB");
            }
        } catch (JMSException | RemoteException e) {
            e.printStackTrace();
            mDC.setRollbackOnly();
        }
    }
}
```

[13]

NUCEPE - 2015 - SEFAZ-PI [adaptada]

O Message Driven Bean é usado para processar mensagens síncronas. Sempre há acoplamento entre a aplicação cliente e o Message Driven Bean.

FCC - 2011 - TRT-24 [adaptada]

b) Um Message-Driven Bean é um EJB que possui as interfaces home e remote e apenas um método que recebe qualquer tipo de mensagem.

FCC - 2012 - TRE-CE [adaptada]

O Message-Driven Bean utiliza a API JMS, facilita a quebra de acoplamento entre o cliente e o destino (*Point-to-point ou Publish- subscriber*), e é acionado de forma assíncrona.

[14] VUNESP - 2009 - CETESB

Em um servidor de aplicações, o tipo de Enterprise Bean que é definido, sem nenhuma interface com o cliente, é

a) BPM Entity.

b) CMP Entity.

c) Stateful Session.

d) Stateless Session.

e) Message Driven.

EJB Lite

- ▷ **JEE 6 (**EJB 3.1 JSR 318)
- ✓subconjunto do EJB full
  - não é um produto JEE
  - baseado em sessions beans
    - ✗ Message-driven Beans, serviços remotos
      JavaEE 6: temporizadores e session beans assíncronos
    - ✓ JavaEE 7: SB assíncrono local e temporizadores sem persistência
- ✓web profile (TomEE)
  - roda também em servidor EJB full
  - empacotado no arquivo WAR
- ✓ suporta
  - ⇒ Session Beans (stateful, stateless, singleton)
  - ⇒ Interface local e No-interface
  - ⇒ Interceptors
  - ⇒ transações (CMP e BMP)
  - ⇒ segurança (programática e declarativa)
  - ⇒ JEE7: SB assíncrono local e TimerService sem persistênicia

Tipos de EJB: Interceptors

- **JEE 5** [EJB 3.0: método interceptor - interceptor 1.0]
- **JEE 6** [JSR 318 EJB 3.1 + interceptors 1.1]
- **JEE 7** [JSR 318 interceptores 1.2]

✓ Programação orientada a aspectos (AOP)
  - adicionar funcionalidades (comportamentos) em código existente

- **artefatos**
  - ✓ Classe interceptor @Interceptor
    - método anotado com @AroundInvoke
      - ⇒ apenas um por classe
      - ⇒ retornar *Object*
      - ⇒ parâmetro InvocationContext
      - ⇒ tratar/lançar Exception
  - ✓ Classe ou método alvo
    - declara os interceptors em @Interceptors
      - ⇒ a ordem declarada é a de invocação

Tipos de EJB: Interceptors

```java
@Interceptor
public class LoggingInterceptor {

    @Inject
    private Logger logger;

    @AroundInvoke
    private Object doLog(InvocationContext context) {
        Object obj = null;
        try {
            logger.entering(context.getTarget().toString(), context.getMethod().getName());
            obj = context.proceed();
        } catch (Exception ex) {

        } finally {
            logger.exiting(context.getTarget().toString(), context.getMethod().getName());
        }
        return obj;
    }
}
```

```java
@Stateful
@StatefulTimeout(value = 10, unit = TimeUnit.MINUTES)
@Interceptors(LoggingInterceptor.class)
public class ShoppingCartEJB { ...}
```

```java
@Stateless
public class CustomerEJB {

    public void createCustomer(Customer customer) { . . . }

    @Timeout
    @Interceptors(LoggingInterceptor.class)
    public void sendBirthdayEmail(Timer timer)
            throws InterruptedException, ExecutionException { . . . }
}
```

Tipos de EJB: Cliente Session Bean

- **artefatos**
  - ✓ Business interface (bean interface)
    - declara métodos que o cliente pode invocar
    - anotada com
      - ⇒ @Remote
        - serializable e parâmetros por valor
      - ⇒ @Local
        - parâmetros por referência
      - ⇒ no-interface : container trata como local
  - ✓ Classe Bean
    - implementa os métodos do negócio declarados na interface

Java Persistence API (JPA)

Java Persistence API

▷ **JEE [**JPA 1.0(220), JPA 2.0(317) e JPA 2.1(338)**]**

- ✓ mapeamento objeto-relacional (ORM)
  - impedance mismatch (incompatibilidade conceitual)
    - BD relacionais (tabelas, colunas e linhas)
    - Java (classes, atributos e objetos)
  - utiliza metadados
    - ⇒ anotações e descritores XML
  - provedor de persistência faz o mapeamento
- ✓ roda nos containers EJB, Web e App Client
  - suporte a JSE
- ✗ não suporta noSQL / Schemaless
- ✓ portabilidade de aplicações entre SGBDs
- ✓ abstração para JDBC

Java Persistence API

▷ **JEE [**JPA 1.0(220), JPA 2.0(317) e JPA 2.1(338)**]**

✓ Antes do JPA?

⇒ Entity Beans (EJB) -> Entity CMP

⇒ conexões JDBC

✓ JPA 1.0 -> mais leve (Hibernate)

✓ implementações

⇒ Hibernate

⇒ TopLink, EclipseLink, OpenJPA, Batoo, JDO, etc

✓ implementação de referência

⇒ JPA 2.0 -> EclipseLink 2.0

⇒ JPA 2.1 -> EclipseLink 2.5

Java Persistence API

▷ J2EE x JPA

```
public class Teste {

   Connection conexaoComBanco = null;
   Usuario usuario = null;

   private Usuario buscaUsuario(int id) throws SQLException {
      PreparedStatement preparedStatement = null;
      String consulta = "SELECT EMAIL,NOME FROM USUARIO WHERE ID = ?";
      try {
         conexaoComBanco = getConexaoComBanco();
         preparedStatement = conexaoComBanco.prepareStatement(consulta);
         preparedStatement.setInt(1, id);
         ResultSet rs = preparedStatement.executeQuery();
         usuario = new Usuario();
         while (rs.next()) {
            String email = rs.getString("EMAIL");
            String nome = rs.getString("NOME");
            usuario.setNome(nome);
            usuario.setId(id);
            usuario.setEmail(email);
         }
      } finally {
         if (preparedStatement != null) {
            preparedStatement.close();
         }
         if (conexaoComBanco != null) {
            conexaoComBanco.close();
         }
      }
      return usuario;
   }

}
```

```
public class Teste {

   Usuario usuario = null;

   @PersistenceContext
   EntityManager entityManager;

   private Usuario buscaUsuarioJPA(int id) {
      usuario = entityManager.find(Usuario.class, id);
      return usuario;
   }
}
```

Java Persistence API: Elementos

- Elementos JPA
  - Persistence Unit
    - ⇒ conexões com BDs ou fontes de dados
  - EntityManager
    - ⇒ operações com o BD, CRUD
  - Entities
    - ⇒ metadados ORM
- quatro áreas (tutorial)
  - Java Persistence API
    - ⇒ Persistence Unit, EntityManager, Entities
  - Query Language
    - ⇒ JPQL (baseada em SQL)
  - Java Persistence Criteria API
    - ⇒ SQL OO >> `criteriaQuery.select(pet).where(pet.get(Pet.color).isNull()));`
  - ORM metadata
    - ⇒ annotation / XML

[15] FCC - 2011 - TCE-PR

A JPA

a) pode ser usada fora de componentes EJB e fora da plataforma Java EE, em aplicações Java SE.

b) utiliza persistência gerenciada por contêiner (CMP), ou seja, as classes de entidade e persistência necessitam de um contêiner presente em um servidor de aplicações para serem executadas.

c) utiliza descritores XML para especificar informações do mapeamento relacional de objeto, mas não oferece suporte a anotações.

d) suporta consultas dinâmicas nomeadas nas classes de entidade que são acessadas apenas por instruções SQL nativas.

e) possui uma interface *EntityBeans* que padroniza operações *Create Read Update Delete* (CRUD) que envolvem tabelas.

[16]

CESPE - 2010 - MPU

A versão 3.0 da API de Persistência Java utiliza descritores de implantação, não permitindo uso de anotações.

CESPE - 2010 - MPU

A API de Persistência Java é embasada em ideias contidas em *frameworks* líderes de mercado, como Hibernate, Oracle TopLink e Objetos de Dados Java.

CESPE - 2013 - CNJ

Os objetos mapeados na linguagem Java que devem ser persistidos como objetos precisam utilizar JPA (Java *persistence* API), pois o JPA permite realizar o mapeamento objeto/relacional automatizado e transparente e sua persistência em um banco de dados relacional.

Java Persistence API: Persistence Unit

- persistence.xml
  - diretório META-INF
  - registrar informações
    - ⇒ conexão com SGBD
    - ⇒ implementação JPA utilizada (eclipseLink, hibernate, etc.)
    - ⇒ tipo de transação (CME ou AME)
- Elementos XML
  - <persistence>
    - ⇒ elemento raiz
    - ⇒ versão e namespaces
  - <persistence-unit>
    - ⇒ usado pelo EntityManager para conectar ao BD
    - ⇒ várias unidades de persistência por aplicação
    - ⇒ atributo transaction-type "RESOURCE_LOCAL" "JTA"
    - ⇒ elementos
    - ⇒ <provider>
    - ⇒ <class> <jta-data-source> <properties>

Java Persistence API: Persistence Unit

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
   xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
   http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
   <persistence-unit name="unit-Local" transaction-type="RESOURCE_LOCAL">
      <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
      <class>ejbModule.jpaPackage.Funcionario</class>
      <class>ejbModule.jpaPackage.Dependente</class>
      <exclude-unlisted-classes>true</exclude-unlisted-classes>
      <properties>
         <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
         <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/MyDB"/>
         <property name="javax.persistence.jdbc.user" value="username" />
         <property name="javax.persistence.jdbc.password" value="password" />
      </properties>
   </persistence-unit>

   <persistence-unit name="unit-JTA" transaction-type="JTA">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <jta-data-source>jdbc/MyDB2</jta-data-source>
      <properties>
         <property name="hibernate.show_sql" value="true" />
         <property name="hibernate.format_sql" value="true" />
         <property name="hibernate.hbm2ddl.auto" value="update" />
      </properties>
   </persistence-unit>
</persistence>
```

[17] FCC - 2015 - TRT - 9ª REGIÃO (PR)

Em uma aplicação que utiliza JPA e Hibernate, no arquivo persistence.xml

a) são definidas as configurações do servidor de aplicação, como número de conexões simultâneas permitidas e dados de log.

b) é definido o mapeamento objeto-relacional entre as tabelas do banco de dados e as classes de entidade da aplicação.

c) são definidas as queries nomeadas, que são queries pré-definidas que podem ser chamadas pelo nome a partir de classes de acesso a dados da aplicação.

d) há o mapeamento de componentes da camada de apresentação para os respectivos componentes da camada de acesso a dados da aplicação.

e) são definidas as propriedades de conexão com o banco de dados, como o driver JDBC, a URL de conexão, o nome do usuário e a senha.

Java Persistence API: EntityManager

- javax.persistence.EntityManager
  - interface implementada pelo provedor de persistência
  - gerar e executar código SQL/JPQL
    - ⇒ controla estado e ciclo de vida das entidades
    - ⇒ manipulação de entidades (CRUD)
    - ⇒ controle de concorrência [otimista ou pessimista(JEE6)]
- Contexto de persistência
  - armazena as entidades gerenciadas
  - criado com instância de EntityManager
    - ⇒ injetado (CME)
      - ⇒ @PersistenceContext (JEE)
      - ⇒ container gerencia transações (JTA)
    - ⇒ instanciado (AME)
      - ⇒ utiliza método fábrica (JSE)
      - ⇒ aplicação gerencia
        - ciclo de vida
        - transações
  - Contexto é gerado a partir do **persistence unit**

Java Persistence API: EntityManager

```
public void persistenciaAME() {

   Book book = new Book("H2G2", "The Hitchhiker's Guide to the Galaxy",12.5F,
                        "1-84023-742-2", 354, false);

   EntityManagerFactory emf = Persistence.createEntityManagerFactory("unit-Local");
   EntityManager em = emf.createEntityManager();

   em.getTransaction().begin();
      em.persist(book);
   em.getTransaction().commit();

   book = em.createNamedQuery("findBookH2G2", Book.class).getSingleResult();
   em.lock(book,LockModeType.OPTIMISTIC);

   em.close();
   emf.close();
}
```

```
@Stateless
public class TesteEJB {

   @PersistenceContext(unitName = "unit-JTA")
   private EntityManager em;

   public void persistenciaCME() {

      Book = new Book("H2G2", "The Hitchhiker's Guide to the Galaxy",12.5F,
                           "1-84023-742-2", 354, false);

      em.persist(book);

      book = em.find(Book.class,5034,LockModeType.PESSIMISTIC_WRITE);
   }
}
```

[18] FCC - 2012 - TJ-PE

Sobre JEE e tecnologias relacionadas é correto afirmar que

a) O EntityManager é uma classe identificada com a anotação @Entity que representa o modelo das tabelas do banco de dados.

b) O EntityManager é o serviço central do JPA para todas as ações de persistência e oferece todas as funcionalidades de um DAO genérico.

c) Um servidor de aplicações Java EE possui um único contêiner conhecido como contêiner EJB.

d) Serlvets e JSP rodam no contêiner EJB do servidor de aplicação JEE.

e) Em aplicações que utilizam EJB com JPA, um arquivo persistence.xml pode definir uma única unidade de persistência.

[19] FCC - 2012 - TJ-PE

Quando se utiliza JPA, um *EntityManager* mapeia um conjunto de classes a um banco de dados particular. Este conjunto de classes, definido em um arquivo chamado persistence.xml, é denominado

a) persistence context.

b) persistence unit.

c) entity manager factory.

d) entity transaction.

e) persistence provider.

[20] FCC - 2014 - TRF 3

Considere a seguinte classe desenvolvida em uma aplicação Java que utiliza *JPA/Hibernate*:

```
import javax.persistence.*;
public class NewClassDao {
    public void conectar() {
        EntityManagerFactory a = Persistence.createEntityManagerFactory("conectar");
        EntityManager b = a.createEntityManager();
        EntityTransaction c = b.getTransaction();
        c.begin();
    }
}
```

É correto afirmar que os diversos métodos para executar operações de inserção, consulta, alteração e exclusão de registros no Banco de Dados (*persist, find, merge, remove, createQuery* etc) podem ser acessados por meio do objeto **...I...** . A *String* "conectar" refere-se ao nome da **..I..I...** .

As lacunas **I** e **II** são preenchidas correta e respectivamente com

a) b – interface local

b) c – unidade de persistência

c) c – interface remota

d) a – interface local

e) b – unidade de persistência

Java Persistence API: Entity

- Objeto persistente
  - evoluiu do **Entity Bean** CMP (J2EE)
  - Plain Old Java Object (POJO)
    - ⇒ métodos getters e setters
    - ⇒ interface Serializable (comunicação remota)
- Estrutura
  - classe anotada com @Entity
  - definir chave primária
    - ⇒ simples
      - ⇒ atributo @Id
    - ⇒ composta
      - ⇒ classe com atributos formando a PK
  - configuração por exceção
    - ⇒ container aplica regras padrões
    - ⇒ desenvolvedor ajusta apenas o necessário
- JPA define mecanismos DDL

```
@Entity
public class Livro {
   @Id
   private Long livroId;
   private String titulo;
   private Float preco;
   private String descricao;
   private String isbn;
   private String editora;
   private String edicao;
   private String idioma;

   public Livro() {
   }
   // Getters, setters
}
```

[21] CESGRANRIO - 2012 - Petrobras

Em aplicações Java Enterprise Edition 6, é comum o uso da API JPA. Nessa API, há o conceito de classe de entidade (entity class). Por definição, uma classe de entidade deve, obrigatoriamente, cumprir os seguintes requisitos, EXCETO

a) estar anotada com a anotação Entity ou representada em um descritor XML.

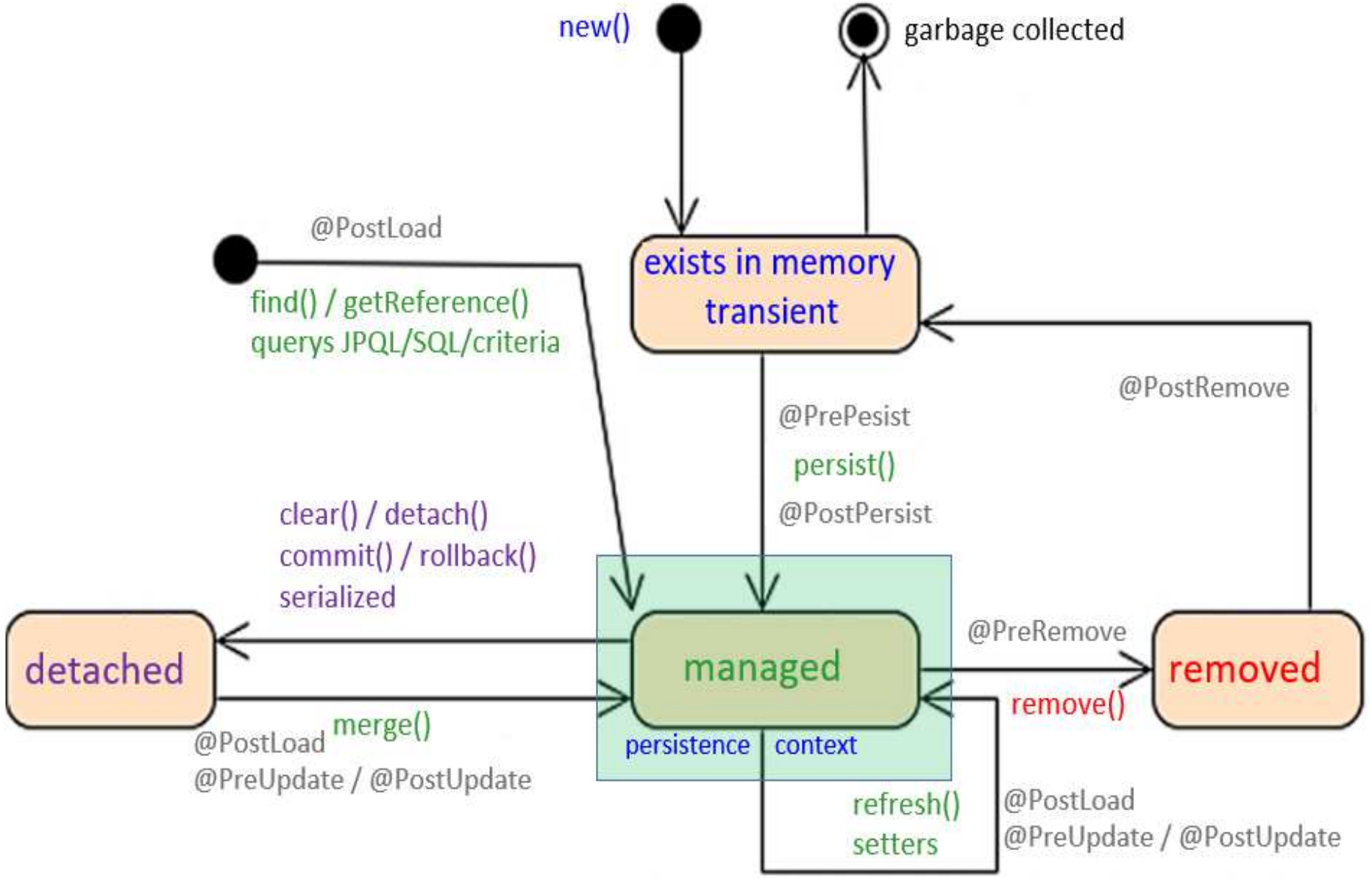
b) não ser qualificada com final.

c) ter as variáveis de instância persistentes qualificadas com private, protected, ou package-private.

d) ter ao menos um construtor, este sem argumentos (no-arg constructor).

e) ter o mesmo nome da tabela correspondente do banco de dados.

Java Persistence API: Entity - Ciclo de vida



[22] FCC - 2010 - TRT 8

Os três estados de objeto definidos pelo framework Hibernate são:

a) Temporário *(Temporary)*, Permanente *(Permanent)* e Resiliente *(Resilient)*.

b) Transiente *(Transient)*, Persistente *(Persistent)* e Resiliente *(Resilient).*

c) Temporário *(Temporary)*, Persistente *(Persistent)* e Destacado *(Detached).*

d) Transiente *(Transient)*, Persistente *(Persistent)* e Destacado *(Detached).*

e) Transiente *(Transient)*, Permanente *(Permanent)* e Resiliente *(Resilient).*

[23] FCC - 2011 - TRT - 19ª Região (AL)

Os estados do ciclo de vida de uma instância de uma entidade, definidos na JPA 2.0, são .

a) novo (new), gerenciado (managed), destacado (detached) e removido (removed).

b) ativo (active), inativo (inactive) e removido (removed).

c) novo (new), temporário (temporary), permanente (permanent) e destacado (detached).

d) novo (new), temporário (temporary) e destacado (detached)

e) gerenciado (managed), temporário (temporary), permanente (permanent) e destacado (detached).

[24] FCC - 2007 - MPU

Objetos que têm uma representação no banco de dados, mas não fazem mais parte de uma sessão do *Hibernate*, o que significa que o seu estado pode não estar mais sincronizado com o banco de dados, são do tipo

a) *transient.*

b) *detached.*

c) *attached.*

d) *persistent.*

e) *consistent.*

Java Persistence API: ORM

- ✓ mapeamento objeto-relacional (ORM)
  - impedance mismatch (incompatibilidade conceitual)
    - BD relacionais (tabelas, colunas e linhas)
    - Java (classes, atributos e objetos)
  - javax.persistence
- ▷ **Tabelas em Classes**
  - instâncias (objetos) = linhas da tabela
  - ✓ @Entity
    - ✓ @Table
    - ✓ @SecondaryTable
      - ⇒ tabela adicional participante de uma mesma entidade
    - ✓ @Embeddable
      - ⇒ separa uma tabela em duas classes
      - ⇒ @Embedded: atributo na Entity

Java Persistence API: ORM

```java
@Entity
@Table(name = "EMPLOYEES")
@SecondaryTable(name = "ADDRESS")
public class Employee {

   @Column(table = "EMPLOYEES")
   private String name;
   @Column(table = "EMPLOYEES")
   private Integer ssid;

   @Column(table = "ADDRESS")
   private String street;
   @Column(table = "ADDRESS")
   private String zipCode;

}
```

```java
@Embeddable
public class Address {
   private String street;
   private String city;
   private String state;
   private String zipcode;
   private String country;
}

@Entity
public class Customer {
   private String firstName;
   private String lastName;
   private String email;
   private String phoneNumber;

   @Embedded
   private Address address;
}
```

Java Persistence API: ORM

- **Colunas em atributos**
  - ✓ @Column
  - ✓ @Temporal
    - ⇒ TemporalType (DATE, TIME, ou TIMESTAMP)
  - ✓ @Transient
    - ⇒ atributo não mapeado para o BD
    - ⇒ também pode usar a keyword ***transient***
  - ✓ @Enumerated
    - ⇒ default: atributo Enum > coluna Integer
    - ⇒ permite definir o tipo a ser mapeado
  - ✓ @Version
    - ⇒ versionamento para bloqueio otimista
- **Tipo de acesso**
  - ⇒ atributos (field) ou métodos getters (property)
  - ✓ @Access
    - ⇒ AccessType (FIELD ou PROPERTY)

Java Persistence API: ORM

```java
@Entity
@Access(AccessType.FIELD)
public class Cliente {

   @Column(name="cliente_cpf", nullable=false)
   private Integer cpf;

   @Column(name="cliente_nome", length=70)
   private String nome;

   @Enumerated(EnumType.STRING)
   private Enum operadora;
   private Integer celular;

   @Temporal(TemporalType.DATE)
   private Date dtNascimento;
   @Temporal(TemporalType.TIMESTAMP)
   private Date dataCriacao;

   @Transient
   private Integer idade;

   @Version
   private Long version;

   // Constructors, getters, setters
}
```

```java
@Entity
@Access(AccessType.PROPERTY)
public class Cliente {

   private Integer cpf;
   private String nome;
   private Enum operadora;
   private Integer celular;
   private Date dtNascimento;
   private Date dataCriacao;
   private Integer idade;
   private Long version;

   @Column(name="cliente_cpf", nullable=false)
   public Integer getCpf() { return cpf; }

   @Column(name="cliente_nome", length=70)
   public String getNome() { return nome; }

   @Enumerated(EnumType.STRING)
   public Enum getOperadora() { return operadora; }

   public Integer getCelular() { return celular; }

   @Temporal(TemporalType.DATE)
   public Date getDtNascimento () { return dtNascimento; }

   @Temporal(TemporalType.TIMESTAMP)
   public Date getDataCriacao() { return dataCriacao; }

   @Transient
   public Integer getIdade() { return idade; }

   @Version
   public Long getVersion() { return version; }

   // Constructors, setters
}
```

Java Persistence API: ORM

- **Colunas em atributos (Chave primária)**
  - **Chave simples**
    - ✓ @Id
    - ✓ @GeneratedValue
      - ⇒ estratégia de geração de PK
      - ⇒ Enum GenerationType {SEQUENCE,TABLE,IDENTITY,AUTO}
        - ✓ @SequenceGenerator
        - ✓ @TableGenerator
  - **Chave composta**
    - ⇒ parâmetro para pesquisa: classe ID (ClassePK)
      - ✓ @IdClass
        - ⇒ ClasseID = POJO (getters, setters, equals e hashcode)
        - ⇒ Entity = atributos da PK com @Id
      - ✓ @EmbeddedId
        - ⇒ ClasseID = @Embeddable
        - ⇒ Entity = ClasseID com @EmbeddedId

Java Persistence API: ORM

```java
@Entity
@SequenceGenerator(name = "Exame_Seq", sequenceName = "Exame_Seq",
allocationSize = 1)
public class Exame {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "Exame_Seq")
    private Integer exameId;

    // Constructors, getters, setters
}
```

```java
@Entity
@TableGenerator(name="Exame_Gen", table="ExameID_Gen", pkColumnName="exameID")
public class Exame {

    @Id
    @GeneratedValue(strategy=GenerationType.TABLE, generator="Exame_Gen")
    private Integer exameId;

    // Constructors, getters, setters
}
```

```java
@Entity
public class Exame {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer exameId;

    // Constructors, getters, setters
}
```

Java Persistence API: ORM

```java
@Entity
@IdClass(DependenteId.class)
public class Dependente {

    @Id
    private Integer dependenteId;
    @Id
    private Integer funcionarioId;

    //getters, setters
}
```

```java
public class DependenteId {

    private Integer dependenteId;
    private Integer funcionarioId;

    //getters, setters, equals, and hashcode
}
```

```java
@Entity
public class Dependente {

    @EmbeddedId
    private DependenteId dependenteId;

    // getters, setters
}
```

```java
@Embeddable
public class DependenteId {

    private Integer dependenteId;
    private Integer funcionarioId;

    // getters, setters, equals, and hashcode
}
```

```java
@Stateless
public class Teste {

    @PersistenceContext
    EntityManager em;

    DependenteId dependenteId = new DependenteId(1, 30);
    Dependente dependente = em.find(Dependente.class, dependenteId);

    Query qryIdClass =
            em.createQuery("SELECT d.funcionarioId FROM Dependente d");

    Query qryEmbeddedId =
            em.createQuery("SELECT d.dependenteId.funcionarioId FROM Dependente d");
}
```

[25]

CESPE - 2015 - TRE GO

A anotação @Entity significa que determinada classe Java é uma entidade do banco de dados. Caso essa entidade tenha nome que não seja o da tabela, será necessário utilizar a anotação @Table.

CESPE - 2015 - TRE GO

Ao se declarar uma coluna que seja a chave primária de uma tabela, é necessário utilizar a anotação @Id.

[26] FCC - 2015 - TRT-4 (RS)

Uma aplicação que trabalha com Hibernate e EJB possui uma classe POJO – Plain Old Java Object utilizada no mapeamento objeto-relacional com uma tabela do banco de dados. Nessa classe, há um atributo calculado chamado valorTotalPedido que, para ser utilizado apenas em tempo de execução e descartado após finalizar o seu serviço temporário, deverá ser anotado com

a) @Embedded

b) @TemporaryAttribute

c) @GeneratedValue

d) @Transient

e) @Basic

[27] FGV - 2014 - PROCEMPA

Considere a seguinte classe com anotações JPA:

Sobre essa classe anotada, analise as afirmativas a seguir.

```
@Entity
@Table(name = "funcionario")
public class Funcionario implements Serializable {
    private static final long serialVersionUID = 2L;
    @Id
    @Column(name = "id", nullable = false)
    private Integer id;
    @Column(name = "nome")
    private String primaryKey;
    @ManyToOne
    private Funcionario chefe;
    // Restante da classe...

}
```

[27] FGV - 2014 - PROCEMPA

I. A anotação @Table é dispensável, neste caso.

II. A chave primária da tabela associada à classe Funcionario é nome.

III. A anotação @ManyToOne introduz, neste exemplo, um autorrelacionamento.

Após o exame das afirmativas, verifica-se que

a) somente I e II são verdadeiras.

b) somente I e III são verdadeiras.

c) somente II e III são verdadeiras.

d) somente II é verdadeira.

e) somente I é verdadeira.

[28] FCC - 2011 - TRT - 23ª REGIÃO (MT)

Considere:

```
@Entity
@Table(name = "domic")
@NamedQueries({
      @NamedQuery(name="Domic.findById", query="SELECT r FROM Domic r WHERE r.id=:id"),
      @NamedQuery(name="Domic.findByNome", query="SELECT r FROM Domic r WHERE r.nome=:nome")
})
public class Domic implements Serializable {

   private static final long serialVersionUID = 1L;
   @Id
   @Column(name = "id", nullable = false)
   private Integer id;
   @Column(name = "nome")
   private String nome;
   @OneToMany(cascade = CascadeType.ALL, mappedBy = "domicId")
   private Collection<Predio> predioCollection;
```

[28] FCC - 2011 - TRT - 23ª REGIÃO (MT)

Em relação à JPA (*Java Persistence API*) é INCORRETO afirmar que

a) @NamedQuery é aplicada para definir várias consultas.

b) @Entity define que haverá correspondência da classe com uma tabela do banco de dados.

c) @Id define que o atributo que está mapeado com tal anotação corresponderá à chave primária da tabela.

d) @Column(name = "id", nullable = false) define que o atributo da classe mapeado com tal anotação deve estar associado à coluna cujo nome é "id", além de definir que tal campo não pode ser nulo.

e) @OneToMany indica que o atributo contém um conjunto de entidades que a referenciam.

Java Persistence API: ORM

- **Relacionamentos**
  - ⇒ tabelas: chaves estrangeiras X classes: atributos
  - ⇒ OneToOne, OneToMany, ManyToOne, ManyToMany
  - ⇒ unidirecionais ou bidirecionais(mappedBy)
  - **unidirecional**
    - ✓ @OneToOne (OneToMany,ManyToOne) [Single-Valued]
      - ✓ @JoinColumn(FK)
    - ✓ @ ManyToMany (OneToMany) [Multi-Valued]
      - ✓ @JoinTable
        - ⇒ joinColumns e inverseJoinColumns (@JoinColumn)
  - **Bidirecional**
    - ✓ mappedBy
      -  @OneToOne, @OneToMany e @ManyToMany
      -  @ManyToOne

Java Persistence API: ORM

Single-valued (unidirecional)

```java
@Entity
public class Endereco {

   @Id
   @Column(name="endereco_id")
   private int Id;
   private String rua;
   private String cep;
}
```

```java
@Entity
public class Cliente {

   @Id
   @Column(name="cliente_id")
   private Long Id;
   @Column(name="cliente_nome")
   private String nome;

   @OneToOne
   @JoinColumn(name="endereco_fk",nullable=false)
   private Endereco endereco;
}
```

```java
@Entity
public class Departamento {

   @Id
   @Column(name="departamento_id")
   private Long id;
   private String nome;
   private String sigla;
}
```

```java
@Entity
public class Empregado {

   @Id
   @Column(name="empregado_id")
   private Long id;
   @Column(name="empregado_nome")
   private String nome;

   @ManyToOne
   @JoinColumn(name="departamento_fk")
   private Departamento departamento;
}
```

Java Persistence API: ORM

Single-valued (unidirecional)

```java
@Entity
public class Cliente {

   @Id
   @Column(name = "cliente_id")
   private Long Id;
   @Column(name = "cliente_nome")
   private String nome;
}
```

```java
@Entity
public class Vendedor {

   @Id
   @Column(name="vendedor_id")
   private Long id;
   private String nome;

   @OneToMany
   @JoinColumn(name="vendedor_fk")
   private List<Cliente> clientes;
}
```

Multi-valued (unidirecional)

```java
@Entity
public class Cliente {

   @Id
   @Column(name="cliente_id")
   private Long Id;
   @Column(name="cliente_nome")
   private String nome;
}
```

```java
@Entity
public class Vendedor {

   @Id
   @Column(name = "vendedor_id")
   private Long id;
   private String nome;

   @OneToMany
   @JoinTable(name="vendedor_cliente",
      joinColumns=@JoinColumn(name="vendedor_fk"),
      inverseJoinColumns=@JoinColumn(name="cliente_fk"))
   private List<Cliente> clientes;
}
```

Java Persistence API: ORM

Multi-valued (unidirecional)

```java
@Entity
public class Livro {

   @Id
   @Column(name="livro_id")
   private Long id;
   private String titulo;
   private Float preco;
   private String isbn;
}
```

```java
@Entity
public class Autor {

   @Id
   @Column(name="autor_id")
   private Long id;
   private String nome;
   private Long cpf;
   @Basic(fetch=FetchType.LAZY)
   private byte[] foto;

   @ManyToMany(fetch=FetchType.EAGER,
      cascade={CascadeType.PERSIST,CascadeType.REMOVE })
   @JoinTable(name = "autor_livro",
         joinColumns=@JoinColumn(name="autor_fk"),
         inverseJoinColumns=@JoinColumn(name="livro_fk"))
   private List<Livro> livros;
}
```

EAGER: @Basic(@Column), @OneToOne e @ManyToOne
LAZY: @OneToMany e @ManyToMany

CascadeType {PERSIST, REMOVE, MERGE, REFRESH, DETACH, ALL}

Java Persistence API: ORM

Single-valued (bidirecional)

```java
@Entity
public class Endereco {

   @Id
   @Column(name="endereco_id")
   private int Id;
   private String rua;
   private String cep;

   @OneToOne(mappedBy="endereco")
   private Cliente cliente;
}
```

```java
@Entity
public class Cliente {

   @Id
   @Column(name="cliente_id")
   private Long Id;
   @Column(name="cliente_nome")
   private String nome;

   @OneToOne
   @JoinColumn(name="endereco_fk",nullable=false)
   private Endereco endereco;
}
```

```java
@Entity
public class Cliente {

   @Id
   @Column(name="cliente_id")
   private Long Id;
   @Column(name="cliente_nome")
   private String nome;

   @ManyToOne
   private Vendedor vendedor;
}
```

```java
@Entity
public class Vendedor {

   @Id
   @Column(name="vendedor_id")
   private Long id;
   private String nome;

   @OneToMany(mappedBy="vendedor")
   @JoinColumn(name="vendedor_fk")
   private List<Cliente> clientes;
}
```

Java Persistence API: ORM

Multi-valued (bidirecional)

```java
@Entity
public class Livro {

   @Id
   @Column(name="livro_id")
   private Long id;
   private String titulo;
   private Float preco;
   private String isbn;

   @ManyToMany(mappedBy="livros")
   private List<Autor> autores;
}
```

```java
@Entity
public class Autor {

   @Id
   @Column(name="autor_id")
   private Long id;
   private String nome;
   private Long cpf;
   @Basic(fetch=FetchType.LAZY)
   private byte[] foto;

   @ManyToMany(fetch=FetchType.EAGER,
      cascade={CascadeType.PERSIST,CascadeType.REMOVE })
   @JoinTable(name="autor_livro",
         joinColumns=@JoinColumn(name="autor_fk"),
         inverseJoinColumns=@JoinColumn(name="livro_fk"))
   private List<Livro> livros;
}
```

[29] FCC - 2014 - TRT - 13ª REGIÃO (PB)

Java Persistence API (JPA) é uma API padrão da linguagem Java para persistência de dados em bancos de dados relacionais.

Em uma aplicação que utiliza JPA

a) pode ser utilizada, como provedor de persistência, as bibliotecas EclipseLink, Hibernate, OracleTopLink, JBossSeam e JDBCProvider.

b) as classes de entidade do banco de dados permitem o mapeamento entre objetos da classe e tabelas do banco de dados, utilizando anotações como @Table, @Entity, @PrimaryKey, @Column, @Constraint, @Foreignkey e @EJB.

[29] FCC - 2014 - TRT - 13ª REGIÃO (PB)

c) todas as operações realizadas nas tabelas do banco de dados, como inserção de dados, alteração, consultas e exclusão, são realizadas sem o uso de instruções SQL, ou seja, o desenvolvedor não precisa conhecer SQL para programar.

d) as configurações de acesso a banco de dados normalmente ficam no arquivo persistence.xml, ligado à aplicação, de forma que se for alterado o servidor de banco de dados não seja necessário alterar o código-fonte Java da aplicação.

e) as relações existentes entre as tabelas do banco de dados não são refletidas nas classes de entidade criadas na aplicação, o que torna a execução mais rápida. O mapeamento de relações é feito em tempo de execução pelas bibliotecas do provedor de persistência.

[30] NC-UFPR - 2015 - COPEL

Em relação ao mapeamento objeto-relacional usando JPA (Java Persistence API) 2.0, assinale a alternativa correta.

a) Quando se usa a anotação @OneToOne(mappedBy="pai"), entende-se que a chave estrangeira desse mapeamento aponta para uma tabela chamada pai.

b) A anotação @Temporal indica que esse atributo é temporário, ou seja, é um sinônimo da anotação @Transient.

c) Quando usamos a anotação @GeneratedValue(strategy= GenerationType.SEQUENCE), devemos informar o valor da propriedade generator, que pode apontar para um @TableGenerator ou @SequenceGenerator.

[30] NC-UFPR - 2015 - COPEL

Em relação ao mapeamento objeto-relacional usando JPA (Java Persistence API) 2.0, assinale a alternativa correta.

d) A anotação @Version está Deprecated e portanto não deve ser utilizada, já que entra em conflito com a JTA (Java Transaction API)

e) A anotação @ManyToMany indica que o relacionamento é bidirecional, e mesmo que seja informado em apenas uma das classes, será possível realizar a navegação (e obter suas respectivas coleções) em ambos os lados.

[31] UERJ - 2015 - UERJ

Java Persistence API (JPA) é uma especificação para frameworks de mapeamento objeto-relacional. Nesse contexto, considere que em um programa em Java existam duas classes, denominadas Pessoa e Projeto. Considere ainda que a classe Pessoa contém a declaração a seguir.

```
@ManyToMany
@JoinTable(name = "PESSOA_PROJETO",
      joinColumns = { @JoinColumn(name = "C1", referencedColumnName = "C2") },
      inverseJoinColumns = { @JoinColumn(name = "C4", referencedColumnName = "C3") })
private List<Projeto> projetos;
```

Dentre as opções abaixo, aquela que lista corretamente as colunas de chaves estrangeiras que devem ser definidas na tabela PESSOA_PROJETO é:

a) C1 e C2

b) C1 e C4

c) C2 e C3

d) C3 e C4

[32] NC-UFPR - 2015 - COPEL

Quanto a JPA (Java Persistence API) 2.0 e seus modos de carregamento (FetchType) Lazy e Eager, identifique as afirmativas a seguir como verdadeiras (V) ou falsas (F):

( ) Eager é o comportamento padrão para relacionamentos muitos-para-muitos.

( ) É preciso cuidar do cascateamento ao usar Eager Load, pois muitos objetos podem ser carregados desnecessariamente.

( ) Lazy apresenta maior consumo de processamento e rede durante a inicialização da aplicação quando comparado com Eager.

[32] NC-UFPR - 2015 - COPEL

Quanto a JPA (Java Persistence API) 2.0 e seus modos de carregamento (FetchType) Lazy e Eager, identifique as afirmativas a seguir como verdadeiras (V) ou falsas (F):

( ) Fazer cache de objetos instanciados via Lazy Load é geralmente desaconselhável, devido ao alto consumo de processamento.

( ) Essas formas de carregamento tornaram-se Deprecated na JPA 2.0.

Assinale a alternativa que apresenta a sequência correta, de cima para baixo.

a) V – F – V – F – V.
b) F – V – F – V – V.
c) V – F – V – V – F.
d) F – F – V – V – F.
e) F – V – F – F – F.

Java Persistence API: ORM

- **Herança**
  - ⇒ conceito OO (impedance mismatch)
  - ⇒ três estratégias
    - ⇒ @Inheritance na entidade raiz
    - ⇒ InheritanceType {SINGLE_TABLE, TABLE_PER_CLASS, JOINED}
  - **Single table (default)**
    - ✓ única tabela
    - ✓ coluna discriminadora (Dtype)
    - melhor desempenho
    - tabela não é normalizada

```java
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="tipo",
   discriminatorType=DiscriminatorType.CHAR)
@DiscriminatorValue("P")
public abstract class Pessoa {

   @Id
   private Long id;
   private String nome;
   private Endereco endereco;
}
```

```java
@Entity
@DiscriminatorValue("F")
public class PessoaFisica extends Pessoa{

   private Long CPF;
}
```

```java
@Entity
@DiscriminatorValue("J")
public class PessoaJuridica extends Pessoa{

   private Long CNPJ;
}
```

Java Persistence API: ORM

- **Table per Class**
  - ✓ uma tabela por classe **concreta**
  - ✓ atributos da superclasse nas tabelas das classes-filhas
  - ✓ tabelas independentes
    - ⇒ controle de chave primária único
  - melhor desempenho
  - tabela não é normalizada

```java
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Pessoa {

   @Id
   private Long id;
   private String nome;
   private Endereco endereco;
}
```

```java
@Entity
public class PessoaFisica extends Pessoa{

   private Long CPF;
}
```

```java
@Entity
public class PessoaJuridica extends Pessoa{

   private Long CNPJ;
}
```

Java Persistence API: ORM

- ▷ **Joined Subclass**
  - ✓ uma tabela por classe **concreta+abstrata**
    - ⇒ cada tabela com seus próprios atributos
  - ✓ chave-primária na tabela raiz
    - ⇒ tabelas-filhas: PK --> referencia tabela-raiz (FK)

✅ mais próxima ao modelo OO ❌ desempenho pior

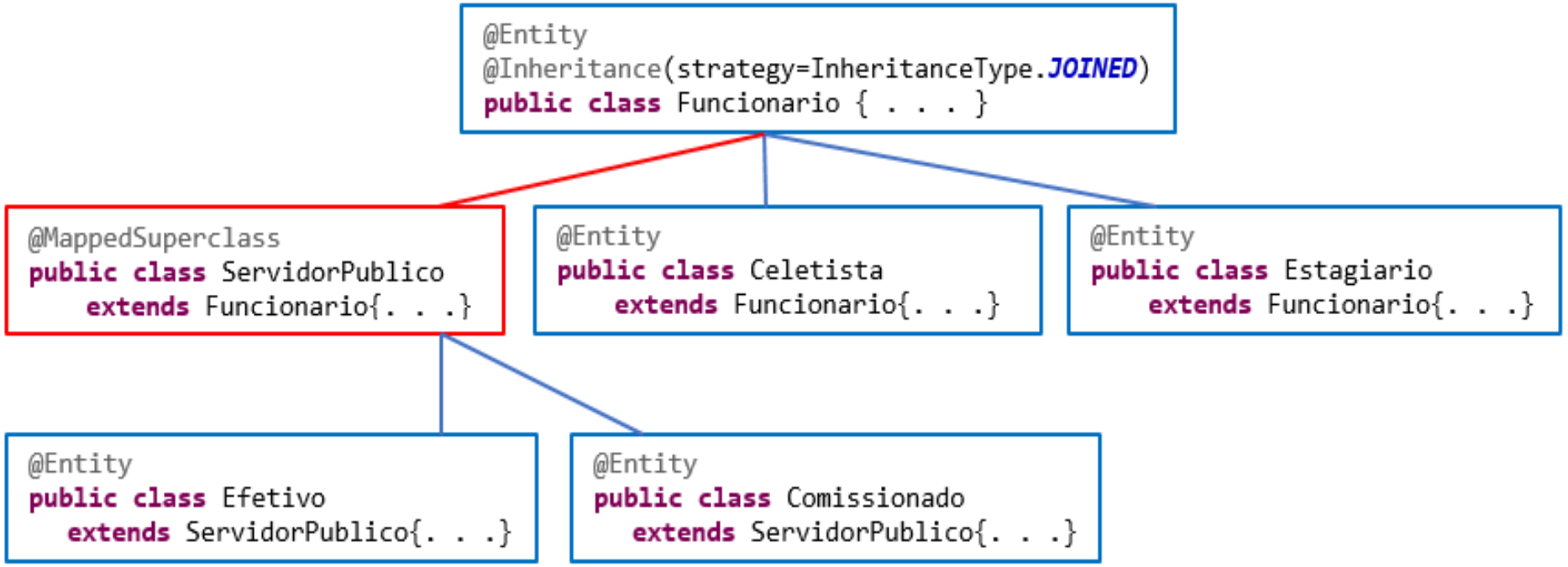
```java
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public abstract class Pessoa {

   @Id
   private Long id;
   private String nome;
   private Endereco endereco;
}
```

```java
@Entity
public class PessoaFisica extends Pessoa{

   private Long CPF;
}
```

```java
@Entity
public class PessoaJuridica extends Pessoa{

   private Long CNPJ;
}
```

Java Persistence API: ORM

- **@MappedSuperclass**
  - ✓ classes da hierarquia que não são persistidas
  - ✓ não visível pelo EntityManager
  - ✓ não suporta relacionamentos
    - ⇒ OneToOne,ManyToOne,OneToMany e ManyToMany
  - ✓ atributos persistidos pelas subclasses (entity)



[33] CESPE - 2015 - FUB

Ao se usar JPA, a forma de armazenamento dos dados das instâncias de uma classe e suas subclasses varia de acordo com a estratégia de herança adotada. Essa variação pode incluir todos os dados armazenados em uma tabela, bem como dados armazenados em tabelas distintas que usam uma coluna de referência.

Java Persistence API: ORM

- ▷ **Recuperação de dados (Queries)**
  - ✓ **Finding by ID (EntityManager)**

```
Dependente dependente = em.find(Dependente.class, 30L);
Cliente cliente= em.getReference(Cliente.class, 101L);
```

  - ✓ **Java Persistence Query Language (JPQL)**
    - ⇒ SQL-OO
    - ⇒ queries nas entidades ou componentes específicos (DAO)
    - ⇒ dinâmicas
      - ⇒ SQL em runtime (custo de tradução)

```
Query qryDinamica = em.createQuery("SELECT d FROM Dependente d WHERE d.nome= :nome");
qryDinamica.setParameter(":nome", "José Augusto Silva");
Dependente dependente = (Dependente) qryDinamica.getSingleResult();
```

Java Persistence API: ORM

- ▷ **Recuperação de dados (Queries)**
  - ✓ **Java Persistence Query Language (JPQL)**
    - ⇒ estáticas
      - ⇒ SQL em start-time (mais eficiente)

```
@Entity
@NamedQuery(name="buscaPorCidade", query="SELECT c FROM Cliente c WHERE c.cidade=?1")
public class Cliente {...}
```

```
public Teste() {
   TypedQuery<Cliente> qryEstatica= em.createNamedQuery("buscaPorCidade",Cliente.class);
   qryEstatica.setParameter(1, "Florianopolis");
   List<Cliente> clientes = qryEstatica.getResultList();
}
```

  - ✓ **Queries nativas (SQL)**
    - ⇒ acoplamento com SGBD específico (portabilidade)
    - ⇒ dinâmica: createNativeQuery()
    - ⇒ estática: @NamedNativeQuery + createNamedQuery()

Java Persistence API: ORM

- ▷ **Recuperação de dados (Queries)**
  - ✓ **Criteria API**
    - ⇒ queries-OO
    - ⇒ cláusulas SQL como métodos
    - ⇒ interfaces CriteriaBuilder e CriteriaQuery

```
@Resource
EntityManager em;

public Teste() {
   CriteriaBuilder builder = em.getCriteriaBuilder();
   CriteriaQuery<Cliente> criteriaQuery= builder.createQuery(Cliente.class);
   Root<Cliente> cliente = criteriaQuery.from(Cliente.class);
   criteriaQuery.select(cliente).where(builder.equal(cliente.get("cidade"),"Florianopolis"));
   TypedQuery<Cliente> query = em.createQuery(criteriaQuery);
   List<Cliente> clientes = query.getResultList();
}
```

[34]

CESPE - 2015 - TRE GO

Para que uma classe Java efetue consultas em determinada entidade do banco de dados, é necessário elaborar o SQL e, depois, convertê-lo para JPQL (Java p*ersistence query language*).

CESPE - 2010 - MPU

A versão 3.0 da API de Persistência Java provê uma linguagem de consulta de persistência Java que é uma forma melhorada da linguagem de consulta do EJB.

CESPE - 2015 - STJ

JPQL (Java Persistence Query Language) é uma linguagem de manipulação de dados adotada para criar, alterar estrutura de tabelas e gatilhos utilizados na especificação JPA (Java Persistence API).

javax.persistence.schema-generation

[35] FCC - 2015 - TRE-AP

Em uma classe de entidade do banco de dados presente em uma aplicação que utiliza JPA existem as seguintes instruções:

```
@NamedQuery(name="Cliente.listarTodos",query="select c from Cliente c")
@Entity
public class Cliente {
// atributos e métodos
}
```

Considere que os atributos e métodos da classe Cliente estão implementados e mapeados adequadamente para a tabela Cliente do banco de dados.

Em uma classe de acesso a dados da mesma aplicação, que possui um objeto **em** válido do tipo EntityManager, para executar a *query* da classe de entidade Cliente e obter os dados retornados em uma lista, utiliza-se:

[35] FCC - 2015 - TRE-AP

a) Query query = em.createNamedQuery("Cliente.listarTodos",Cliente.class);
List <Clientes> clientes = query.getResultList();

b) ResultSet query = em.createQuery("Cliente.listarTodos",Cliente.class);
ArrayList <Clientes> clientes = query.getResultList();

c) Query query = em.createNativeQuery("Cliente.listarTodos",Cliente.class);
List <Clientes> clientes = query.getList();

d) Query query = em.createQuery("Cliente.listarTodos",Cliente.class);
List <Clientes> clientes = query.getResultList();

e) List query = em.createNamedQuery("Cliente.listarTodos",Cliente.class);
ArrayList clientes = query.getResultSet();

[36] UERJ - 2015 - UERJ

Java Persistence Query Language (JPQL) é uma linguagem de consulta que faz parte da especificação JPA. Considere uma aplicação em Java que usa JPA, na qual está definida uma classe de entidade denominada br.app.acme.Cliente.

Além disso, essa aplicação contém o trecho de código abaixo, que cria um objeto do tipo javax.persistence.Query, cuja referência é qry.

```
javax.persistence.Query qry = entityManager.createQuery(
        "SELECT OBJECT (c) FROM br.app.acme.Cliente c " +
        "WHERE c.uf = :uf");
qry.setParameter("uf", "Rio de Janeiro");
```

A expressão adequada para execução da consulta em JPQL representada pela referência qry é:

a) java.util.Collection clientes = qry.getResultList()

b) java.util.Collection clientes = qry.executeQuery()

c) br.app.acme.Cliente[] clientes = qry.getResultList()

d) java.util.Collection clientes = qry.getSingleResult()

[37] FCC - 2015 - TRE-AP

Considere o fragmento de código a seguir, presente em uma classe ideal de acesso a dados de uma aplicação que utiliza JPA.

```
String jpql = "select e from Empregado e where e.cargo = :c";
Query q = entityManager.createQuery(jpql, Empregado.class);
...I...
List <Empregados> empregados = q.getResultList ();
```

Para completar corretamente o fragmento de código de forma que a consulta retorne os empregados cujo cargo seja Gerente, a lacuna **I**deve ser preenchida por

a) q.setString("c", "Gerente");

b) q.setParameter("cargo", "Gerente");

c) q.setString(c, "Gerente");

d) q.setAttribute(c, "Gerente");

e) q.setParameter("c", "Gerente");

Java Database Connectivity

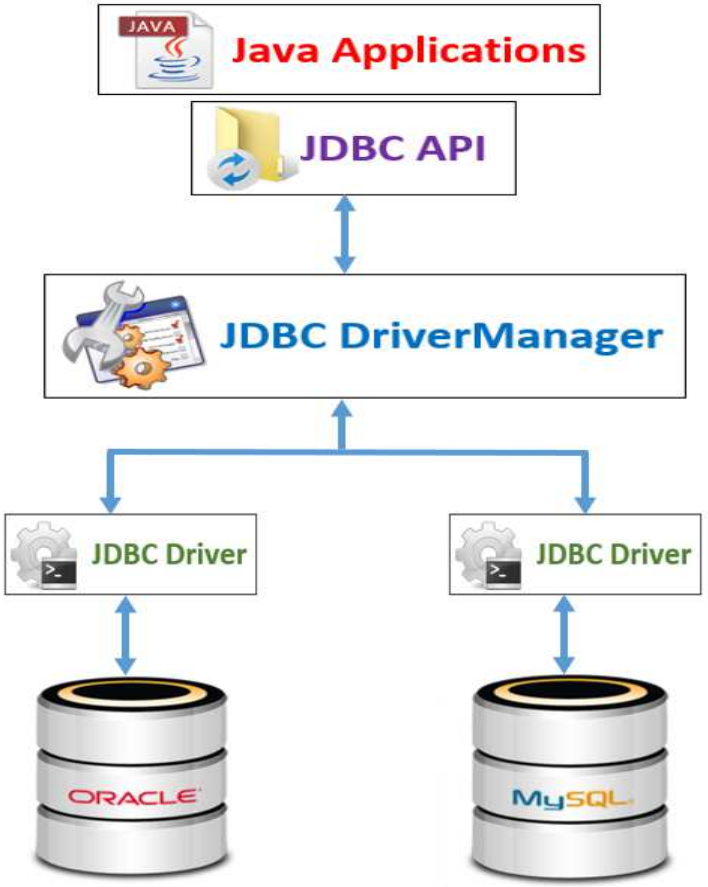
▷ **JSE [**JDBC 4.0 (JSR 221)**]**

✓ interagir com Bancos de dados SQL

- call-level interface
  - ODBC e JDBC

✓ Arquitetura JDBC

- JDBC API (interfaces)
  - ⇒ java.sql.Driver
  - ⇒ java.sql.Connection
  - ⇒ java.sql.Statement
    - ⇒ java.sql.PreparedStatement
      - > performance e SQL Injection
    - ⇒ java.sql.CallableStatement
      - > stored procedures
  - ⇒ java.sql.ResultSet
    - ⇒ javax.sql.RowSet

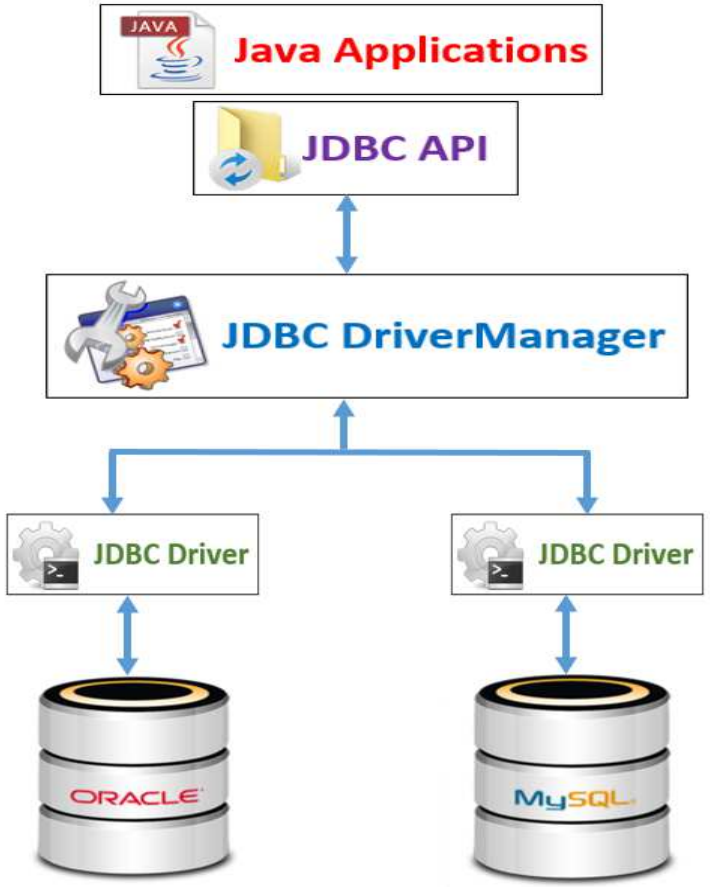


Java Database Connectivity

▷ **JSE [**JDBC 4.0 (JSR 221)**]**

✓ Arquitetura JDBC

- JDBC DriverManager
  - ⇒ classe java.sql.DriverManager
    - ⇒ carga/registro do driver
      - ⇒ JDBC 3: Class.ForName()
      - ⇒ JDBC 4: automático
  - ⇒ DataSource (pooling)
    - ⇒ performance e escalabilidade
    - ⇒ transações distribuídas
- JDBC Driver
  - ⇒ 1. JDBC-ODBC brigde
  - ⇒ 2. Java + Native code
  - ⇒ 3. JDBC-Net + pure Java
  - ⇒ 4. All Java (100% pure Java)



Java Database Connectivity

```
public class Conexao {

   public static Connection getConexao() {
      try {
         Class.forName("oracle.jdbc.OracleDriver");

         return DriverManager.getConnection(
               "jdbc:oracle:thin:@localhost:1521:DBExemplo",
               "user","password");

      } catch (Exception e) {
         return null;
      }
   }

   public static Connection getConexaoDataSource() {
      try {
         InitialContext context = new InitialContext();
         DataSource dataSource = (DataSource) context
               .lookup("java:comp/env/jdbc/DataSourceExemplo");

         return dataSource.getConnection();

      } catch (Exception e) {
         return null;
      }
   }
}
```

Java Database Connectivity

```
public List<Cliente> getClientes() throws SQLException {
   Connection con=null;
   PreparedStatement prepSt=null;
   Cliente cliente;
   Endereco endereco;
   List<Cliente> clientes = new ArrayList<Cliente>();
   try {
      con = Conexao.getConexao(); //Conexao.getConexaoDataSource()
      prepSt = con.prepareStatement("SELECT * FROM clientes WHERE cidade=?1");
      prepSt.setString(1, "Florianopolis");
      ResultSet rs = prepSt.executeQuery();
      while (rs.next()) {
         cliente = new Cliente();
         endereco = new Endereco();
         cliente.setNome(rs.getString("NOME"));
         endereco.setRua(rs.getString("ENDERECO"));
         endereco.setCep(rs.getString("CEP"));
         cliente.setEndereco(endereco);
         cliente.setTelefone(rs.getString("TELEFONE"));
         clientes.add(cliente);
      }
      return clientes;
   } catch (SQLException e) {
      return null;
   } finally {
      prepSt.close();
      con.close();
   }
}
```

Java Database Connectivity

```
public int inserirCliente(Cliente cliente) throws SQLException{
   JdbcRowSet rowSet = null;
   try {
      rowSet = new JdbcRowSetImpl(Conexao.getConexao());
      rowSet.setCommand("SELECT * FROM clientes WHERE cidade=?");
      rowSet.setString(1, "Florianopolis");
      rowSet.execute();

      rowSet.moveToInsertRow();
      rowSet.updateLong("id", cliente.getId());
      rowSet.updateString("NOME", cliente.getNome());
      rowSet.updateString("ENDERECO", cliente.getEndereco().getRua());
      rowSet.updateString("CEP", cliente.getEndereco().getCep());
      rowSet.updateString("TELEFONE", cliente.getTelefone());
      rowSet.insertRow();
      return 1;
   }catch (SQLException e) {
      return 0;
   }finally {
      rowSet.close();
   }
}
```

[38]

CESPE - 2012 - TJ-RO

JDBC, uma biblioteca vinculada a API da arquitetura JEE, define como um cliente pode acessar bancos de dados OO exclusivamente.

CESPE - 2014 - TJ-SE

JDBC faz conexão persistente entre as instâncias beans e as chamadas aos bancos de dados conectados, sendo, portanto, incompatível com sessões do tipo bean stateful.

CAIP-IMES - 2014 - Prefeitura SP [Adaptada]

O JDBC é um conjunto de interfaces e classes que tem como objetivo padronizar o modo com que um aplicativo qualquer se conecte com banco de dados. Possui independência da plataforma do Sistema Operacional e também visa a obter independência de banco de dados.

[39] NUCEPE - 2015 - SEFAZ - PI

No Java, a classe DriverManager fornece os serviços básicos para gerenciamento de drivers JDBC. Quais três argumentos normalmente são passados como parâmetros em seu método getConnection?

a) String url, String user e String password.

b) String url, String port e String database.

c) String url, String user e String database.

d) String user, String port e String database.

e) String user, String password e String database.

```
DriverManager.getConnection( "jdbc:oracle:thin:@localhost:1521:DBExemplo",
                             "user",
                             "password");
```

[40] CESPE - 2009 - IBAMA

```
1   import java.sql.*;
2   public class JoltReport {
3      public static void main(String args[]) {
4         String URL = "jdbc:odbc:CafeJolt";
5         String username = "";
6         String password = "";
7         try {
8            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
9         } catch (Exception e) {
10           System.out.println("“Error");
11           return;
12        }
13        //continua ....
```

As linhas de 7 a 12 permitem carregar o driver que informa às classes JDBC como se comunicar com a fonte de dados. No código, é usada a classe JdbcOdbcDriver e a linha 10 informa a ocorrência de erro no carregamento do driver JDBC/ODBC.

[41] CESPE - 2009 - CEHAP-PB

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
```

Considerando a utilização da linha de código acima no estabelecimento de uma conexão com JDBC, assinale a opção correta.

a) EmbeddedDriver é o principal tipo de driver de conexão JDBC e ODBC em java.

b) Na linha de código, a chamada para Class.forName automaticamente cria uma instância de um driver e o registra com o DriverManager.

c) O trecho de código dado faz os dois passos necessários para a conexão a uma base com JDBC; o EmbeddedDriver faz todo o restante do trabalho de conexão.

d) Class.forName faz parte de outra classe denominada DriverProperty.JDBC.Main, que é utilizada com a tecnologia JDBC.

[42] FCC - 2013 - TRT-15

O método a seguir foi extraído de uma classe Java que permite o acesso a um banco de dados relacional.

```
public int conectar(String a, String b, String c, String d) {
   try {
      Class.forName(a);
      x = DriverManager.getConnection(b, c, d);
      y = x.createStatement();
      return 1;
   } catch (ClassNotFoundException ex) {
      return 2;
   } catch (SQLException ex1) {
      return 3;
   }
}
```

Sobre este método é correto afirmar que :

a) o parâmetro d refere-se ao endereço do banco de dados.

b) x é um objeto da interface SQLConnection.

c) y é um objeto da interface PreparedStatement.

d) o parâmetro b refere-se ao nome do usuário do banco de dados.

e) o parâmetro a refere-se ao driver JDBC.

[43] FCC - 2012 - MPE-AP

Analise as linhas a seguir presentes em um programa Java que não apresenta erros.

```
a = DriverManager.getConnection("jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};DBQ=E:\\bd.mdb", "", "");
b = a.createStatement( );
c = b.executeQuery("select * from cliente where id = "+ valor +"");
```

Considere que os objetos a, b e c são de interfaces contidas no pacote java.sql. Pode-se concluir que esses objetos são, respectivamente, das interfaces

a) Connection, SessionStatement e Result.

b) DriverManager, PreparedStatement e RecordSet.

c) ConnectionStatement, PreparedStatement e RecordSet.

d) Connection, Statement e ResultSet.

e) DaoConnection, Statement e ResultSet.

[44] FCC - 2013 - TRT-12

Considere, abaixo, os métodos encontrados em classes de aplicações Java que acessam banco de dados.

Método 1

```
public int inserir(int varId, String VarNome, double varRenda) {
   int retorno;
   try {
      Class.forName("com.mysql.jdbc.Driver");
      Connection conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/bd007", "root", "1234");
      PreparedStatement st = conn
            .prepareStatement("insert into cliente (id,nome,renda) values (?,?,?");
      st.setInt(1, varId);
      st.setString(2, VarNome);
      st.setDouble(3, varRenda);
      retorno = st.executeUpdate();
   } catch (ClassNotFoundException ex) {
      retorno = 2;
   } catch (SQLException ex1) {
      retorno = 3;
   }
   return retorno;
}
```

Método 2

```
public int inserir(int varId, String VarNome, double varRenda) {
   int retorno;
   try {
      Class.forName("com.mysql.jdbc.Driver");
      Connection conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/bd007", "root", "1234");
      Statement st = conn.createStatement();
      retorno = st.executeUpdate("insert into cliente values (" + varId
            + "," + VarNome + "," + varRenda + ")");
   } catch (ClassNotFoundException ex) {
      retorno = 2;
   } catch (SQLException ex1) {
      retorno = 3;
   }
   return retorno;
}
```

[44] FCC - 2013 - TRT-12

Nas classes, nas quais estes métodos se encontram, foram importados todos os recursos necessários para a execução. O banco de dados, a tabela e o driver JDBC existem e funcionam corretamente.

É correto afirmar que

a) o Método 1 está incorreto, pois o método executeUpdate da interface PreparedStatement precisa receber como parâmetro a instrução SQL insert a ser executada.

b) o Método 2 está incorreto, pois o método executeUpdate da interface Statement não pode receber parâmetros. A instrução insert passada como parâmetro nesse método deveria ser passada como parâmetro para o método createStatement da interface Connection.

[44] FCC - 2013 - TRT-12

c) ambos os métodos estão corretos e executam a mesma operação, apresentando os mesmos resultados.

d) ambos os métodos estão incorretos, pois o método presente tanto na interface Statement como na interface PreparedStatement para incluir dados na tabela do banco de dados é o método executeInsert e não executeUpdate .

e) o Método 1 está incorreto, pois a instrução insert passada como parâmetro para o método PreparedStatement da interface Connection está incompleta. No lugar dos pontos de interrogação devem ser colocados os valores que devem ser incluídos nos campos id , nome e renda da tabela.

Java EE

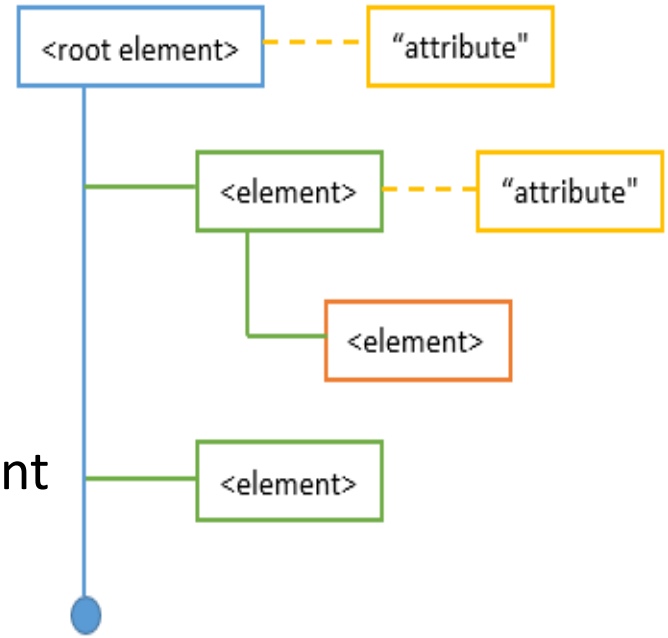

SOAP
Web
services
WSDL
Services
service
REST
XML
Object
Language
method
RPC
SOA
Service
set
messages
use
often
using
architecture
operations
Representational
interface
edit
HTTP
Apache
W3C
also
web

JEE WebServices

- **Serviço**
  - ✓ independente de implementação
    - ⇒ plataforma, SO, frameworks, etc
  - ✓ baixo nível de acoplamento
    - ⇒ interface para acesso (contrato)
    - ⇒ iteração consumidor/provedor de serviço
  - ✓ Interoperabilidade
- **web service**
  - ✓ baseado em XML
    - ⇒ SOAP + WSDL + UDDI
    - ⇒ JAX-WS + JAXB
  - ✓ baseado em HTTP
    - ⇒ RESTful
    - ⇒ JAX-RS

JEE WebServices: JAXB

- **JSE [**JAXB 2.2 (JSR 222)**]**
- **Java Architecture for XML Binding**
  - ✓ mapeamento Java-XML
    - ⇒ documento XML ou Schema Definitions (XSD)
  - Componentes {javax.xml.bind}
    - ✓ container Java WebService - JWS (metro)
    - ✓ contexto JAXB [JAXBContext]
      - mecanismos de conversão
        - ⇒ marshalling: Java => XML
        - ⇒ unmarshalling: XML => Java
    - ✓ annotations
      - @XmlRootElement
        - ⇒ @XmlAttribute e @XmlElement
        - ⇒ @XmlTransient
        - ⇒ @XmlNs



JEE WebServices: JAXB

```java
@XmlRootElement(name="cliente")
@XmlAccessorType(XmlAccessType.FIELD)
public class Cliente {

   @XmlAttribute(required=true)
   private Integer cpf;
   @XmlAttribute(required=true)
   private String nome;

   @XmlElement
   private Integer telefone;

   @XmlElement(required=true)
   private Endereco endereco;

   @XmlTransient
   private Date dataCriacao;
}
```

```java
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Endereco {

   @XmlAttribute(required=true)
   private String rua;
   @XmlAttribute
   private String bairro;

   private String cep;
   private String cidade;
   private String uf;
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<cliente cpf="99999999999" nome="José Augusto Oliveira">
   <telefone>6130331234</telefone>
   <endereco rua="R. Afonso Pena, 2500" bairro="Centro">
      <cep>35500010</cep>
      <cidade>Belo Horizonte</cidade>
      <uf>MG</uf>
   </endereco>
</cliente>
```

```java
public class TestaXml {

   private JAXBContext context;

   public Cliente getCliente(URL url) throws JAXBException {
      context = JAXBContext.newInstance(Cliente.class);
      Unmarshaller unmarshaller = context.createUnmarshaller();
      Cliente cliente = (Cliente) unmarshaller.unmarshal(url);
      return cliente;
   }

   public DOMResult getClienteXml(Cliente cliente) throws JAXBException {
      DOMResult result = new DOMResult();
      context = JAXBContext.newInstance(Cliente.class);
      Marshaller marshaller = context.createMarshaller();
      marshaller.marshal(cliente, result);
      return result;
   }
}
```

JEE WebServices: JAX-WS

▷ Java API for XML-Based Web Services (JSR 224)

- ✓ web service soap-based
  - ⇒ implementa padrões W3C

- ✓ Java Web Services (JWS)
  - ⇒ Java API for XML-Based Web Services (JSR 224)
    - renomeada de JAX-RPC 2.0
  - ⇒ Java Architecture for XML Binding
    - mapeamento Java/XML
  - ⇒ WebServices Metadata (JSR 181)
    - anottations para definição e deploy
    - mapeamento WSDL
  - ⇒ Implementing Enterprise WebServices (JSR 109)
    - definir container JEE para webservices

JEE WebServices: JAX-WS

- **Java API for XML-Based Web Services (JSR 224)**
  - Componentes
    - ✓ container JWS (metro)
      - ⇒ suporta JAX-RPC 1.1 (JSR-101) - Sistemas Legados
    - ✓ **serviço (producer)**
      - WSDL define o serviço
        - ⇒ top-down (wsimport)
          - gera classes a partir do WSDL
        - ⇒ bottom-up (wsgen)
          - gera o WSDL a partir das classes Java
      - @WebService (javax.jws)
        - ⇒ pode combinar com EJB (@Stateless ou @Singleton)
        - ⇒ mapeamento para WSDL
        - ⇒ marshal/unmarshal métodos para mensagens SOAP
        - ⇒ descritor: webservices.xml

JEE WebServices: JAX-WS

- **Java API for XML-Based Web Services (JSR 224)**
  - Componentes
    - ✓ **serviço (producer)**
      - @WebService
        - ⇒ mapeamento para WSDL (javax.jws)
          - @WebMethod
          - @WebParam e @WebResult
          - @OneWay
        - ⇒ ligação (binding) SOAP (javax.jws.soap)
          - @SOAPBinding
            - style: Document ou RPC
            - use: Literal ou Encoded
          - @SOAPMessageHandler (deprecated)

JEE WebServices: JAX-WS

▷ Java API for XML-Based Web Services (JSR 224)

- ▷ Componentes
  - ✓ **serviço (producer)**
    - Contexto WebService (WebServiceContext)
      - ⇒ informações do contexto do serviço
        - pode ser injetado com @Resource
  - ✓ **Cliente (consumer)**
    - gerar artefatos do serviço (WSDL)
      - ⇒ wsimport (metro)
        - @WebServiceClient
      - ⇒ proxy para o endpoint do serviço
        - representação Java do webservice
    - obter uma referência do serviço
      - ⇒ JEE: @WebServiceRef
        - injetada com @Resource
      - ⇒ JSE: new carrega o proxy

JEE WebServices: JAX-WS

▷ Producer

```
@WebService
@SOAPBinding(style = Style.DOCUMENT, use = Use.LITERAL)
public class CardProcessor {

   @WebMethod(operationName = "ValidateCreditCard")
   @WebResult(name = "IsValid")
   public boolean validate(
        @WebParam(name = "CreditCard", mode = WebParam.Mode.IN)
        CreditCard card) {
     // ... código validação
   }

   @WebMethod(operationName = "AuthorizeCreditCard")
   @WebResult(name = "IsAuthorized")
   public boolean authorize(
        @WebParam(name = "CreditCard", mode = WebParam.Mode.IN)
        CreditCard card) {
     // ... código autorização
   }
}
```

JEE WebServices: JAX-WS

▷ WSDL

```
<wsdl:definitions>

  <wsdl:types>

  <wsdl:message name="authorizeRequest">
    <wsdl:part element="impl:authorize" name="parameters">
  </wsdl:message>

  <wsdl:portType name="CardProcessor">
    <wsdl:operation name="validate">

    <wsdl:operation name="authorize">


  <wsdl:binding name="CardProcessorSoapBinding" type="impl:CardProcessor">
    <wsdlsoap:binding style="document"
          transport="http://schemas.xmlsoap.org/soap/http"/>


  <wsdl:service name="CardProcessorService">
    <wsdl:port binding="impl:CardProcessorSoapBinding" name="CardProcessor">
      <wsdlsoap:address
        location="http://localhost:8080/cart/services/CardProcessor"/>
```

JEE WebServices: JAX-WS

▷ Consumer

```
@WebServiceClient
@Stateless
public class ShoppingCart {
  @WebServiceRef
  private CardProcessorService service;
  private CreditCard creditCard;

  @Resource
  private WebServiceContext wsContext;

  public boolean checkout(List<Item> cartItems) throws
                              ServiceException,RemoteException {
    // código finalização da compra

    HttpServletRequest request = (HttpServletRequest) wsContext
        .getMessageContext().get(MessageContext.SERVLET_REQUEST);

    CardProcessor cardProcessor = service.getCardProcessor();

    if (cardProcessor.validate(creditCard)) {
      // ... validado
    }

    if (cardProcessor.authorize(creditCard)) {
      // ... autorizado
    }
  }
}
```

JEE WebServices: JAX-RS

- **Java API for RESTful Web Services (JSR 339)**
  - ✓ web service HTTP-based
    - ⇒ estilo arquitetura REST
    - ⇒ métodos HTTP (PUT, POST, GET, DELETE)
    - ⇒ **recurso**
      - ⇒ identificado por URI
      - ⇒ múltiplas representações (MediaType)
  - Componentes
    - ✓ container JAX-RS (jersey)
    - ✓ serviço (JSR 311 - JEE 6)
      - Root Resource class
        - ⇒ anotada com @Path (javax.ws.rs)
          - define a URI do recurso (completa/parcial)
        - ⇒ pode combinar com EJB (@Stateless ou @Singleton)

```
@Path("/cotacao")
public class CotacaoResource {
```

https://myservices.com.br/cotacao

JEE WebServices: JAX-RS

▷ Java API for RESTful Web Services (JSR 339)

- ▷ Componentes
  - ✓ serviço (JSR 311 - JEE 6)
    - ‣ Resource methods
      - ⇒ anotados com Requests Methods Designators
        - @GET, @POST, @PUT, @DELETE, @HEAD
        - CRUD de entidades
      - ⇒ subrecursos
        - anotação @Path no método
        - URI concatenada com Resource class

```
@Path("/cotacao")
public class CotacaoResource {

  @GET
  @Path("/DollarToReal")
  public String getCotacaoDollarToReal(){
    //implementação
  }
}
```

https://myservices.com.br/cotacao/DollarToReal

JEE WebServices: JAX-RS

▷ Java API for RESTful Web Services (JSR 339)

- ▷ Componentes
  - ✓ serviço (JSR 311 - JEE 6)
    - ‣ Path Templates
      - ⇒ URIs com variáveis embutidas
        - definida entre chaves { }

```
@Path("/usuario/{usuarioID}")
public class UsuarioResource {
```

https://myservices.com.br/usuario/25283
https://myservices.com.br/usuario/10937

      - ⇒ permite expressões regulares

```
@Path("/usuario/{usuarioID : [0-9]* }")
@Path("/usuario/{usuarioID : \\d+ }")
```

      - ⇒ tratamento de parâmetros
        - @PathParam, @QueryParam, @MatrixParam
        - @FormParam
        - @CookieParam e @HeaderParam

JEE WebServices: JAX-RS

▷ Java API for RESTful Web Services (JSR 339)

▷ Componentes

- ✓ serviço (JSR 311 - JEE 6)
  - ‣ Path Templates
    - ⇒ tratamento de parâmetros

```
@GET
@Path("/{m1}/{m2}")
public String getCotacao(@PathParam("m1") String m1, @PathParam("m2") String m2)
```

https://myservices.com.br/cotacao/dolar/real

```
@GET
public String getCotacao(@QueryParam("m1") String m1, @QueryParam("m2") String m2)
```

https://myservices.com.br/cotacao?m1=dolar&m2=real

```
@GET
public String getCotacao(@MatrixParam("m1") String m1, @MatrixParam("m2") String m2)
```

https://myservices.com.br/cotacao;m1=dolar;m2=real

```
@POST
public String getCotacao(@FormParam("m1") String m1, @FormParam("m2") String m2)
```

https://myservices.com.br/cotacao

JEE WebServices: JAX-RS

- ▷ Componentes
  - ✓ serviço (JSR 311 - JEE 6)
    - ‣ Tipos de conteúdo
      - ⇒ 1 recurso várias representações
        - definidas como media types (MIME)
          - tipos MIME ou classe MediaType
      - ⇒ @Produces e @Consumes
        - root resource (classe) ou subresourses (métodos)

```
@Path("cliente")
@Produces(MediaType.TEXT_PLAIN)
public class ClienteResource {

   @GET
   public Response getAsPlainText(){ ...}

   @GET
   @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
   public Response getAsJsonAndXML() {...}

   @PUT
   @Consumes({"application/xml,application/json"})
   public void setCliente(Request request) {...}
}
```

JEE WebServices: JAX-RS

▷ Java API for RESTful Web Services (JSR 339)

- ▷ Componentes
  - ✓ cliente (JAX-RS 2.0 - JSR 339 - JEE 7)
    - JAX-RS 1.1
      - ⇒ Java baixo-nível ou frameworks proprietários
    - JAX-RS 2.0 (*javax.ws.rs.client*)
      - ⇒ construção requests HTTP
      - ⇒ permite expressões regulares
      - ⇒ principais componentes
        - Client
          - criado com ClientBuilder
        - WebTarget
          - representa um recurso (URI)
          - gerar requests e obter responses HTTP
        - Response
          - contém resultado da requisição

JEE WebServices: JAX-RS

```java
@XmlRootElement
@Entity
@NamedQuery(name="clientes", query="SELECT c FROM Cliente c")
public class Cliente {
  private int id;
  private String nome;
  private String endereco;

  //getters e setters
}
```

```java
@Stateless
public class ClienteService {

  @PersistenceContext
  EntityManager em;

  public Cliente getCliente(int id) {
    return em.find(Cliente.class, id);
  }

  public List<Cliente> getClientes() {
    TypedQuery<Cliente> qry = em
        .createNamedQuery("clientes", Cliente.class);
    return qry.getResultList();
  }

  public void save(Cliente cliente) {
    em.persist(cliente);
  }
}
```

JEE WebServices: JAX-RS

```java
@Path("/cliente")
@Produces(MediaType.APPLICATION_XML)
@Consumes(MediaType.APPLICATION_XML)
public class ClienteResource {
  @Inject
  private ClienteService clienteService;


  @GET
  @Path("/{clienteId}")
  public Cliente getCliente(@PathParam("clienteId")int id) {
    return clienteService.getCliente(id);
  }

  @POST
  public void createCliente(Cliente cliente) {
    clienteService.save(cliente);
  }
}
```

https://myservices.com.br/cliente/12345

https://myservices.com.br/cliente

JEE WebServices: JAX-RS

```
public class ClienteRestClient {

  @Inject
  ClienteService clienteService;

  private Cliente getCliente(int id) {
    Client client = ClientBuilder.newClient();
    URI uri = UriBuilder.
        fromUri("https://myservices.com.br/cliente/" + id).build();
    WebTarget target = client.target(uri);
    Invocation invocation = target.request().buildGet();
    Response response = invocation.invoke();
    return response.readEntity(Cliente.class);
  }

  private void createCliente() {
    Cliente cliente = clienteService.clienteToForm();

    Client client = ClientBuilder.newClient();
    URI uri = UriBuilder.
        fromUri("https://myservices.com.br/cliente").build();
    WebTarget target = client.target(uri);
    Invocation invocation = target.request().buildPost(
        Entity.entity(cliente, MediaType.APPLICATION_XML));
    Response response = invocation.invoke();
  }
}
```

Gabarito

[01]	C
[02]	C
[03]	A
[04]	E-C
[05]	B
[06]	B
[07]	B
[08]	A
[09]	B
[10]	E-E-E
[11]	D

[12]	C
[13]	E-E-C
[14]	E
[15]	A
[16]	E-C-C
[17]	E
[18]	B
[19]	B
[20]	E
[21]	E
[22]	D

[23]	A
[24]	B
[25]	C-C
[26]	D
[27]	B
[28]	A
[29]	D
[30]	C
[31]	B
[32]	E
[33]	C

[34]	E-C-E
[35]	A
[36]	A
[37]	E
[38]	E-E-C
[39]	A
[40]	C
[41]	B
[42]	E
[43]	D
[44]	C

Java Enterprise Edition módulo IV

PROVAS DE TI
TUDO PARA VOCÊ PASSAR

Leonardo Marcelino
http://www.itnerante.com.br/profile/LeonardoMarcelino