EJB notes.

EJB Callbacks (javax.ejb package)

EntityBean (extends EnterpriseBean)

Defines *7* container callbacks :

- `void setEntityContext (EntityContext ctx)`
- `void unsetEntityContext ()`
- `void ejbActivate ()`
- `void ejbPassivate ()`
- `void ejbLoad ()`
- `void ejbStore ()`
- `void ejbRemove ()`

SessionBean (extends EnterpriseBean)

Defines *4* container callbacks :

- `void setSessionContext (SessionContext ctx)`
- `void ejbActivate ()`
- `void ejbPassivate ()`
- `void ejbRemove ()`

SessionSynchronization

Defines *3* container callbacks :

- `void afterBegin ()`
- `void beforeCompletion ()`
- `void afterCompletion (boolean committed)`

EJB API (javax.ejb package)

EJBHome (extends java.rmi.Remote)

- `EJBMetaData getEJBMetaData ()`
- `HomeHandle getHomeHandle ()` `// EJB1.1 or above`
- `void remove (Handle h)` `// invalidates client stub`
- `void remove (Object primaryKey)` `// invalidates client stub`

EJBObject (extends java.rmi.Remote)

- `EJBHome getEJBHome ()`
- `Handle getHandle ()` `// Serialized stub`
- `Object getPrimaryKey ()` `// RemoteException if called by Session bean`
- `boolean isIdentical (EJBObject o)`
- `void remove ()` `// invalidates client stub`

EJBContext (SessionContext, EntityContext)

- `Principal getCallerPrincipal () / boolean isCallerInRole (String name)`
- `EJBHome getEJBHome ()`
- `EJBObject getEJBObject ()` `// SessionContext/EntityContext`
- `Object getPrimaryKey ()` `// EntityContext only`
- `void setRollbackOnly () / boolean getRollbackOnly ()`

EJBMetaData

- `EJBHome getEJBHome ()`
- `Class getHomeInterfaceClass ()` `// N.B. no method for bean class -`
- `Class getPrimaryKeyClass ()` `//  never exposed to client`
- `Class getRemoteInterfaceClass ()`
- `boolean isSession`
- `boolean isStatelessSession ()` `// EJB1.1 or above`

List the required classes/interfaces that must be provided for an EJB component.

Common

- Home interface (extends javax.ejb.EJBHome) - defines the create methods (+ finders for Entity)
- Remote Interface (extends javax.ejb.EJBObject) - defines the business logic interface
- Bean class - implements the EJB callbacks, "implied" EJB callbacks and business methods
- Primary Key class – Entity beans only

Entity Beans.

	Description
Remote interface	Defines the business methods

Home interface	Defines the factory and locator methods :
	
	`// required methods`
	`<REMOTE_IF> findByPrimaryKey (<PK> pk)`
	
	`// optional methods`
	`<REMOTE_IF> create (<PARAMS>)  // multiple create methods are allowed`
	
	`<REMOTE_IF> find<METHOD_NAME> (<PARAMS>)`
	`Collection find<METHOD_NAME> (<PARAMS>)  // EJB 1.1 or above`
	`Enumeration find<METHOD_NAME> (<PARAMS>)  // EJB 1.0`

Entity Bean class	Implements javax.ejb.EntityBean (ejbLoad (), etc) and the business methods defined in the remote interface. Also implements the "implied" EJB callbacks defined in the home interface : • `<REMOTE_IF> create (<PARAMS>)` becomes `<PK> ejbCreate (<PARAMS>)` For each `ejbCreate (<PARAMS>)` there must be a matching `void ejbPostCreate (<PARAMS>)` • `<REMOTE_IF> find<METHOD_NAME>(<PARAMS>)` becomes `<PK> ejbFind<METHOD_NAME>(<PARAMS>)` Finders that return a collection in the interface return a collection of primary keys in the bean. `ejbCreate` has different return values/types depending on the EJB version and persistence type : • EJB1.1, CMP : `<PRIMARY_KEY> ejbCreate (<PARAMS>)` – returns null (container knows PK) • EJB1.x, BMP : `<PRIMARY_KEY> ejbCreate (<PARAMS>)` – returns primary key • EJB1.0, CMP : `void ejbCreate (<PARAMS>)` - (container knows PK) The primary key returned by `ejbCreate` is cached in the EJBObject so that identity can be determined on activation `ejbFind` methods that return a collection have different returns depending on the EJB version : • EJB1.1 : if no matches found, return an empty collection • EJB1.0 : if no matches found, return null
Primary Key class	You can use an existing class (e.g. String) or create one. The primary key class **must :** • implement java.io.Serializable • provide a default constructor (the container uses Class.newInstance (); additional constructors are allowed) • override hashCode () and equals () to ensure that storage and comparison operate as expected (a *must* in EJB1.1, a *should* in EJB1.0) EJB1.1 allows "undefined" primary keys – coded generically as Object and set specifically by the deployer.

Session Beans.

	Description
Remote interface	Defines the business methods

Home Interface (Stateless Session Bean)	Define the factory method :
	
	`// required method`
	`// multiple create methods are not allow`
	`<REMOTE_IF> create ()`

Home Interface (Stateful Session Bean)	Defines the factory methods :
	
	`// required methods`
	`<REMOTE_IF> create ()`
	`<REMOTE_IF> create (<PARAMS>)`
	
	`// multiple create methods are allowed, at least one must be defined`

Session Bean class	Implements javax.ejb.SessionBean (ejbActivate (), etc) and the business methods defined in the remote interface. Also implements the "implied" EJB callbacks defined in the home interface : • `SLSB : <REMOTE_IF> create ()` becomes `void ejbCreate ()` • `SFSB : <REMOTE_IF> create (<PARAMS>)` becomes `void ejbCreate (<PARAMS>)` `ejbActivate () / ejbPassivate () :` • SLSB : these methods are meaningless, SLSB aren't swapped to/from disk • SFSB – the following types are maintained on activate/passivate EJB1.x : primitives, serializable objs, SessionContext, remote refs to other beans EJB1.1 : JNDI naming contexts, home refs Any other types must be labelled as `transient` or set to null in `ejbPassivate ()` and restored in `ejbActivate ()` – e.g. on passivate, close socket client and set to null; on activate, re-open and set var. *N.B. transient fields won't be reset on activate – they may be set to whatever the last bean that used them left them as. Therefore, transient vars should be reset in* `ejbActivate ()`

EJB 2.0

As of EJB2.0, local versions are available :

- Local Home/Local interfaces (extend javax.ejb.EJBLocalHome/EJBLocalObject)
- Bean
  Still implement javax.ejb.SessionBean/EntityBean but have to use the local context – e.g. ctx.getEJBLocalHome ()

EJB2.0 also introduces the MessageDrivenBean :

- no local/remote interface
- Bean implements javax.ejb.MessageDrivenBean and javax.jms.MessageListener

Distinguish between stateful and stateless session beans

Stateful Session Bean

- models business logic, processes, workflow, etc.
- associated with a single client for the life-time of the bean.
- maintains client-specific state between method calls – i.e. the interaction is somewhat drawn out
- can be viewed as an extension/agent of the client
- resource management : swapped

- creates less network traffic than stateless – no need to pass state every time

Stateless Session Bean

- models a service as a re-usable object
- associated with a single client for the duration of the method call.
- no client-specific state maintained, state passed in as parameters and the result returned – i.e. the interaction is short
- resource management : pooled

- more scalable than stateful as a small number can service a large number of clients

Distinguish between session and entity beans

"like .. a script for a play .. and the actors that perform the play .. entity beans are the actors and props, the session bean is the script"

Session Bean

- "owned" by a single client
- models a short lived business process (only lasts as long as the client session; dies on system crash)

Entity Bean

- shared by many clients
- object "view" (provides safe/consistent access) of a long-lived/persistent business entity (survives a system crash)
- resource management : pooled

- more re-usable than session beans

Recognize appropriate uses for entity, stateful session and stateless session beans

Entity Bean

- User, Customer, Account, Order

Stateful session.

- Shopping cart, AccountManager

Stateless session.

- Credit card authorization, currency converter, SocketClientService

State the benefits and costs of container-managed persistence

Pros.

- productivity – don't have to write/debug tedious database mapping code
- cleaner code – bean isn't bogged down with persistence code
- performance – the container can implement caching
- portability – as DB code is handled by the container, CMP beans can be moved to other containers and be persistent (*PJC : don't agree, deployment isn't portable)*

Cons.

- portability – in EJB1.x, deployment of CMP wasn't portable
- simplistic mapping – in EJB1.x, CMP didn't support relationships, dependent objects, etc.
- lack of control
- legacy support – not every data store is a database but most vendors only support DB mapping

State the transactional behaviour in a given scenario for an enterprise bean method with a specified transactional attribute as defined in the deployment descriptor

BMT (Bean Managed Transactions).
EJB1.x : clients can manage TX using JTA; TX isolation level can be set for each specific resource manager, e.g. JDBC
EJB1.1 : session beans can manage TX using JTA and setting `<transaction-type>Bean<transaction-type>`
EJB1.0 : session & entity beans can manage TX using JTA and setting the TX attribute to `TX_BEAN_MANAGED`

CMT (Container Managed Transactions)
The recommended practice for EJB is to use CMT :

- cleaner code – bean isn't cluttered with transactional code
- distributed support – the container can organise multiple ResourceManagers
- reusable - beans can be reused in different circumstances by changing the deployment descriptor
- flexible – can be specified per method, per bean (N.B. if any method use BMP, all must)

The transactional *context* (a.k.a. *scope*) defines what beans are participating in the current transaction.
When using CMT the container starts and ends the transaction when a transactional method is called (i.e. declared as starting a transaction in the deployment descriptor).
Consequently, the transactional context will be propagated to all *targets* (methods/beans used within the method that "started" the transaction).

The targets can control their participation in the propagated transaction by setting one of ***6*** *transactional attributes* in the deployment descriptor (e.g. `<trans-attribute>Required</trans-attribute>`) :

Attribute	***Description***
NotSupported (TX_NOT_SUPPORTED)	Current TX suspended until called method completes
Supports (TX_SUPPORTS)	If caller bean part of a TX, called method/bean joins TX and executes. Otherwise, called method executes as is.
Requires (TX_REQUIRED)	If caller bean part of a TX, called method/bean joins TX and executes. Otherwise called method/bean starts a new TX (applies to called method/bean and any other methods/beans it calls). After the called method/bean completes, the new TX scope ends.
RequiresNew (TX_REQUIRES_NEW)	Called method/bean always starts a new TX (applies to called method/bean and any other methods/beans it calls). After the called method/bean completes, the new TX scope ends. If caller bean part of a TX, current TX is suspended until called method completes.
Mandatory (TX_MANDATORY)	If caller bean part of a TX, called method/bean joins TX and executes. Otherwise called bean throws javax.transaction.TransactionRequired
Never (EJB1.1 only)	If caller bean part of a TX, throw RemoteException Other, called bean/method executes as is.
TX_BEAN_MANAGED (EJB1.0 only)	Caller bean/method creates a new TX using JTA, the current TX is suspended until the called method completes.

In EJB1.0 CMT, it was possible to control transaction further by setting the transaction isolation level in the deployment descriptor on a per method basis
In EJB1.1 CMT, it's still possible to change the isolation level but it's container dependent.

Miscellaneous

A transaction represents a unit-of-work (one or more tasks that must all complete) and embodies the concept of an exchange between two parties. In EJB each task is represented as a method so the unit-of-work is represented as one or more method calls on one or more beans.

A transactional system must enforce the ACID properties :

Property	***Description***
Atomic	All or nothing – if any task fails the transaction if rolled back; if all complete the transaction is committed (i.e. made Durable).
Consistent	The state of the system must reflect the real world. This is enforced in two parts : 1. the transactional system ensures that data is Atomic, Isolated and Durable 2. the application developer ensures the system has appropriate constraints – e.g. no primary key violation, referential integrity, etc.
Isolated	The transactional system ensures that data involved in a current transaction cannot be affected by other transactions until the current transaction completes. The degree of isolation is controlled by setting the "isolation level".
Durable	The transactional system must ensure that the data is persistent before the transaction can be considered to be complete – i.e. until committed, the data is still isolated. Additionally, the transactional system ensures that the data survives system failures through 2PC (two-phase commit) and transaction logging.

A transactional system has to be able to deal with concurrency issues :

Issue	***Description***
Dirty read	TX2 reads uncommitted changes made by another TX1; if TX1 rolls back, TX2 has dirty data. E.g. TX1 attempts to book the last seat but fails on payment; TX2 didn't see the seat so thinks it's sold out.
Non-repeatable read	"Lost update" TX1 reads; TX2 reads and updates; TX1 updates and blats over TX2's update
Phantom read	TX1 gets a list of seats; TX2 adds a seat; TX1 is unaware of the update

Concurrency issues can be controlled using one of ***4*** *transaction isolation levels* :

	Read Type		
Isolation Level	***Dirty***	***Non-Repeatable***	***Phantom***
TRANSACTION_READ_UNCOMMITTED	Y	Y	Y
TRANSACTION_READ_COMMITTED	N	Y	Y
TRANSACTION_REPEATABLE_READ	N	N	Y
TRANSACTION_SERIALIZABLE	N	N	N

These isolation levels have to be viewed from the perspective of each client :

Isolation Level	***Description***
READ_UNCOMMITTED	Reader & Writer No locks/waiting; anything goes. Reader picks up all column changes (synched at row level)
READ_COMMITTED	Reader - only wants "real" data (i.e. that made durable). Wait until available. Writer Wait until others have finished reading before making changes. If writer gets there first, readers wait until writer has finished.
REPEATABLE_READ	Reader - Blocks writer until finished reading or waits until writer is finished Writer No updates or deletions to existing rows; insert of rows allowed. Don't allow any updates / deletions to stuff writer is working on – writer could overwrite updates.
SERIALIZABLE	Reader - Blocks writer until finished reading or waits until writer is finished Writer No updates, deletions or insertions except by the writer

Given a requirements specification detailing security and flexibility needs, identify architectures that would fulfill those requirements

Architectures.
Refer to the "Common Architecture" and "Security" documents.
In summary :

- a modular architecture is the most flexible but requires more effort to manage
- security is required at all levels and a modular architecture supports this
- Java has good security support – JVM, security APIs, etc
- typical elements that support security and flexibility are : firewalls, load balancers, clustered application servers, etc
- EJB security is flexible – declarative so can be adjusted as required; can be mapped to environment specifics (LDAP, mainframe, etc.)

J2EE Security.
Within an EJB component *security role*s are defined - a role is a logical user (e.g. admin, customer). When the component is deployed, the role is mapped to "real" J2EE users/groups (e.g. PJC is able to assume the admin role).

To access secure resources, the *Subject* (client) must be authenticated and authorized :

- Authentication.
  To authenticate, the Subject must present their *Credential*s (e.g. username/password) to the container.
  If the authentication is successful, the Subject is associated with one or more *Principal* objects.

  N.B. authentication is not currently part of the EJB specification – e.g. the credentials may be passed as properties to an InitialContext, JAAS may be used, etc.

- Authorisation.
  Once authenticated, the Principal(s) can be checked against the Role to decide if the client is authorised to access the secure resource.
  Role based authorisation is fully supported by J2EE – security constraints in web.xml, method level security in EJB deployment descriptor, "runAs" in deployment descriptor.

  N.B. instance based authorisation (e.g. PJC is only allowed access to PJC's account) isn't yet specified

J2EE also has other built-in security features :

- web-tier authentication – via HTTP authentication, FORM/j_security_check, client certificates
- secure communications – SSL; web-tier authentication can use SSL for HTTP/FORM auth
- security packages – digests, digital signatures, ciphers, certificates, etc.

EJB security
EJB supports declarative security (programmatic security is also supported) by setting the appropriate values in the deployment descriptor :

```
<method-permission>
        <role-name>admin</role>
        <method>
                <ejb-name>Account</ejb-name>
                <method-name>*</method-name>
        </method>
</method-permission>
```

Not very portable at the moment – authentication and "runAs" are container specific; container to container security propagation is undefined.

Identify costs and benefits of using an intermediate data access object between an entity bean and the data resource

Pros.

- transparency – specifics of data access are hidden from the entity bean
- modular – the DAO could be swapped for another with little disruption to the entity bean code
- cleaner code - data access factored out so simplifies entity bean code
- reusable – DAO could be re-used in a "Fast Lane Reader"

Cons.

- only suitable for BMP
- extra code - have to write DAO class, DAO wrapper calls in entity bean, maybe some DAO factories

Miscellaneous.

General restrictions.

- threads – no synchronized modifiers or blocks; no thread management
- networking - no socket servers or multicast; can't change socket factory
- security - no reflection; no access to ClassLoader/SecurityManager; exit JVM; native libs
- no AWT
- no read/write static fields – static final fields OK

Bean restrictions.

- bean class – class modifier must be public (abstract, final not allowed); no finalize ()
- business & callback methods
  method modifier must be public (static, final not allowed); arguments must be legal RMI types

Exceptions

Application exceptions represent business logic failures so are expected to be returned to the remote client.
Some common application exceptions are provided in the javax.ejb package but can also be user-defined.

System exceptions represent system level failures and are excepted to be caught by the container.
The container handles the exception and re-packages it as RemoteException before returning it to the client.

The container will deal with all unchecked exceptions – e.g. NullPointerException
Bean methods can catch exceptions are re-throw within a method – e.g. catch JDBC error and re-throw as EJBException.

In EJB1.0, methods could throw RemoteException to indicate a system/non-application error.
In EJB1.1, methods are now expected to throw EJBException (or another RuntimeException).
However, all of the sample code (including Petstore) still uses RemoteException.

The bean method implementations must match the interface definitions in terms of exceptions apart from :

- RemoteException - the container handles remote exception issues
- EJBException - extends RuntimeException so doesn't have to be explicitly listed in the "throws" clause.

Method	***Application Exceptions***
Any	Application/custom exceptions
`create ()`	`CreateException, DuplicateKeyException`
`findXXX ()`	`FinderException, ObjectNotFoundException` (single return finders only)
`remove ()`	`RemoveException`
business methods, ejbLoad, ejbStore	`NoSuchEntityException` (object has been removed elsewhere, as of EJB1.1)

Primary Services.

There are *7* primary services provided by the container : *concurrency* (to maintain safe/consistent access to shared entity bean), *lifecycle management* (pooling/swapping), *persistence* (CMP), *distributed objects* (EJB uses RMI-IIOP), *naming* (JNDI can hookup to a number of naming services), *transactions* (CMT) and *security* (EJB builds on Java2 security to add method level security).