Autores : Eduardo Guerra, Johanlemberg, Ednelson, Cavalero e Douglas

FREE: SCBCD 5.0 Beta Certification Exam Detailed Exam Objectives: Sun Certified Business Component Developer (SCBCD)

Section 1: EJB 3.0 Overview

1. Identify the uses, benefits, and characteristics of Enterprise JavaBeans technology, for version 3.0 of the EJB specification.

- Desenvolvimento de aplicações de negócio **orientadas a objetos** e **distribuídas**.
- Suporte para desenvolvimento, deployment e uso de Web Services.
- Facilita o desenvolvimento de aplicações, sendo que o desenvolvedor não precisa se preocupar com tarefas de baixo nível (conection pool, multi-threading e gerenciamento de transações)
- Pode ser deployed em vários lugares sem ser recompilado
- Cobre aspectos de desenvolvimento, deployment e runtime.
- Combinação de componentes desenvolvidos por diferentes fornecedores.
- Interoperabilidade entre componentes Java, Java EE e componentes não-java.
- Compatível com outras APIs de Java e CORBA.

2. Identify the APIs that all EJB 3.0 containers must make available to developers.

- EJB 3.0 APIs + Java Persistent API
- JTA 1.1
- JMS 1.1
- JavaMail 1.4
- JAF 1.1
- JAXP 1.2
- JAXR 1.0
- JAX-RPC 1.1
- JAX-WS 2.0
- SAAJ 1.3
- Conector 1.5
- Web Services 1.2
- Web Services Metadata 2.0
- Common Annotations 1.0
- StAX 1.0

As seguintes APIs são providas pela J2SE 5.0

- JDBC
- RMI-IIOP
- JNDI
- JAXP
- Java IDL
- JAAS

3. Identify correct and incorrect statements or examples about EJB programming restrictions.

- Não deve possuir campos estáticos (somente para leitura com `final`). Alguns containers podem utilizar mais de uma VM para executar os EJBs.
- Não deve usar sincronização de threads de forma primitiva (não funciona se houver mais de uma VM).
- Não devem usar AWT para tentar exibir ou pegar entrada de dados de um usuário.
- Não deve usar o pacote `java.io` para acessar arquivos diretamente.
- Não deve usar socket para aceitar conexões, escutar um socket ou usar para multicast. Ele pode ser cliente de socket mas não um servidor!!!
- Não deve acessar campos privados ou protegidos (que não seriam acessíveis normalmente) usando Reflection.
- Não deve: criar um classloader; obter o classloader; setar o classloader; setar o security manager; criar um security manager; parar a JVM; mudar o input, output e error streams (isto é reservado para o container)
- Não deve procurar setar uma socket factory usada por ServerSocket, Socket ou stream handler factory usado por URL (funções de networking devem ser tratadas pelo container).
- Não deve gerenciar threads: iniciar, parar, suspender ou "resume", ou mudar a prioridade ou o nome. O mesmo para thread groups.
- Não deve tentar ler ou escrever no deployment descriptor.
- Não deve tentar obter a security policy de uma fonte de código.
- Não deve carregar uma biblioteca nativa.
- Não deve tentar ganhar acesso a packages ou classes que não estão disponíveis para os EJBs.
- Não deve tentar definir uma classe em um package.
- Não deve tentar acessar ou modificar objetos de configuração de segurança: Policy, Security, Provider, Signer e Identity.
- Não deve permitir o uso de subclasses e substituição de objetos para modificar o algoritmo de serialização.
- Não deve retonar ou passar como parâmetro:
  - `this`
  - `SessionContext.getBusinessObject`
  - `SessionContext.getEJBObject`
  - `SessionContext.getEJBLocalObject`
  - `EntityContext.getEJBObject`

- `EntityContext.getEJBLocalObject`

4. **Match the seven EJB roles with the corresponding description of the role's responsibilities.**

- Enterprise bean provider
  - Produtor de Enterprise beans
  - Passa um ejb-jar com as classes dos beans
  - Passa os beans metadata (annotation ou deployment descriptor) com informação estrutural e declaração de dependências externas.
- Application Assembler
  - Combina os EJBs em uma aplicação deployable
  - Combina os EJBs com outros tipos de componentes
  - Acontece antes do deploy
- Deployer
  - Realiza o deploy dos ejb-jar no container
  - Deve suprir todas as dependências (conection pools, filas JMS)
  - Usa ferramentas providas pelo container
  - Quando necessário, gera as classes (stubs) utilizadas pelo container.
- EJB Server Provider
  - Especialista em gerenciamento de transações, objetos distribuídos e outros serviços de baixo nível.
  - Provê o servidor que irá gerenciar os EJBs.
  - Por enquanto, é similar ao EJB Container Provider, já que não existe uma interface definida entre eles.
- EJB Container Provider
  - Provê ferramentas para fazer o deploy
  - Runtime suport para os EJBs
  - Provê aos beans: controle de transação, gerência de segurança, distribuição para clientes remotos, gerência escalável de recursos e outros serviços.
- Persistence Provider
  - Provê a API de persistência
  - Desenvolve uma escalável e transacional plataforma para gerência de persistência.
  - Pode ser o mesmo EJB Container Provider ou um forncedor que provê uma API de persistência plugável.
- System Administrator
  - Responsável pelo ambiente em que a aplicação é executada
  - Inclui o EJB server e container

5. **Describe the packaging and deployment requirements for enterprise beans.**

Arquivo ejb-jar

- Deve conter um deployment descriptor em META-INF/ejb-jar.xml
- Deve conter incluídos ou por referência (o arquivo está contido em um jar que é referenciado no atributo classpath do Manisfest File, ou é referenciado por este jar referenciado):

    - Enterprise bean class
    - Enterprise bean business interfaces
    - Web Services Endpoint Interfaces
    - Home e Component interfaces
    - Interceptor classes
    - Primary Key classes (se houver entity beans)
    - Quaisquer outras dependencies retornadas ou usadas como parâmetros.

- Não deve conter os stubs, que devem ser gerados pelo container no momento do deploy ou dinamicamente.

Arquivo ejb-client

- Deve possuir as interfaces acessadas pelo cliente (no caso dos stubs gerados estaticamente, os stubs para acessar o container).
- Deve possuir as classes dependentes, usadas como retorno e parâmetro

6. Describe the purposes and uses of annotations and deployment descriptors, including how the two mechanisms interact, how overriding is handled, and how these mechanisms function at the class, method, and field levels.

- As configurações podem ser feitas no deployment descriptor ou com o uso de annotation nas classes, métodos ou atributos.
- Se definido a nível de classe, aquela propriedade vale para todos os métodos da classe. Se hover a definição no método, ela sobrepõe a definição da classe.
- As anotações de atributos podem ser usadas nos métodos setter dos mesmos.
- O deployment descriptor sempre sobrepõe as definições feitas pelas annotations.

Section 2: General EJB 3.0 Enterprise Bean Knowledge

1. Identify correct and incorrect statements or examples about the lifecycle of all 3.0 Enterprise Bean instances, including the use of the @PostConstruct and @PreDestroy callback methods.

- Os EJBs são criados pelo container. A sua criação e destruição é controlada pelo container (exceto statefull, será descrito na sessão 3), podendo criar de acordo com a demanda, ou manter as instâncias em um pool. (se pergundar alguma coisa na prova colocar como implementation dependent).
- Os callback methods podem ser colocados no próprio bean ou em interceptors. Para configura-los basta colocar a anotação @PostConstruct e @PreDestroy.

- Em um bean a assinatura é a seguinte: void <METHOD>()
- Em um interceptor a assinatura é a seguinte: void <METHOD>(InvocationContext)
- Os métodos podem ter qualquer modificador de acesso, mas Não pode ser `static` ou `final`.
- Um mesmo método pode ser anotado para mais de um callback.
- Os callback são executados no mesmo stack.

Ordem de execução de callback (pode ser modificada no deployment descriptor pelo elemento `interceptor-order`):

- Default Interceptors (somente definidos no deployment descriptor) são invocados primeiro.
- Os callback definidos em interceptors são executados antes dos definidos no próprio bean.
- A ordem dos interceptors colocada na anotation @Interceptors é a ordem de execução dos mesmos.
- Se um interceptor possui uma superclasse que define um callback method, o método da superclasse é invocado primeiro.
- Se o bean possui superclasses com callback methods, os da superclasses são invocados primeiro.
- Se um método com callback for override, a não ser que o método possua uma annotation, o método que sobrepôs não será invocado (e sim o da superclasse).

2. Identify correct and incorrect statements or examples about interceptors, including implementing an interceptor class, the lifecycle of interceptor instances, @AroundInvoke methods, invocation order, exception handling, lifecycle callback methods, default and method level interceptors, and specifying interceptors in the deployment descriptor.

Considerações Gerais sobre Interceptor

- Um interceptor não precisa implementar interface ou estender alguma classe, só precisa ser declarado como interceptor (precisa ter um construtor vazio).
- Um interceptor pode receber dependency injection e pode acessar praticamente tudo que o bean pode…
- O ciclo de vida de um interceptor é o mesmo do bean que está ligado.
- É criado antes do @PostActivate e @PostCreate e destruído depois do @PreDestroy e @PrePassivate.
- Um interceptor pode capturar e tratar as exceptions lançadas pelo método e enviar runtime exceptions e exceptions do tipo que estiver declarado na business interface.
- Se um interceptor que possuir o @ArroudInvoke jogar uma exception antes de chamar proceed(), os seguintes não serão chamados.

Definindo Interceptors:

- Os default-interceptors:
  - Se aplicam para todos os components no ejb-jar
  - São declarados no deployment descriptor
  - A annotation @ExcludeDefaultInterceptors ou o elemento `exclude-default-interceptors` pode ser usado para excluir algum default interceptor (pode ser aplicado para uma classe ou um método).
- Method-level interceptors:
  - Definidos com a annotation @Interceptors no método
  - A mesma forma pode ser utilizada no bean, e os interceptors serão aplicados em todos os business methods.
  - A annotation @ExcludeClassInterceptors e o elemento `exclude-class-interceptors` podem ser usados para excluir os interceptors da classe para algum método.

Interceptors no Deployment Descriptor

- Declaração:
  - O elemento `interceptor` é usado para declarar um interceptor
  - Os elementos around-invoke, pre-construct, post-destroy, pré-passivate e post-activate são usados para declarar os callback methods.
- Binding

```
<interceptor-binding>
   <ejb-name> //Nome do EJB ou * (wildcard)
   <interceptor-class> //Classe do interceptor
   <method-name> //Nome do método (opcional)
   <method-params> //Possui vários method-param com os params
(opcional)
</interceptor-binding>
```

Interceptors do tipo @AroundInvoke

- Precisa ter a seguinte assinatura:

`public Object <METODO>(InvocationContext) throws Exception`

```
public interface InvocationContext{
  Object getTarget() //Bean invocado
  Method getMethod() //Metodo interceptado
  Object[] getParameters() //Parametros dos Métodos
  void setParameters(Object[]) //Altera os parâmetros
  Map<String,Object> getContextData() //Passar parâmetros entre invoc.
  Object proceed() throws Excep.//Executa método ou prox. interceptor
}
```

3. Identify correct and incorrect statements or examples about how enterprise beans declare dependencies on external resources using JNDI or dependency injection, including the general rules for using JNDI, annotations and/or deployment descriptors, EJB references, connection factories, resource environment entries, and persistence context and persistence unit references.

Tipo de Dependências (em fields ou métodos setter – getters não são anotados)

- @Resource
  - type: A classe do recurso (default igual ao tipo do campo anotado ou ao parâmetro do método setter)
  - name: O JNDI name do recurso em `java:comp/env` (default igual ao nome do campo)
  - authenticationType: APPLICATION / CONTAINER
  - shareable: se o recurso é compartilhado (true/false)
  - mappedName: específico e não-portável
- @EJB
  - beanInterface: Classe do business interface (default igual ao tipo do campo anotado ou ao parâmetro do método setter)
  - name: O JNDI name do recurso em `java:comp/env` (default igual ao nome do campo)
  - mappedName: específico e não-portável

Usando JNDI diretamente

- A interface EJBContext possui um método `Object lookup(String name)` que pode ser utilizado para obter uma referência.
- Pode-se também criar um `InitialContext` e usar o seu método `lookup()`, não sendo necessário o uso do narrow.

Declarando no Deployment Descriptor

- Declaração de parâmetros:

```
<env-entry>
  <description>
  <env-entry-name>
  <env-entry-type>
  <env-entry-value>
</env-entry>
```

- Referências de EJBs:

```
<ejb-ref> ou <ejb-local-ref>
  <description>
  <ejb-ref-name>
```

```
  <ejb-ref-type>
  <local> e/ou <remote> e/ou <home> (interface)
  <ejb-link> (ligação com outros ejbs declarados, normalmente com outro nome)
</ejb-ref> ou </ejb-local-ref>
```

- Outros Recursos (Datasources, Queue Conection Factory)

```
<resource-ref>
   <description>
   <res-ref-name>
   <res-type>
   <res-auth>
   <res-sharing-scope>
</resource-ref>
```

- Destino de Mensagens (Queues ou Topics)

```
<message-destination-ref>
  <description>
  <message-destination-ref-name>
  <message-destination-type> (exemplo : javax.jms.Queue)
  <message-destination-usages> (Produces e/ou Consumes)
</message-destination-ref>
```

- Persistence Unit e Context

```
<persistence-unit-ref> ou <persistence-context-ref>
  <description>
  <persistence-unit-ref-name> ou <persistence-context-ref-name>
  <persistence-unit-name> (opcional : nome da persistence unit definido no
                                  persistence.xml)
<persistence-unit-ref>
```

4. Identify correct and incorrect statements or examples about Timer Services, including the bean provider's view and responsibilities, the TimerService, Timer and TimerHandle interfaces, and @Timeout callback methods.

Considerações Gerais

- Agenda serviços em um horário específico e em intervalos.
- Não deve ser usado para eventos de tempo real.
- Timers não podem ser criados para statefull ou entities (EJB 3.0). Para entity do EJB 2.1 pode!
- Os métodos são chamados dentro de uma transação.
- Caso o conteiner caia, os timers são mantidos!

Interface TimerService

- Pode ser obtido via getTimerService() do EJBContext, ou dependency injection.
- 4 métodos createTimer():
  - Duração
  - Duração + Intervalo
  - Data Inicial
  - Data Inicia + Intervalo
- Os métodos createTimer recebe um parâmetro serializable, que é armazenada no Timer e pode ser recuperada depois.
- O método getTimers() retorna os timers associados com o bean.

Chamada no Timeout

- O bean teve (ou):
  - Implementar a interface TimedObject e o método ejbTimeout(Timer timer)
  - Ter um método com a anotação @Timeout com assinatura void <METHOD>(Timer timer) e não deve ser declarado final ou static.
  - O método deve ser declarado no deployment descriptor com o elemento `timeout-method`.
- O método não deve jogar application exceptions.
- Se vários timers tiverem o timeout próximos, não é garantida a ordem que o callback method vai ser chamado.
- O método getCallerPrincipal() deve retornar a representação de um usuário não autenticado.
- Se um timer só executa uma vez, ele é removido depois do callback.
- Qualquer método em Timer retorna NoSuchObjectLocalException depois que o callback termina.
- Se um timer expirar várias vezes enquanto o container estiver fora, ele deve executar pelo menos uma vez o método ao reiniciar.
- O container tenta o timeout novamente se a transação sofrer um rollback.

Chamada no Timeout

- A interface Timer

```
public interface Timer{
   void cancel(); //Cancela um timer
   long getTimeRemaining(); //Tempo para próximo timer
   Date getNextTimeout();//Data do próximo timeout
   TimerHandler getHandler();//Representação serializada do Timer
                             //Tem o método getTimer()
   Serializable getInfo();//Passada como parâmetro na criação
}
```

5. **Identify correct and incorrect statements or examples about the EJB context objects that the container provides to 3.0 Session beans and 3.0 Message-Driven beans, including the security, transaction, timer, and lookup services the context can provide.**

Interface MessageDrivenContext

- getCallerPrincipal() – retorna o invoker (java.security.Principal)
- isCallerInRole() – verifica se o invoker está em um "role" (não pode ser chamado por um MDB)
- setRollbackOnly() – marca uma transação para rollback (somente com container managed transaction)
- getRollbackOnly() – Verifica se a transação está marcada para rollback (somente com container managed transaction)
- getUserTransaction() – Retorna a transação (javax.transaction.UserTransaction) – somente com bean managed transaction
- getTimerService() – retorna um TimerService (somente Stateless)
- getEJBHome();getEJBLocalHome() – Retorna as respectivas interfaces (não podem ser usados por MDBs)
- lookup() – busca no JNDI

Interface SessionContext (Todos acima mais os seguintes)

- getMessageContext() – retorna o contexto de mensagem de um web service endpoint (somente Stateless).
- getBusinessObject(Class) – retorna uma business interface.
- getInvokedBusinessInterface() – retorna o business session interface que foi invocado.
- getEJBObject(); getEJBLocalObject();– Retorna as respectivas interfaces.

6. **Identify correct and incorrect statements or examples about EJB 3.0 / EJB 2.x interoperability, including how to adapt an EJB 3.0 bean for use with clients written to the EJB 2.x API and how to access beans written to the EJB 2.x API from beans written to the EJB 3.0 API.**

Clientes de EJB 2.X em beans acessando EJB 3.0

- Deve prover a interface Home ou LocalHome
- O Session bean deve designar as interfaces usa as anotações @RemoteHome e @LocalHome
- Devem possuir o método create() para criar uma instância do bean (Stateless)
- Devem possuir pelo menos um método create<METHOD>() para a construção de uma instância do bean (Statefull)

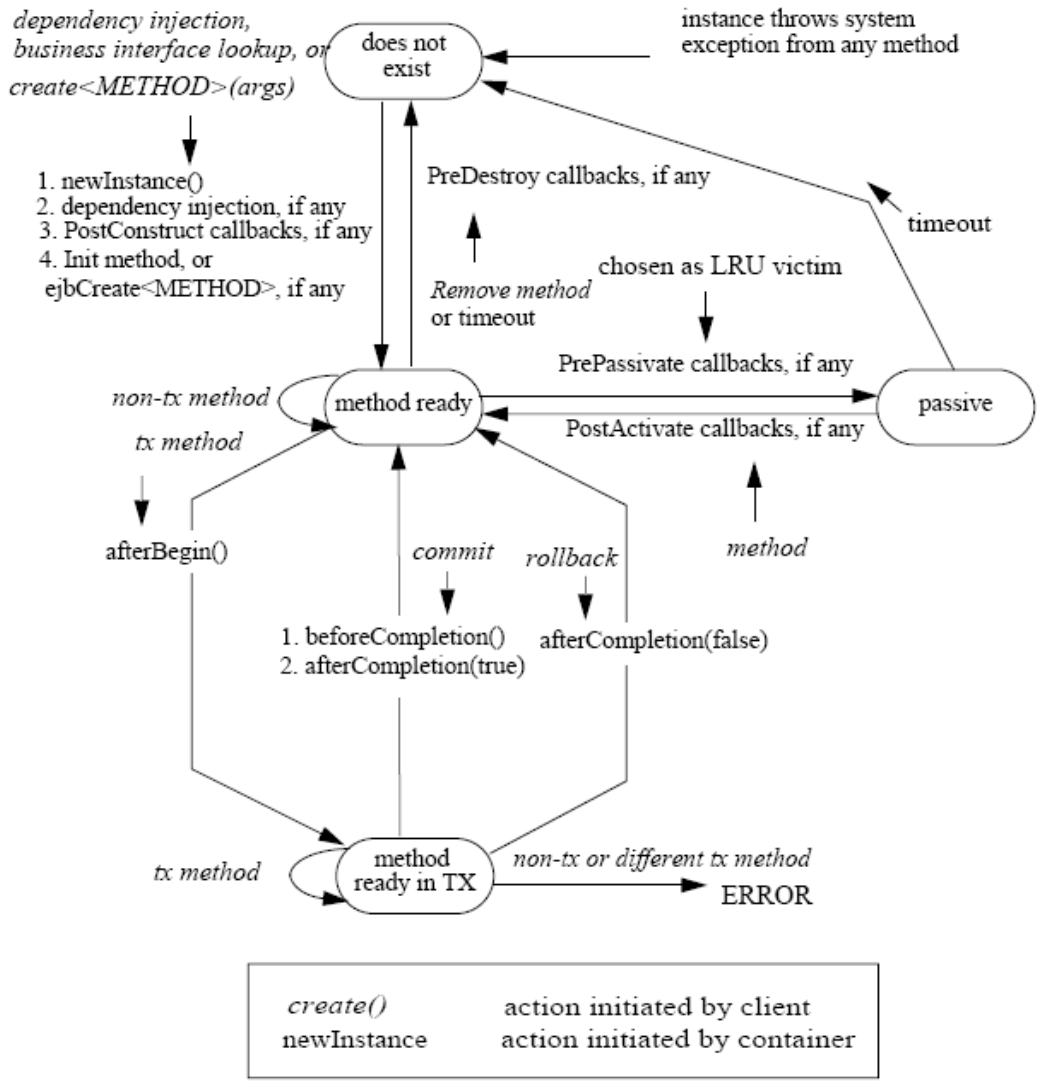
- O container deve prover a implementação das classes EJBHome e EJBObject (ou as variações com Local)

EJB 3.0 sendo clientes de EJB 2.X

- Pode ser usada a anotação @EJB para recuperar a interface Home por injeção de dependência.
- Eles podem compartilhar a mesma transação

Section 3: EJB 3.0 Session Bean Component Contract & Lifecycle

Figure 5 Life Cycle of a Stateful Session Bean Instance



1. **Identify correct and incorrect statements or examples that compare the purpose and use of Stateful and Stateless Session Beans.**

2. **Identify correct and incorrect statements or examples about remote and local business interfaces for Session Beans.**

- Um Bean pode ter mais de uma business interface.
- Se só tiver uma interface, esta é a business interface;
- Esta é Local por padrão. Para ser Remote, necessita da anotação @Remote na classe ou na interface ou no deployment descriptor.
- Se tiver mais de uma, (com exceção de Serializable, Externalizable, javax.ejb.*) alguma deve ser explicitamente designada como business interface do Bean pelo uso da anotação @Local ou @Remote na classe, na interface ou pelo deployment descriptor.
- Uma interface não pode ser Local e Remote ao mesmo tempo.
- A business interface não pode estender javax.ejb.EJBObject ou EJBLocalObject
- Se um bean tem uma interface local e nao a implementa nem especifica uma usando a anotação, é preciso usar <bussiness-local> com o nome completo.
- Idem para beans remotos.
- Tipos: session, entity e message-driven. Os tipos **não** podem ser mudados pelo xml se já tiverem sido escolhidos nas anotações.

```
<session>
<ejb-name>
<ejb-class>
<business-local>
<business-remote>
<home>
<remote>
<local-home>
<local>
<session-type>
</session>
```

```
@Target(TYPE) @Retention(RUNTIME)
public @interface Remote {
Class[] value() default {}; // list of remote business interfaces
}
@Target(TYPE) @Retention(RUNTIME)
public @interface Local {
Class[] value() default {}; // list of
```

3. **Write code for the bean classes of Stateful and Stateless Session Beans.**

```
@Stateful public class CartBean implements ShoppingCart {
private float total;
private Vector productCodes;
public int someShoppingMethod(){...};
...
}
```

```
@Stateless @Remote
public class CalculatorBean implements Calculator {
```

```
public float add (int a, int b) {
return a + b;
}
public float subtract (int a, int b) {
return a - b;
}
}
public interface Calculator {
public float add (int a, int b);
public float subtract (int a, int b);
}
```

4. Identify correct and incorrect statements or examples about the lifecycle of a Stateful Session Bean including the @PrePassivate and @PostActivate lifecycle callback methods and @Remove methods.

The objects that are assigned to the instance's non-`transient` fields and the non-`transient` fields of its interceptors after the `PrePassivate` method completes must be one of the following.

- A serializable object[10].
- A `null`.
- A reference to an enterprise bean's business interface.
- A reference to an enterprise bean's remote interface, even if the stub class is not serializable.
- A reference to an enterprise bean's remote home interface, even if the stub class is not serializable.
- A reference to an entity bean's local interface, even if it is not serializable.
- A reference to an entity bean's local home interface, even if it is not serializable.
- A reference to the `SessionContext` object, even if it is not serializable.
- A reference to the environment naming context (that is, the `java:comp/env` JNDI context) or any of its subcontexts.
- A reference to the `UserTransaction` interface.
- A reference to a resource manager connection factory.
- A reference to a container-managed `EntityManager` object, even if it is not serializable.
- A reference to an `EntityManagerFactory` object obtained via injection or JNDI lookup, even if it is not serializable.
- A reference to a `javax.ejb.Timer` object.
- An object that is not directly serializable, but becomes serializable by replacing the references to an enterprise bean's business interface, an enterprise bean's home and component interfaces, the references to the `SessionContext` object, the references to the `java:comp/env` JNDI context and its subcontexts, the references to the `UserTransaction` interface, and the references to the `EntityManager` and/or `EntityManagerFactory` by serializable objects during the object's serialization.

This means, for example, that the Bean Provider must close all JDBC™ connections in the `PrePassivate` *method and assign the instance's fields storing the connections to* `null`.

The `PrePassivate` and `PostActivate` lifecycle callback interceptor methods are only called on stateful session bean instances.
The `PrePassivate` callback notification signals the intent of the container to passivate the instance.
The `PostActivate` notification signals the instance it has just been reactivated. Because containers automatically maintain the conversational state of a stateful session bean instance when it is passivated, these notifications are not needed for most session beans. Their purpose is to allow stateful session beans to maintain those open resources that need to be closed prior to an instance's passivation and then reopened during an instance's activation.
The `PrePassivate` and `PostActivate` lifecycle callback interceptor methods execute in an unspecified transaction and security context.

5. Given a list of methods of a Stateful or Stateless Session Bean class, define which of the following operations can be performed from each of those methods: SessionContext interface methods, UserTransaction methods, access to the java:comp/env environment naming context, resource manager access, and other enterprise bean access.

Table 1 defines the methods of a stateful session bean class from which the session bean instances can access the methods of the `javax.ejb.SessionContext` interface, the `java:comp/env` environment naming context, resource managers, `Timer` methods, the `EntityManager` and `Entity-ManagerFactory` methods, and other enterprise beans.

If a session bean instance attempts to invoke a method of the `SessionContext` interface, and that access is not allowed in Table 1, the container must throw the `java.lang.IllegalStateException.`

If a session bean instance attempts to access a resource manager, an enterprise bean, an entity manager or entity manager factory, and that access is not allowed in Table 1, the behavior is undefined by the EJB architecture.

If a session bean instance attempts to invoke a method of the `Timer` interface and the access is not allowed in Table 1, the container must throw the `java.lang.IllegalStateException`.

Bean method	Bean method can perform the following operations	
	Container-managed transaction demarcation	**Bean-managed transaction demarcation**
constructor	-	-
dependency injection methods (e.g., setSessionContext)	SessionContext methods: *getEJBHome, getEJBLocalHome, lookup* JNDI access to java:comp/env	SessionContext methods: *getEJBHome, getEJBLocalHome, lookup* JNDI access to java:comp/env
PostConstruct, PreDestroy, PrePassivate, PostActivate lifecycle callback interceptor methods	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, getUserTransaction, lookup* UserTransaction methods JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access
business method from business interface or from component interface; business method interceptor method	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, getRollbackOnly, isCallerInRole, setRollbackOnly, getEJBObject, getEJBLocalObject, getInvokedBusinessInterface, lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access Timer methods	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, getInvokedBusinessInterface, getUserTransaction, lookup* UserTransaction methods JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access Timer methods

Bean method	Bean method can perform the following operations	
	Container-managed transaction demarcation	**Bean-managed transaction demarcation**
afterBegin beforeCompletion	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, getRollbackOnly, isCallerInRole, setRollbackOnly, getEJBObject, getEJBLocalObject, lookup* JNDI access to java:comp/env Resource manager access Enterprise bean access EntityManagerFactory access EntityManager access Timer methods	N/A (a bean with bean-managed transaction demarcation cannot implement the SessionSynchronization interface)
afterCompletion	SessionContext methods: *getBusinessObject, getEJBHome, getEJBLocalHome, getCallerPrincipal, isCallerInRole, getEJBObject, getEJBLocalObject, lookup* JNDI access to java:comp/env	

- The `PostConstruct`, `PreDestroy`, `PrePassivate`, `PostActivate`, `Init`, and/or `ejbCreate<METHOD>`, `ejbRemove`, `ejbPassivate`, and `ejbActivate` methods of a session bean with container-managed transaction demarcation execute with an unspecified transaction context. Refer to Subsection 13.6.5 for how the container executes methods with an unspecified transaction context.

Additional restrictions:

- The `getRollbackOnly` and `setRollbackOnly` methods of the `SessionContext` interface should be used only in the session bean methods that execute in the context of a transaction. The container must throw the `java.lang.IllegalStateException` if themethods are invoked while the instance is not associated with a transaction.

The reasons for disallowing the operations in Table 1 follow:

- Invoking the `getBusinessObject` method is disallowed if the session bean does not define an EJB 3.0 business interface.
- Invoking the `getInvokedBusinessInterface` method is disallowed if the session bean does not define an EJB 3.0 business interface or was not invoked through a business interface.
- Invoking the `getEJBObject` and `getEJBHome` methods is disallowed if the session bean does not define a remote client view.
- Invoking the `getEJBLocalObject` and `getEJBLocalHome` methods is disallowed if the session bean does not define a local client view.
- Invoking the `getRollbackOnly` and `setRollbackOnly` methods is disallowed in the session bean methods for which the container does not have a meaningful transaction context, and to all session beans with bean-managed transaction demarcation.
- Accessing resource managers and enterprise beans is disallowed in the session bean methods for which the container does not have a meaningful transaction context and/or client security context.
- The `UserTransaction` interface is unavailable to enterprise beans with container-managed transaction demarcation.
- The `TimerService` interface is unavailable to stateful session beans.
- Invoking the `getMessageContext` method is disallowed for stateful session beans.
- Invoking the `getEJBObject` and `getEJBLocalObject` methods is disallowed in the session bean methods in which there is no session object identity established for the instance.

6. Identify correct and incorrect statements or examples about implementing a session bean as a web service endpoint, including rules for writing a web service endpoint interface and use of the @WebService and @WebMethod annotations.

7. Identify correct and incorrect statements or examples about the client view of a session bean, including the client view of a session object's life cycle, obtaining and using a session object, and session object identity.

Obtaining a Session Bean's Business Interface

A client can obtain a session bean's business interface through dependency injection or lookup in the JNDI namespace.
For example, the business interface `Cart` for the `CartBean` session bean may be obtained using dependency injection as follows:

```
@EJB Cart cart;
```

The `Cart` business interface could also be looked up using JNDI as shown in the following code segment using the `lookup` method provided by the EJBContext interface. In this example, a reference to the client bean's SessionContext object is obtained through dependency injection:

```
@Resource SessionContext ctx;
...
Cart cart = (Cart)ctx.lookup("cart");
```

Client View of Session Object's Life Cycle

From the point of view of the client, a session object exists once the client has obtained a reference to its business interface—whether through dependency injection or from lookup of the business interface in JNDI.

A client that has a reference to a session object's business interface can then invoke business methods on the interface and/or pass the reference as a parameter or return value of a business interface method.[3]

A client may remove a stateful session bean by invoking a method of its business interface designated as a `Remove` method.

The lifecycle of a stateless session bean does not require that it be removed by the client. Removal of a stateless session bean instance is performed by the container, transparently to the client.

Example of Obtaining and Using a Session Object

In this example, the client obtains a reference to the Cart's business interface through dependency injection. The client then uses the business interface to initialize the session object and add a few items to it.
The `startShopping` method is a business method that is provided for the initialization of the session object.

```
@EJB Cart cart;
...
cart.startShopping();
cart.addItem(66);
cart.addItem(22);
```

Finally the client purchases the contents of the shopping cart, and finishes the shopping activity.[4]

```
cart.purchase();
```

```
cart.finishShopping();
```

Stateless Session Beans Identity

```
@EJB Cart cart1;
@EJB Cart cart2;
...
if (cart1.equals(cart1)) { // this test must return true
...
}
...
if (cart1.equals(cart2)) { // this test must also return true
...
}
```

Stateful Session Beans Identity

```
@EJB Cart cart1;
@EJB Cart cart2;
...
if (cart1.equals(cart1)) { // this test must return true
...
}
...
if (cart1.equals(cart2)) { // this test must return false
...
}
```

Section 4: EJB 3.0 Message-Driven Bean Component Contract

1. Develop code that implements a Message-Driven Bean Class.

```
package examples.messaging;
import javax.jms.*;
import javax.ejb.*;
import javax.annotation.*;
@MessageDriven(activationConfig = {
@ActivationConfigProperty(propertyName = “destinationType”,
propertyValue = “javax.jms.Topic”)
})
public class LogBean implements MessageListener {
	public LogBean() {
		System.out.println(“LogBean created”);
	}
	public void onMessage(Message msg) {
		if (msg instanceof TextMessage) {
			TextMessage tm = (TextMessage) msg;
			try {
				String text = tm.getText();
				System.out.println(“Received new message : “ + text);
			} catch (JMSException e) {
				e.printStackTrace();
			}
		}
	}
	@PreDestroy
	public void remove() {
		System.out.println(“LogBean destroyed.”);
	}
}
```

2. **Identify correct and incorrect statements or examples about the interface(s) and methods a JMS Message-Driven bean must implement, and the required metadata.**

   - Uma classe message-driven bean deve ter a anotação @MessageDriven ou então deve ter esta informação denotada no deployment descriptor.
   - Uma classe message-driven bean deve implementar a message listener interface apropriada para o tipo de mensagem que o message-driven bean suporta, ou especificar a message listener interface usando a anotação @MessageDriven ou ainda usando o elemento do deployment descriptor messaging-type.
   - Uma classe message-driven bean NÃO precisa implementar a interface javax.ejb.MessageDrivenBean.

3. **Describe the use and behavior of a JMS message driven bean, including concurrency of message processing, message redelivery, and message acknowledgement.**

   Concurrency: Um conteiner permite a execução concorrente de muitas instâncias de uma message-driven bean class, permitindo assim o processamento concorrente da pilha de mensagens. Não há nenhuma garantia da ordem com que as mensagens são entregues as instâncias. Message-driven beans devem estar preparados para tratar messagem fora de sequência (cancelar uma reserva antes da solicitaçao da reserva ser feita).

   Redelivery: ???

   Acknowledgement:

   - Se o message-driven usa demarcação de transação gerenciada pelo container a confirmação da mensagem é tratada automaticamente como parte do commit da transação.
   - Se o message-driven usa demarcação de transação gerenciada pelo bean a "message receipt" não pode fazer parte da transação gerenciada pelo bean, e, nesse caso, a "receipt" é confirmada pelo container.
   - Se é usada a demarcação de transação gerenciada pelo bean , o Bean provider pode indicar se o modo de confirmação deve ser AUTO_ACKNOWLEDGE ou DUPS_ OK_ ACKNOWLEDGE usando o elemento activationConfig da anotação @MessageDriven ou usando o elemento activation-config-property do deployment descriptor, a property name usada para atribuir o modo de confirmação é acknowledgeMode .

4. **Identify correct and incorrect statements or examples about the client view of a message driven bean.**

- Da perspectiva de um cliente, a existência de um message-driven bean fica completamente oculta atrás do destination ou end-point do qual cada message-driven bean é listener.

Section 5: Java Persistence API Entities

1. Identify correct and incorrect statements or examples about the characteristics of Java Persistence entities.

- A API de persistência pode ser utilizada tanto em ambientes J2EE e J2SE
- Uma entidade é um lightweight persistent domain object.
- Uma entity class pode utilizar classes auxiliares que servem como classes helper ou para representar o estado de uma entidade.
- A entity class DEVE ser anotada com @Entity denotada em um xml descriptor.
- A entity class DEVE ter um construtor padrão (sem argumentos) com o escopo de public ou protected. Essa mesma entidade pode ter outros construtores também.
- A entity class DEVE ser uma top-level class. Uma enum ou interface não deve ser designada como uma entity.
- A entity class NÃO PODE ser final. Nenhum método ou qualquer variavel persistente de instância podem ser final.
- Se uma instância da entidade for passada por valor como um detached object (isto é, através de uma interface remota), a entity class DEVE implementar a interface Serializable.
- Entities suportam herança, associações polimórficas, e queries polimórficas.
- Tanto classes concretas como abstratas podem ser utilizadas como entities. Entities podem extender (herdar) de classes non-entity assim como classes entity, e non-entity classes podem extender de entity classes.
- O estado persistente de uma entity é representado por variáveis de instância, no qual podem corresponder a Java-Beans properties. Uma variável de instância pode ser diretamente acessada apenas de dentro dos métodos da instância da própria entity. Variáveis de instância não devem ser accessados por clientes de uma entity. O estado de uma entity está disponivel aos clientes somente pelos metodos get e set) ou por outros métodos de negócio. Variaveis de instância DEVEM ser private, protected, ou com visibilidade de pacote (nenhum modificador).

2. Develop code to create valid entity classes, including the use of fields and properties, admissible types, and embeddable classes.

- Anotações não são aplicadas a fields ou properties que estão com o modificador “transient” ou com a annotation @Transient..
- Quando properties são utilizados deve seguir o padrão dos JavaBeans (criação dos pares get/set)

- Quando collections são utilizadas DEVE-SE utilizar uma das seguintes interfaces java.util.Collection, java.util.Set, java.util.List, java.util.Map.

- Os persistent fields ou properties de uma entidade pode ser um dos seguintes tipos: tipos primitivos Java (int, double, float, etc); java.lang.String; outros tipos serializaveis Java (incluindo wrappers dos tipos primitivos, java.math.BigInteger, java.math.BigDecimal, java.util.Date,
- java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp,
- tipos serializaveis definidos pelo usuário, byte[], Byte[], char[], and Character[]); enums; entity types e/oo collections de entity types; e embeddable classes.

Exemplo :

```
@Entity
public class Customer implements Serializable {
private Long id;
private String name;
private Address address;
private Collection<Order> orders = new HashSet();
private Set<PhoneNumber> phones = new HashSet();
// No-arg constructor
public Customer() {}
@Id // property access is used
public Long getId() {
return id;
}
public void setId(Long id) {
this.id = id;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public Address getAddress() {
return address;
}
public void setAddress(Address address) {
this.address = address;
}
@OneToMany
public Collection<Order> getOrders() {
return orders;
}
public void setOrders(Collection<Order> orders) {
this.orders = orders;
}
@ManyToMany
public Set<PhoneNumber> getPhones() {
return phones;
}
public void setPhones(Set<PhoneNumber> phones) {
this.phones = phones;
}

// Business method to add a phone number to the customer
public void addPhone(PhoneNumber phone) {
this.getPhones().add(phone);
```

```
// Update the phone entity instance to refer to this customer
phone.addCustomer(this);
}
}
```

3. Identify correct and incorrect statements or examples about primary keys and entity identity, including the use of compound primary keys.

- Toda entity DEVE possuir uma chave primária.
- A chave primária deve ser definida na entity que é a raiz da hierarquia ou em uma mapped
- superclass hierarquia. A chave primária DEVE ser definida uma única vez na hierarquia de entidades.
- Uma chave primária simples (non-composite) DEVE corresponder a um unico persistent field ou property da classe entity. A annotation @Id é usada para denotar uma chave primária simples.
- Uma chave primária composta DEVE corresponder ou a um único persistent field ou property or para a conjunto de tais fields ou properties. Uma chave primária deve ser definida para representar uma chave primária composta. Chave primárias compostas tipicamente são utilizados quando há o mapeamento a partir de banco de dados legados quando a chave primária da tabela é constituída de multiplas colunas. As annotations @EmbeddedId e @IdClass são utilizadas para denotar chaves primárias compostas.

    - A chave primária (ou field ou uma property de uma chave primária composta) deve ser um dos seguintes tipos : qualquer tipo primitivo; any primitive wrapper type; java.lang.String; java.util.Date; java.sql.Date. Não é recomendável utilizar tipos de ponto flutuante em chaves primárias. Entities cujas chaves primárias fazem uso desses tipos de dados não serão portáveis.
    - Se a chave primária é gerada, apenas tipos inteiros serão portáveis. Se java.util.Date é utilizado como chave primária,o tipo temporal deve ser especificado como DATE.

As seguintes regras se aplicam para chaves primárias compostas :

- A classe da chave primária DEVE ser public e DEVE possuir um construtor public com nenhum argumento..
- Se o acesso baseado em property é utilizad, as properties da chave primária DEVE ser public ou protected.
- A classe da chave primária DEVE ser serializable.
- A classe da chave primária DEVE definir os métodos equals e hashCode. As semanticas de equalidade de valor para esses métodos DEVE ser consistente com a equalidade do banco de dados para os tipos em que são mapeados.

- Uma chave primária composta DEVE ser ou representada e mapeada como uma embeddable class (EmbeddedId Annotation) ou DEVE ser representada e mapeada para múltiplos fields ou properties da classe Entity (IdClass Annotation).
- Se a classe da chave primária composta é mapeada para multiplos fields ou properties da classe entity,
- Os nomes dos campos da chave primária ou properties na classed a chave primária e aqueles da classe entity DEVE corresponde e seus tipos DEVEM ser o mesmo.
- A aplicação NÃO DEVE MUDAR o valor da chave primária. O comportamento é indefinido se isso ocorrer.

4. Implement association relationships using persistence entities, including the following associations: bidirectional for @OneToOne, @ManyToOne, @OneToMany, and @ManyToMany; unidirectional for @OneToOne, @ManyToOne, @OneToMany, and @ManyToMany.

@OneToOne bidirecional

```
@Entity
public class Employee {
private Cubicle assignedCubicle;
@OneToOne
public Cubicle getAssignedCubicle() {
return assignedCubicle;
}
public void setAssignedCubicle(Cubicle cubicle) {
this.assignedCubicle = cubicle;
}
...
}
@Entity
public class Cubicle {
private Employee residentEmployee;
@OneToOne(mappedBy="assignedCubicle")
public Employee getResidentEmployee() {
return residentEmployee;
}
public void setResidentEmployee(Employee employee) {
this.residentEmployee = employee;
}
...
}
```

- Entity Employee referencia a uma unica instancia da Entity Cubicle.
- Entity Cubicle referencia a uma unica instancia da Entity Employee.
- Entity Employee é o mandante (ou o dono) do relacionamento.

Os seguintes mapeamentos padrões se aplicam :

- Entity Employee é mapeado para a tabela EMPLOYEE.
- Entity Cubicle é mapeado para a tabela CUBICLE.
- A tabela EMPLOYEE contem a chave estrangeira para a tabela CUBICLE. A coluna da chave estrangeira é nomeada ASSIGNEDCUBICLE_<PK of

CUBICLE>, onde <PK of CUBICLE> denota o nome da coluna da chave primária da tabela CUBICLE. A coluna da chave estrangeira possui o mesmo tipo da chave primária de CUBICLE, e há uma unique key constraint nela.

@ManyToOne e @OneToMany bidirecional

Assumindo que:

- Entity A referencia a uma unica instancia da Entity B.
- Entity B referencia uma collection da Entity A.
- Entity A DEVE ser é o mandante (ou o dono) do relacionamento.

Os seguintes mapeamentos padrões se aplicam :

- Entity A é mapeada para tabela A.
- Entity B é mapeada para tabela B.
- Tabela A contem a chave estrangeira para a tabela B. O nome da coluna da chave estrangeira é formada na concatenção dos seguintes : o nome da propriedade do relacionamento ou field da entity A; "_"; o nome da coluna da chave primária na tabela B. A coluna da chave estrangeira tem o mesmo tipo como o da chave primária da tabela B.

```
@Entity
public class Employee {
private Department department;
@ManyToOne
public Department getDepartment() {
return department;
}
public void setDepartment(Department department) {
this.department = department;
}
...
}
@Entity
public class Department {
private Collection<Employee> employees = new HashSet();
@OneToMany(mappedBy="department")
public Collection<Employee> getEmployees() {
return employees;
}
public void setEmployees(Collection<Employee> employees) {
this.employees = employees;
}
...
}
```

Nesse exemplo:

- Entity Employee referencia a uma unica instancia da Entity Department.
- Entity Department referencia a uma collection da Entity Employee.
- Entity Employee é o mandante do relacionamento.
- Os seguintes mapeamentos padrões se aplicam:
- Entity Employee é mapeado para a tabela EMPLOYEE.

- Entity Department é mapeada para a tabela DEPARTMENT.

A tabela EMPLOYEE contem a chave estrangeira para a tabela DEPARTMENT. A coluna da chave estrangeira é nomeado DEPARTMENT_<PK of DEPARTMENT>, onde <PK of DEPARTMENT> denota o nome da coluna da chave primária da tabela DEPARTMENT. A coluna da chave estrangeira tem o mesmo tipo como o da chave primária da tabela DEPARTMENT.

@ManyToMany bidirecional

Assumindo que:

- Entity A referencia a collection da Entity B.
- Entity B referencia a collection da Entity A.
- Entity A é o mandante do relationship.
- Os seguintes mapeamentos padrões se aplicam :
- Entity A é mapeada para a tabela A.
- Entity B is mapeada para a tabela B.
- Há uma join table que é nomeada A_B (nome do mandante primeiro). Esta join table possui duas colunas de chaves estrangeiras. Uma coluna da chave estrangeira refere-se a tabela A e tem o mesmo tipo como o da chave primária da tabela A. O nome dessa coluna da chave estrangeira é formada pela concatenação dos seguintes:
- O nome do relacionamento (property ou field) da entity B; "_"; o nome da coluna da chave primária na tabela A. A outra coluna da chave estrangeira refere-se a tabela B e tem o mesmo tipo assim como o tipo da chave primária da tabela B. O nome da coluna da chave estrangeira á formada pela concatenação dos seguintes: o nome do relacionamento (property ou field) da entity A; "_"; o nome da coluna da chave primária na tabela B.

```
@Entity
public class Project {
private Collection<Employee> employees;
@ManyToMany
public Collection<Employee> getEmployees() {
return employees;
}
public void setEmployees(Collection<Employee> employees) {
this.employees = employees;
}
...
}
@Entity
public class Employee {
private Collection<Project> projects;
@ManyToMany(mappedBy="employees")
public Collection<Project> getProjects() {
return projects;
}
public void setProjects(Collection<Project> projects) {
this.projects = projects;
```

```
}
...
}
```

Neste exemplo:

- Entity Project referencia a collection da Entity Employee.
- Entity Employee referencia a collection da Entity Project.
- Entity Project é o mandante do relacionamento.
- Os seguintes mapeamentos padrões se aplicam:
- Entity Project é mapeada para a tabela PROJECT.
- Entity Employee é mapeada para a tabela EMPLOYEE.
- Há uma join table nomeada PROJECT_EMPLOYEE (nome do mandante primeiro). Esta join table
- possui duas colunas de chave estrangeira. Uma delas refere-se a tabela PROJECT e possui o mesmo tipo assim como a chave primária de PROJECT. O nome da coluna desta chave estrangeira é PROJECTS_<PK of PROJECT>, onde <PK of PROJECT> denota o nome da coluna da chave primária da tabela PROJECT. A outra coluna da chave estrangeira refere-se a tabela EMPLOYEE e tem o mesmo tipo igual a chave primária de EMPLOYEE. O nome desta coluna da chave estrangeira é
- EMPLOYEES_<PK of EMPLOYEE>, onde <PK of EMPLOYEE> denota o nome da coluna da chave primária da tabela EMPLOYEE.

@OneToOne unidirecional

Os seguintes mapeamentos padrões se aplicam:

- Entity A é mapeada para tabela A.
- Entity B é mapeada para a tabela `B`.
- Tabela `A contem a chave estrangeira para a tabela B`. A coluna da chave estrangeira é formada pela contatenação dos seguintes : o nome do relacionamento (property ou field) da entity A; "_"; o nome da coluna da chave primária na tabela `B`. A coluna da chave estrangeira possui o mesmo tipo da chave primária da tabela B e há uma unique key constraint nela.

Examplo:

```
@Entity
public class Employee {
private TravelProfile profile;
@OneToOne
public TravelProfile getProfile() {
return profile;
}
public void setProfile(TravelProfile profile) {
this.profile = profile;
}
```

```
...
}
@Entity
public class TravelProfile {
...
}
```

Neste exemplo :
Entity `Employee referencia a uma única instancia da` Entity `TravelProfile`.

Entity `TravelProfile não possui referencia para a` Entity `Employee`.
Entity `Employee é o mandante` do relacionamento.

Os seguintes mapeamentos padrões se aplicam:

- Entity `Employee é mapeada para a tabela` `EMPLOYEE`.
- Entity `TravelProfile é mapeada para a tabela` `TRAVELPROFILE`.
- Tabela `EMPLOYEE contem a chave estrangeira para a tabela` `TRAVELPROFILE`. A coluna da chave estrangeira é nomeada `PROFILE_`<PK of TRAVELPROFILE>, onde <PK of TRAVELPROFILE> denota o nome da coluna da chave primária da tabela `TRAVELPROFILE`. A coluna da chave estrangeira possui o mesmo tipo da chave primária de `TRAVELPROFILE`, e há uma unique key constraint nela.

Unidirectional ManyToOne

Os seguintes mapeamentos padrões se aplicam:

- Entity A é mapeada para tabela A.
- Entity B é mapeada para a tabela `B`.
- Tabela A contem a chave estrangeira para a tabela B. A coluna da chave estrangeira é formada pela contatenação dos seguintes : o nome do relacionamento (property ou field) da entity A; "_"; o nome da coluna da chave primária na tabela B. A coluna da chave estrangeira possui o mesmo tipo da chave primária da tabela B .

Exemplo:

```
@Entity
public class Employee {
private Address address;
@ManyToOne
public Address getAddress() {
return address;
}
```

```
public void setAddress(Address address) {
this.address = address;
}
...
}
@Entity
public class Address {
...
}
```

Neste exemplo:

- Entity Employee referencia a uma única instancia da Entity Address.
- Entity `Address` não possui referencia para a Entity `Employee`.
- Entity `Employee` é o mandante do relacionamento.

Os seguintes mapeamentos padrões se aplicam:

- Entity Employee é mapeada para a tabela EMPLOYEE.
- Entity Address é mapeada para a tabela ADDRESS.
- Tabela EMPLOYEE contem a chave estrangeira para a tabela ADDRESS. A coluna da chave estrangeira é nomeada ADDRESS_<PK of ADDRESS>, onde <PK of ADDRESS> denota o nome da coluna da chave primária da tabela ADDRESS. A coluna da chave estrangeira possui o mesmo tipo da chave primária de ADDRESS.

@OneToMany Unidirecional

Os seguintes mapeamentos padrões se aplicam:

- Entity A é mapeada para tabela A.
- Entity B é mapeada para a tabela B.
- Há uma join table que é nomeada `A_B` (nome do mandante primeiro). Esta join table possui duas colunas de chaves estrangeiras. Uma delas referece-se a tabela `A` `e possui o mesmo tipo da chave primária da tabela A.` O nome dessa coluna da chave estrangeira é formada pela concatenação dos seguintes : o nome da entity A; "_"; o nome da coluna da chave primária na tabela `A`. A outra coluna da chave estrangeira refere-se a tabela `B` e possui o mesmo tipo da chave primária da tabela `B` e há uma unique key constraint nela. O nome da coluna dessa chave estrangeira é formada pela concatenação dos seguintes: o nome do relacionamento (property ou field) da entity A; "_"; o nome da coluna da chave primária na tabela B.

Exemplo:

```
@Entity
```

```
public class Employee {
private Collection<AnnualReview> annualReviews;
@OneToMany
public Collection<AnnualReview> getAnnualReviews() {
return annualReviews;
}
public void setAnnualReviews(Collection<AnnualReview>
annualReviews)
{
this.annualReviews = annualReviews;
}
...
}
@Entity
public class AnnualReview {
...
}
```

Neste exemplo :

- Entity `Employee referencia a uma` collection of Entity `AnnualReview`.
- Entity `AnnualReview` não se referencia a Entity `Employee`.
- Entity `Employee` é o mandante do relacionamento.

Os seguintes mapeamentos padrões se aplicam:

- Entity `Employee é mapeada para a tabela` `EMPLOYEE`.
- Entity `AnnualReview é mapeada para a tabela` `ANNUALREVIEW`.
- Há uma join table `EMPLOYEE_ANNUALREVIEW` (nome do mandante primeiro). Esta join table possui duas colunas de chave estrangeira. Uma coluna de chave estrangeira refere-se a tabela `EMPLOYEE`
- e tem o mesmo tipo da chave primária de `EMPLOYEE`. Esta coluna de chave estrangeira é nomeada `EMPLOYEE_`<PK of EMPLOYEE>, onde <PK of EMPLOYEE> denota o nome da coluna da chave primária da tabela `EMPLOYEE`. A outra coluna da chave estrangeira refere-se a tabela `ANNUALREVIEW`
- e possui o mesmo tipo da chave primária de `ANNUALREVIEW`. Esta coluna de chave estrangeira é nomeada `ANNUALREVIEWS_`<PK of ANNUALREVIEW>, ondee <PK of ANNUALREVIEW> denota o nome da coluna da chave primária da tabela `ANNUALREVIEW`. Há uma unique key constraint na chave estrangeira que refere-se a tabela `ANNUALREVIEW`.

@ManyToMany unidirecional

Os seguintes mapeamentos padrões se aplicam:

- Entity A é mapeada para tabela A.

- Entity B é mapeada para a tabela B.
- Há uma join table `A_B` (nome do mandante vem primeiro). Esta join table possui duas colunas de chaves estrangeiras. Uma das colunas refere-se a tabela `A e tem o mesmo` tipo da da chave primária da tabela A. O nome desta coluna de chave estrangeira é formada pela concatenação dos seguintes : o nome da entity `A`; "_"; o nome da coluna da chave primária na tabela `A`. A outra coluna de chave estrangeira refere-se a tabela `B e tem o mesmo tipo da chave primária da tabla B`. O nome da coluna da chave estrangeira é formada pela concatenação dos seguintes : o nome do relacionamento (property ou field) da entity `A`; "_"; o nome da coluna da chave primária da tabela `B`.

```
@Entity
public class Employee {
private Collection<Patent> patents;
@ManyToMany
public Collection<Patent> getPatents() {
return patents;
}
public void setPatents(Collection<Patent> patents) {
this.patents = patents;
}
...
}
@Entity
public class Patent {
...
}
```

Neste exemplo:

- Entity `Employee` referencia a collection da Entity `Patent`.
- Entity `Patent não possui referencia para a` Entity `Employee`.
- Entity `Employee` é o mandante do relacionamento.

Os seguintes mapeamentos padrões se aplicam:

- Entity `Employee é mapeada` para a tabela `EMPLOYEE`.
- Entity `Patent é` mapeada para a tabela `PATENT`.
- Há uma join table `EMPLOYEE_PATENT` (nome do mandante primeiro). Esta join table
- possui duas colunas de chave estrangeira. Uma coluna de chave estrangeira refere-se a tabela `EMPLOYEE e tem o mesmo tipo da chave primária de EMPLOYEE`. Esta coluna de chave estrangeira é nomeada `EMPLOYEE_`<PK of EMPLOYEE>, onde <PK of EMPLOYEE> denota o nome da coluna da chave primária da tabela `EMPLOYEE`. A outra coluna de chave estrangeira refere-se a tabela `PATENT`

- e tem o mesmo tipo da chave primária de `PATENT`. Esta coluna de chave estrangeira é nomeada `PATENTS_`<PK of PATENT>, onde <PK of PATENT> denota o nome da coluna da chave primária da tabela `PATENT`.

5. Given a set of requirements and entity classes choose and implement an appropriate object-relational mapping for association relationships.

As possíveis associações foram descritos acima

6. Given a set of requirements and entity classes, choose and implement an appropriate inheritance hierarchy strategy and/or an appropriate mapping strategy.

O mapeamento da hierarquia de classes é especificado através de metadados.

Existem 3 estratégias básicas que são usadas para mapear uma classe ou uma hierarquia de classes para um banco de dados relacional:

- a single table per class hierarchy (um único "tabelão" para toda a hierarquia de classes).
- a table per concrete entity class (uma tabela para cada classe concreta)
- Uma estratégia no qual os campos que são especificados para subclasses são mapeados para uma tabela separada (essa tabela filha possui relacionamento para a tabela pai) e um join é executado para instanciar a subclasse).

Uma implementação é requerida para suportar uma única tabela por hierarquia de classes (single table per class hierarchy inheritance mapping strategy) e a joined subclass strategy.

Suporte para table per concrete class inheritance mapping strategy é opcional nesse release.

Support para a combinação de estratégias de herança dentro de uma única hierarquia de herança de entities não é requerida nessa especificação.

Single Table per Class Hierarchy Strategy

Nessa estratégia, todas as classes na hierarquia são mapeadas para uma única tabela. A tabela possui uma coluna que serve como um "discriminator", isto é, a coluna cujo valor identifica a subclasse especifica.

Esta estratégia prove bom suporte a relacionamentos polimorficos entre entities e para queries.
Porém, possui um inconveniente, que requer que as colunas que correspondem a uma estado especifico das subclasses sejam nullable.

Table per Concrete Class Strategy

In this mapping strategy, each class is mapped to a separate table. All properties of the class, including inherited properties, are mapped to columns of the table for the class.

This strategy has the following drawbacks:
- It provides poor support for polymorphic relationships.
- It typically requires that SQL UNION queries (or a separate SQL query per subclass) be issued for queries that are intended to range over the class hierarchy.

Joined Subclass Strategy

In the joined subclass strategy, the root of the class hierarchy is represented by a single table. Each subclass is represented by a separate table that contains those fields that are specific to the subclass (not inherited from its superclass), as well as the column(s) that represent its primary key. The primary key column(s) of the subclass table serves as a foreign key to the primary key of the superclass table.

This strategy provides support for polymorphic relationships between entities.

It has the drawback that it requires that one or more join operations be performed to instantiate instances of a subclass. In deep class hierarchies, this may lead to unacceptable performance. Queries that range over the class hierarchy likewise require joins.

7. Describe the use of annotations and XML mapping files, individually and in combination, for object-relational mapping.

Section 6: Java Persistence Entity Operations

1. Describe how to manage entities, including using the EntityManager API and the cascade option.

- Um EntityManager está ligado a um persistent context
- Um persistent context é uma um grupo de instâncias de entities que para cada identidade existe uma única instância daquele entity.
- O grupo de entities que poderem ser gerenciadas por um EntityManager formam uma entity unit (relacionadas e agrupadas pela aplicação e mapeadas por m único banco de dados) .
- Os métodos se aplicam as entidades passadas como parâmetro e as ligadas a eles pelo atributo CASCADE da annotation de relacionamento.
- Segue os principais métodos da interface EntityManager:

```
public interface EntityManager{
   public void persist(Object entity);//Faz o insert
   public <T> T merge(T entity);//Faz o update
   public void remove(Object entity);//Faz o delete
   public <T> T find(Class<T>,Object pk);//Busca por PK
   public <T> T getReference(Class<T>,Object pk);Busca lazy
   public void flush();//Faz alterações no BD
   public void lock(Object, LockModeType);//Obtém o lock
   public void refresh(Object);//Busca estado do BD
   public void clear();//Todas Managed -> Deatached
   public boolean contains(Object);//Pertence ao contexto?
   //métodos createQuery na sessão de query language
   public void joinTransaction();//Une a transação em um contexto JTA
   public Object getDelegate();//Impemntation Specific
   public void close();//Fecha o EntityManager
   public boolean isOpen();//Vê se está aberto
   public EntityTransaction getTransaction();//Recupera transação
}
```

- Os métodos persist, merge, remove e refresh precisam estar no contexto de uma transação.
- Os métodos find, getReference não precisam de contexto de transação.
- Os objetos Query e EntityTransaction são válidos somente enquanto o EntityManager está aberto (IllegalStateException)
- Runtime exceptions fazem a transação fazer rollback
- Os métodos close, isOpen, getTransaction e joinTransactrion só podem ser utilizados em Application-Managed Entity Managers.

2. Identify correct and incorrect statements or examples about entity instance lifecycle, including the new, managed, detached, and removed states.

Os estados possíveis de um entity são:

- New: sem persistent identity e não associado com o persistent context.
- Managed: tem persistent identity e está associado a um persistent context.
- Detached: tem persistent identity e não está associado a um persistent context.
- Removed: tem persistent identity, está associado a um persistent context e está marcado para ser removido.

Resultados de operações nos entities:

- persist()
  - New – vira managed e é inserida no banco (no flush() ou commit()).
  - Managed – ignora o comando, mas os relacionamentos com cascade são afetados.
  - Detached – EntityExistsException ao chamar o método ou no flush ou no commit.
  - Removed – vira managed

- remove()
  - New – ignora, mas os relacionamentos com cascade são afetados.
  - Managed – remove do banco de dados (no flush ou commit) e passa a ser removed. Os cascade são afetados.
  - Detached – IllegalArgumentException.
  - Removed – ignora
- flush()
  - Managed – Sincronizada com o banco
  - Para relacionamentos com cascade – Sincronizado com BD
  - Para relacionamentos sem cascade – Sem é New ou Removed é lançado um IlegalStateException e se é Detached é sincronizado se o objeto principal é o dono do relacionamento.
  - Removed – É removida do BD (sem cascade).
- merge()
  - Detached – o estado é copiado para uma instância Managed pré-existente com a mesma identidade ou é criada uma instância Managed com a mesma identidade.
  - New – é criada uma nova instância Managed em que o estado é copiado para ela.
  - Removed – IllegalArgumentException
  - Managed – ignora, mas é propagado para os cascade.
  - Se X referencia Y e possui cascade, Y é feito o merge para Y' e em X' é setada uma referência para Y'.
  - Se X referencia Y e não possui cascade, não é feito o merge de Y e em X' é setada uma referência para uma instância Y' managed com a mesma identidade de Y.
- contains()
  - Managed – true
  - New – false se não foi chamado persist() e true se foi chamado persist().
  - Detached – false
  - Removed – false
  - Para efeito do método contains() as operações tem efeito imediato, por mais que elas ainda não tenham tido efeito.

3. Identify correct and incorrect statements or examples about EntityManager operations for managing an instance's state, including eager/lazy fetching, handling detached entities, and merging detached entities.

- Coisas que podem fazer uma instância ser detached
  - Commit
  - Rollback
  - Usar o método clear()
  - Chamar o método close() no EntityManager
  - Serializar um entity
  - Passar uma entidade por valor

- O acesso a campos de uma instância detached é disponível quando:
  - O campo não está marcado com fetch=LAZY
  - O campo foi acessado antes de se tornar detached
- O acesso a campos de relacionamento de uma instância detached é disponível quando:
  - A entidade localizada usando find()
  - Uma entidade recuperada utilizando explicitamente uma query ou uma cláusula FETCH JOIN
  - Uma entidade em que foi acessado uma propriedade com non-primary-key persistent state.
  - Uma variável com associação marcada com fecth=EAGER.

4. Identify correct and incorrect statements or examples about Entity Listeners and Callback Methods, including: @PrePersist, @PostPersist, @PreRemove, @PostRemove, @PreUpdate, @PostUpdate, and @PostLoad, and when they are invoked.

- Default entity listeners são definidos no deployment descritor e valem para todos os entitys (são os primeiros a serem executados)
- A annotation @EntityListeners é usada para definir os listeners da classe.
- A ordem pode ser redefinida no deployment descriptor.
- Uma única classe não pode ter mais de um nétodo para o memso callback.
- Os listeners são stateless e devem possuir um construtor sem argumentos.
- Os callbacks podem lançar runtime exceptions (causam rollback), invocar JNDI, JMS JDBC, EJBs e etc, mas para serem portáveis não devem mexer com EntityManager.
- Um método callback possui a seguinte assinatura:

```
void <METHOD>() -> No próprio entity
void <METHOD>(Object) -> Em uma classe separada
```

- Os métodos podem ter qualquer modificador de acesso mas não podem ser static ou final.
- Invocação dos callbacks:
  - PrePersist – Antes de um merge (que criar um insert) ou persist (chamado para os cascades).
  - PreRemove – Antes do remove (chamado para os cascades)
  - PostPersist – Depois de um merge (que criar um insert) ou persist (chamado para os cascades). As primary-key geradas estão disponíveis neste método.
  - PostRemove – Depois do remove (chamado para os cascades)
  - PostUpdate e PreUpdate – Antes e depois de um objeto ser atualizado. É implementation dependent quantas vezes o método será chamado se ele for alterado mais de uma vez na mesma transação.

    - PostLoad – Depois de uma instância ser carregada ou chamado o método refresh(). Invocado antes do retorno de uma query e antes do acesso a uma associação.
- É implementation dependent se os callback serão invocados antes ou depois dos cascade.
- A annotation ExcludeDefaultListeners faz com que os listeners default não sejam chamados
- A annotation ExcludeSuperclassListeners faz com que os listeners da superclasse não sejam chamados
- Os métodos callback do próprio entity são chamados depois dos listeners, sempre os das superclasses primeiro.

5. Identify correct and incorrect statements about concurrency, including how it is managed through the use of @Version attributes and optimistic locking.

- A especificação prevê o uso de isolamento read-commited
- O optimistic locking é utilizado para garantir que não seja feito update concorrente em um objeto e as modificações sejam sobrepostas no banco.
- Para isto cria-se um campo com @Version que será gerenciado pelo container
- A aplicação não deve alterar atributos do tipo @Version (a não ser em bulk updates)
- O atributo @Version é incrementado em cada update e quando o valor não é consistente é lançado um OptimisticLockException
- O uso do método lock() só é garantido em entidades que tem um campo @Version (prevê dirty read e non repetable read)
- O uso do método lock() com LockModeType.WRITE faz um incremento na coluna @Version

Section 7: Persistence Units and Persistence Contexts

1. Identify correct and incorrect statements or examples about JTA and resource-local entity managers.

- Um JTA entity manager é gerenciado pelo container e deve ser obtido pela aplicação utilizando dependency injection ou por JNDI.
- Para utilizar o dependency injection deve-se usar a anotação @PersistenceContext(type = EXTENDED/TRANSACTION, unitName = “nome da persistent unit”)

```
@PersistentContext
EntityManager em;
```

- O tipo EXTENDED pode ser utilizando somente em Statefull (neste caso o contexto de persistência se mantém entre as chamadas de métodos e só é fechado no método @Remove pelo container).

- Para fazer o lookup, o @PersistentContext deve ser declarado:

```
@Stateless
@PersistentContext(unitName=”teste”)
public class Sessionbean implements BusinessInterface{
 @Resource SessionContext ctx;
   public void metodo(){
      EntityManager em =
(EntityManager)ctx.lookup(“teste”);
 }
}
```

- Um resource-local entity manager é obtido através de uma EntityManagerFactory.
- Uma entity manager factory pode ser obtida por dependendy injection ou pela classe javax.persistence.Persistence.
- Por injeção de dependência é utilizada a annotation @PersistenceUnit:

```
@PersistenceUnit
EntityMangerFactory emf;
```

- Em um ambiente Java SE, o Entity Manager Factory deve ser obtido pela classe javax.persistence.Persistence com o método createEntityMangerFactory():

```
EntityMangerFactory emf;
emf = Persistence.createEntityManagerFactory(“NOME”);
EntityManger em = emf.createEntityManger();
```

- A interface EntityManagerFactoy tem o seguintes métodos:

```
public interface EntityManagerFactory{
 public EntityManager createEntityManger();
 public EntityManager createEntityManger(Map map);
 public void close();
 public void isOpen();
}
```

- O mapa de parâmetros pode possuir diversos parâmetros específicos do fornecedor e propriedades não reconhecidas devem ser ignoradas.

2. Identify correct and incorrect statements or examples about container-managed persistence contexts.

Ver parte 1.

3. Identify correct and incorrect statements or examples about application-managed persistence contexts.

Ver parte 1.

4. Identify correct and incorrect statements or examples about transaction management for persistence contexts, including persistence context propagation, the use of the EntityManager.joinTransaction() method, and the EntityTransaction API.

- Uma aplicação que utiliza o EntityManager gerenciado pelo container tem as transações controladas por JTA.
- Uma aplicação que utiliza o EntityManager gerenciado pela própria aplicação pode utilizar a API EntityTransaction para controlar as transações ou JTA.
- Uma instância de EntityTransaction pode ser obtida pelo método EntityManager.getTransaction() e tem a seguinte interface:

```
public interface EntityTransaction{
 public void begin();
 public void commit();
 public void rollback();
 public void setRollbackOnly();
 public boolean getRollbackOnly();
 public boolean isActive();
}
```

- O método joinTransaction() deve ser chamado explicitamente para associar um entity manager com uma transação JTA.
- Um persistence context é propagado juntamente com a propagação da transação JTA (somente localmente e nunca para camadas remotas).
- Se o método executa sem transação ou a transação não é propagada, o persistence context não é propagado.
- Se a propagação acontecer de um contexto EXTENDED para um TRANSACTION, não ocorre a propagação, mas de um EXTENDED para outro pode ser propagado.
- Se uma transação JTA com um persistent context é propagada para um statefull com persistent context extended já existente, é lançada uma EJBException.

5. Identify correct and incorrect statements or examples about persistence units, how persistence units are packaged, and the use of the persistence.xml file.

- Uma Persistent Unit é composta por:
  - Entity manager factory e seus entity managers, com suas configurações.
  - As entidades gerenciadas incluídas na persistent unit
  - Mapping Metatada (annotations e/ou XML)
- Persistent Unit Packaging
  - O root de um persistent unit é o arquivo que possui o arquivo persistent.xml no diretório META-INF e pode ser um dos seguintes:

        - EJB-JAR
        - WEB-INF/classes em um WAR
        - Um jar em WEB-INF/lib
        - Um jar no root de um EAR
        - Um jar no diretório de library de um EAR
        - Um application client jar
    - Todo persistent unit tem um nome e não podem haver mais de um persistent unit com o mesmo nome em um EJB-JAR, WAR ou EAR.
- O arquivo persistence.xml

```
<persistence>
   <persistent-unit name="nome" transaction-type="JTA/RESOURCE_LOCAL">
      <provider>  Nome do persistence provider
      <jta-data-source> Fonte de dados JTA
      <non-jta-data-source> Fonte de dados não-JTA
      <mapping-file> arquivo de mapeamento
      <jar-file> jar para procura de classes
      <class> nome explícito de classe
      <exclude-unlisted-classes>
      <properties>
         <property name="nome" value="valor"/>
      </properties>
   </persistent-unit>
</persistence>
```

- Devem ser especificados pelo menos: um ou mais arquivos de mapeamento, um ou mais jars, uma lista explícita de classes, ou as classes anotadas estarem no root da persistent unit.

6. Identify correct and incorrect statements or examples about the effect of persistence exceptions on transactions and persistence contexts.

- Quando um entity manager lança uma runtime exception, a transação é marcada para rollback.
- Na interface EntityTransaction, a IllegalStateException é lançada se isActive() é true no método begin() e se isActive() é false nos outros métodos.

Section 8: Java Persistence Query Language

2. Develop queries that use the SELECT clause to determine query results, including the use of entity types, use of aggregates, and returning multiple values.

- A cláusula SELECT é obrigatória!
- O que estiver no SELECT determina o que será retornado

- O uso de OBJECT(), como "SELECT OBJECT(o) FROM Ordem o" é opcional
- DISTINCT faz com que resultados não sejam repetidos
- Cláusulas opcionais: WHERE, GROUP BY, HAVING, ORDER BY
- As funções agregadas são: COUNT, MAX, MIN, AVG, SUM

3. Develop queries that use Java Persistence Query Language syntax for defining the domain of a query using JOIN clauses, IN, and prefetching.

- Um inner join faz um produto carteziano utilizando um relacionamento. Pode-se usar JOIN, INNER JOIN e IN. As queries abaixo são similares:

```
SELECT c FROM Customer c JOIN c.orders o WHERE o.status = 1
SELECT c FROM Customer c INNER JOIN c.orders o WHERE o.status = 1
SELECT c FROM Customer c, IN(c.orders) o WHERE o.status = 1
```

- O LEFT JOIN ou LEFT OUTER JOIN faz com que a recuperação ocorra mesmo que o lado "direito" seja vazio.

```
SELECT c FROM Customer c LEFT JOIN c.orders o WHERE o.status = 1
```

- O uso de FETCH depois do JOIN faz com que os dados do relacionamento sejam também inicializados. Lembrar que se 1 departamento tem 5 empregados, serão retornadas 5 referências ao mesmo departamento (usar o DISTINCT para filtrar).

```
SELECT d FROM Departamento d LEFT JOIN FETCH d.empregados
```

4. Use the WHERE clause to restrict query results using conditional expressions, including the use of literals, path expressions, named and positional parameters, logical operators, the following expressions (and their NOT options): BETWEEN, IN, LIKE, NULL, EMPTY, MEMBER [OF], EXISTS, ALL, ANY, SOME, and functional expressions.

- Literais usam aspas simples
- As propriedades das entidades podem ser acessadas com o operador ".".
- Os named parameters utilizam dois pontos na frente *:parametro*
- Os positional parameters são colocados ?1, ?2, ?3 e etc… Eles são numerados iniciando de 1 e não precisam aparecer na ordem correta na query.
- BETWEEN pode ser usado com número, String e data.
- IN pode ser usado em uma subquery ou lista ('A', 'B', 'C')
- LIKE é igual ao SQL normal
- IS EMPTY pode ser usado com collections
- MEMBER OF verifica se um elemento é o item de uma coleção
- EXISTS vê se uma subquery apresenta algum resultado
- ALL / ANY / SOME é usado para fazer uma comparação como uma subquery. Exemplo: WHERE m.salario > ALL(SELECT …)

5. Develop Java Persistence Query Language statements that update a set of entities using UPDATE/SET and DELETE FROM.

- O contexto não é sincronizado
- O cascade não se aplica
- Passa direto pelas verificações de Optimistic Lock
- Não é garantido que o valor da propriedade @Version vai ser atualizado
- Exemplos:

DELETE FROM Customer c WHERE c.status = 'inativo'
UPDATE Customer c SET c.status='ativo' WHERE c.balance>1000

6. Declare and use named queries, dynamic queries, and SQL (native) queries.

- Uma named query é declarada da seguinte forma:

```
@NamedQuery(name="nome", query="query")
```

- Uma native named query é declarada da seguinte forma:

```
@NamedNativeQuery(name="nome", query="query")
```

- As anotações @NamedQueries e @NamedNativeQueries contém um array de queries.
- As funções da classe EntityManager para criar uma query são:
  - `createQuery(qlString)`
  - `createNamedQuery(name)`
  - `createNativeQuery(qlString)`
  - `createNativeQuery(qlString, Class resultClass)`
  - `createNativeQuery(qlString, String resultSetMapping)`

7. Obtain javax.persistence.Query objects and use the javax.persistence.Query API.

```
public interface Query{
  List getResultList(); // Recupera lista de resultados
  Object getSingleResult(); // Para selects q retornam só um resultado
  int executeUpdate(); // Retorna numero de entidades afetadas
  Query setMaxResults(int); // Usado para paginação
  Query setFirstResult(int); // Usado para paginação
  Query setParameter(String name, …) // Setar um named parameter
  Query setParameter(int position, …) // Setar um posicional parameter
```

```
  Query setFlushMode()
}
```

- O FlushMode pode ser COMMIT ou AUTO (default), sendo que o AUTO faz com que os updates fiquem visíveis no contexto da transação.

Section 9: Transactions

1. Identify correct and incorrect statements or examples about bean-managed transaction demarcation.

- Deve-se usar javax.transaction.UserTransaction para controlar as transações
- Uma transação inicia em UserTransaction.begin e termina em UserTransaction.commit
- A classe UserTransaction pode ser obtida por injeção de dependência ou pela classe EJBContext.getUserTransaction().
- O método UserTransaction.rollback() faz o rollback
- O método UserTransaction.getStatus() retorna o status da transação.
- A instância que iniciou a transação deve terminar a mesma antes de iniciar outra.
- Não podem ser usados os mecanismos de controle de transação específico de cada recurso (exemplo: EntityManager, Connection e etc...).
- Em um MDB ou Stateless Session Bean a transação deve ser finalizada antes do final do Business Method. Em um statefull ela pode ser mantida entre as chamadas de métodos.
- Os métodos getRollbackOnly() e setRollbackOnly() não deve ser utilizado.

2. Identify correct and incorrect statements or examples about container-managed transaction demarcation, and given a list of transaction behaviors, match them with the appropriate transaction attribute.

- Não pode usar os seguintes métodos:
  - java.sql.Connection: commit, setAutoCommit, rollback
  - java.jms.Session: commit, roolback
- Não pode obter ou usar uma instância de UserTransaction.
- Transaction Attributes : próxima sessão.

3. Identify correct and incorrect statements or examples about transaction propagation semantics.

- Os valores para a Annotation são todos em maiúscula (NOT_SUPOORTED) e os do deployment descriptor são como nomes de classes (NotSupported).
- Tipos de Transaction Attributes para os métodos:

    - Mandatory – O método deve ser chamado no contexto de uma transação anterior senão ocorre um erro.
    - Required (default) – Deve ser executado no contexto de uma transação. Se houver uma esta é utilizada e se não tiver uma nova é criada.
    - RequiresNew – Sempre inicia uma nova transação independente de haver uma ou não.
    - Supports – Roda no contexto de uma transação caso esta exista anteriormente, porém roda fora do contexto de transação caso esta não exista.
    - NotSupported – Nunca roda no contexto de uma transação, esta existindo ou não anteriormente.
    - Never – Só roda em contextos for a de transação e não pode ser chamado de um método com contexto transacional.
- Em um Bean Managed Transaction, o método nunca executa no contexto de uma transação anterior, sempre rodando sem transação ou em uma nova (controlada pelo Bean Provider).
- As transações de Bean Managed Transactionpodem ser propagadas normalmente para métodos com Container Managed Transaction.

4. Identify correct and incorrect statements or examples about specifying transaction information via annotations and/or deployment descriptors.

- Para definir o tipo de transação deve ser usada annotation TransactionManagement que pode ter os valores BEAN ou CONTAINER. Usada na classe do Bean.
- Também pode ser utilizado o atributo transaction-type (o default é CONTAINER).
- Para definir o transaction atribute deve ser usada a anotação TransactionAttribute (pode ser usado em um método ou em uma classe).
- Quando @TransactionAttribute for definido pela classe, vale para todos os business methods.
- O @TransactionAttribute definido para um método sobrepõe o valor definido para a classe.
- O @TransactionAttribute definido para a superclasse do EJB vale para todos os métodos definidos na superclasse.
- O método sobreposto na subclasse sobrepõe a definição feita na superclasse (inclusive se o método na subclasse não definir nada, ele é redefinido de acordo com a definição na subclasse – REQUIRED por default) – olhar exemplo da página 337.
- Para o DeploymentDescriptor:

```
<container-transaction>
   <method>
      <ejb-name>
      <method-name>
   </method>
   <trans-attribute> (é aqui que é definido o tipo de transação)
</container-transaction>
```

5. **Identify correct and incorrect statements or examples about the use of the EJB API for transaction management, including getRollbackOnly, setRollbackOnly and the SessionSynchronization interfaces.**

- Usando Container Managed Transactions o controle da transação acontece todo pela classe EJBContext (superclasse de SessionContext ou MessageDrivenContext).
- O método setRollbackOnly() marca a transação para roolback e getRollbackOnly() verifica se a transação está marcada para rollback.
- A interface SessionSynchoronization:
  - Só pode ser implementada por Statefull Session Beans com Container Managed Transaction.
  - Povê aos beans uma forma de ser avisado sobre notificações de eventos relativos as transações.
  - Possui os métodos: afterBegin, beforeCompletion e afterCompletion (o único sem transaction context).

Section 10: Exceptions

1. **Identify correct and incorrect statements or examples about exception handling in EJB.**

- Application Exceptions:
  - Para retratar exceções de negócio
  - Não devem ser usadas para erros de sistema.
  - Pode ser definida na business interface, home interface, component interface, message listener interface ou web service endpoint.
  - Pode ser uma subclasse de Exception (checked) ou uma subclasse de RuntimeException (desde que possua a anotação @ApplicationException)
  - Não pode ser subclasse de RemoteException
  - As exceptions legadas CreateException, RemoveException e FinderException são consideradas ApplicatonException.
  - Uma ApplicationException não marca a transação para rollback, a não ser que a anotação @ApplicationException possua o atributo rollback = true.
- System Exceptions:
  - Para reportar erros do sistema
  - Sempre marca a transação para rollback
  - É uma RemoteException (ou subclasses) ou uma RuntimeException não marcada com a anotação @ApplicationException.
  - EJBException é uma exemplo de System Exception
  - A instância do bean será descartada.

2. **Identify correct and incorrect statements or examples about application exceptions and system exceptions in session beans and message-driven beans, and defining a runtime exception as an application exception.**

- Explicado na sessão anterior...

3. **Given a list of responsibilities related to exceptions, identify those which are the bean provider's, and those which are the responsibility of the container provider. Be prepared to recognize responsibilities for which neither the bean nor container provider is responsible.**

- Bean Provider: Declarar as exceptions nos métodos; Tratar corretamente as exceptions; Definir as exceptions de sistema e de aplicação
- Container Provider: Definir o comportamento do container para que tenha o comportamento adequado quando uma exception form lançada.

4. **Identify correct and incorrect statements or examples about the client's view of exceptions received from an enterprise bean invocation.**

- Explicado na próxima sessão...

5. **Given a particular method condition, identify the following: whether an exception will be thrown, the type of exception thrown, the container's action, and the client's view.**

- Caso seja lançada uma exception de um método que possua Conatiner Managed Transaction:
    - Application Exception:
        - O container marca a transação para rollback caso o atributo rollback = true
        - O container lança a exception para o cliente
        - Se o cliente rodar na mesma transação e não for dado o rollback ele pode tratar a exception e continuar a lógica, e caso contrário continuar não faz sentido.
    - System Exception:
        - Faz o log da exception
        - Marca a transação para rollback (se houver uma)
        - Discarta a instânca do bean
        - Lança EJBTransactionRolledbackException (caso o cliente rode na mesma transação) ou EJBException (caso o cliente não rode na mesma transação – e neste caso o cliente pode tratar a exception e continuar se quiser) – ou RemoteException quando o cliente for remoto.
- Em um Bean Managed Transaction:
    - Uma application exception é lançada para o cliente
    - Uma system exception é logada pelo container, caso exista uma transação aberta ela é marcada para rollback e é lançado para o cliente uma EJBException.

- Os métodos de callback como PostConstruct e PreDestroy só podem lançar system exceptions e neste caso o container deve fazer o log e descartar a instância.
- Os métodos de timeout tem o mesmo comportamento, porém também é dado o rollback na transação (se houver).

Section 11: Security Management

1. Match security behaviors to declarative security specifications (default behavior, security roles, security role references, and method permissions).

Descrito nos outros tópicos.

2. From a list of responsibilities, identify which roles are responsible for which aspects of security: application assembler, bean provider, deployer, container provider, system administrator, or server provider.

- Bean Provider e/ou Application Assembler
  - Acesso programático ao contexto de segurança (só bean provider)
  - Declaração dos roles utilizados programaticamente (só bean provider)
  - Definição dos Security Roles
  - Definição das permissões dos métodos
  - Fazer o link entre os security roles references e os security roles
  - Especificar os security identities no deployment descriptor (colocar os run-as)
- Deployer
  - Security Domain e Principal Realm Assignment
  - Assignment of Security Roles
  - Principal Delegation
  - Security Management of Resource Access
  - General Notes on Deployment Descriptor Processing
- EJB Container Provider
  - Deployment Tools
  - Security Domains
  - Security Mechanisms
  - Passing Principals on EJB Calls
  - Security Methods on EJBContext
  - Secure Access to Resources Managers
  - Principal Mapping (mecanismos e ferramentas)
  - Runtime Security Enforcement
  - Audit Trail
- System Administrator
  - Security Domain Administration
  - Principal Mapping
  - Audit Trail Review

3. Identify correct and incorrect statements or examples about use of the isCallerInRole and getCallerPrincipal EJB programmatic security APIs.

- Ambos os métodos estão presentes na interface EJBContext.
- O método getCallerPrincipal() retorna um java.security.Principal que é representação da identificação do cliente (caller).
- Existe um método deprecated que é getCallerIdentity().
- O método isCallerInRole() recebe uma String como parâmetro e retorna um boolean dizendo se o caller pertence ou não a um determinado "role".
- Existe um método overloaded deprecated que recebe como parâmetro um objeto da classe Java.security.Identity.
- Ambos os métodos retornam o caller do EJB não sendo influenciado por alguma configuração de run-as.
- Para ser utilizado no código do bean, o "role" deve ser declarado com a annotation @DeclaredRoles ou com o elemento de deployment descriptor <security-role-ref>, que possui os elementos <role-name> e <description>.

4. Given a security-related deployment descriptor tag or annotation, identify correct and incorrect statements and/or code related to that tag.

- As "roles" utilizadas na aplicação devem ser definidas pela aplicação em annotations ou deplyment descriptor:
  - Para annotations deve ser declarado no Bean as annotations @DeclareRoles ou @RolesAllowed.
  - No deployment descriptor pode ser definido com <security-role> com o elemento <role-name> e <description>
- Definição de permissões de métodos:
  - As annotations @RolesAllowed, @PermitAll e @DenyAll podem ser utilizadas (as duas primeiras também podem ser aplicadas a nivel de classe)
  - No caso de haver uma superclasse do bean, somente os métodos sobrepostos alteram as definições de segurança feitas da superclasse (mesmo que implicitamente através das definições feitas a nível de classe)
  - Os métodos que não possuem definição são considerados "unchecked"
  - Pode ser definido também no deployment descriptor:

```
<method-permission>
   <role-name>
   <method>
      <ejb-name>
      <method-name>
      <method-parameters>
   </method>
</method-permisson>
```

- Criando o Link entre referências e os roles:

- Os roles podem ser linkados explicitamente utilizando o mesmo nome dos roles na annotation @DeclareRoles
- Se for usado um nome diferente, o seguinte pode ser usado para ligar a referência ao role realmente usado pela aplicação:

```
<session>
   ….
   <security-role-ref>
      <role-name>//Nome da role usado no bean
      <role-link>//Nome real da role
      <description>
   </security-role-ref>
</session>
```

- Pode ser definido um "run-as" para que um bean possa executar os métodos que chamar utilizando um outro role:
  - Pode ser usado a anotação de classe @RunAs
  - No deployment descriptor é definido assim:

```
<session>
   ….
   <security-identity>
      <run-as>
         <role-name>
      </run-as>
   </security-identity>
</session>
```