EJB Container Model notes.

General.
Distributed EJB objects are achieved by use of the following elements :

- client - orchestrates lookup/creation of stubs; uses stubs for business operations
- client stub - responsible for network aspects (marshalling, etc.) via a transparent business logic facade
- server skeleton - responsible for servicing client requests and passes them onto the EJBObject
- EJBObject - responsible for security, transactions, etc. before delegating method calls to the bean
- bean class - actually implements the business logic

Behind the scenes.

Code Fragment	*Description*

`Context ic = new InitialContext ();`
`Object ref = ic.lookup`
`("java:comp/env/ejb/Account");`
 	
`AccountHome ah = (AccountHome)`
`PortableRemoteObject.narrow (ref);`
	`// CREATES EJBHome`
	`RETURNS CLIENT STUB`

`Account a = ah.create ("123456", "PJC");`	`// CREATES EJBObject`
	`CREATES SKELETON LISTENING ON IP/PORT (CONCEPTUALLY)`
	`ASSOCIATES SKELETON WITH EJBObject`
	`ASSOCIATES BEAN WITH EJBObject`
	`RETURNS CLIENT STUB WITH DETAILS OF SKELETON IP/PORT`

`String name = a.getName ();`	`// GETS CONNECTION TO SKELETON (IF HAVEN'T ALREADY)`
	`FLATTENS REQUEST`
	`TRANSMITS REQUEST ON CONNECTION`
	`GETS RESPONSE ON CONNECTION`
	`OBJECTIFIES RESPONSE`

State the benefits of bean pooling in an Enterprise JavaBeans container.

- reduces the resource requirements for a single server – pooled beans are context switched as required
- dynamic growth – pool can be expanded/contracted as demand requires
- pooled objects are instantiated on startup instead of every time – may be "expensive" to instantiate.
- fine-grained control of resources – able to set max/min beans
- transparent to clients

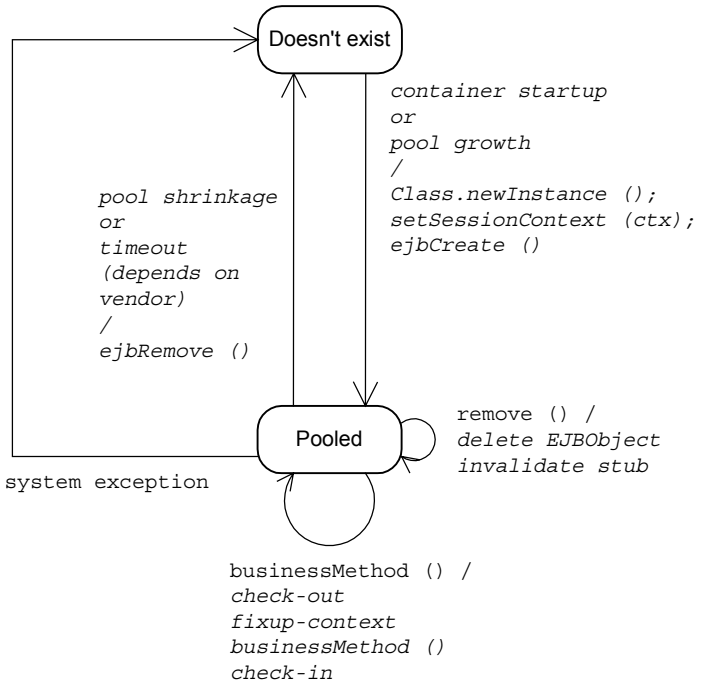
Explain how the Enterprise JavaBeans container has the capability to increase scalability.
Dynamic resource management
Lifecycle management enables a fine-degree of control over resources :

- a small pool of beans can service many clients
- resources can be swapped in/out as demand requires
- container can employ caching

Clustering
Containers have good support for clustering at all levels :

- distribute load over several machines
- built in load balancing – JNDI, smart stubs, request forwarding

Stateless Session Bean Lifecycle.



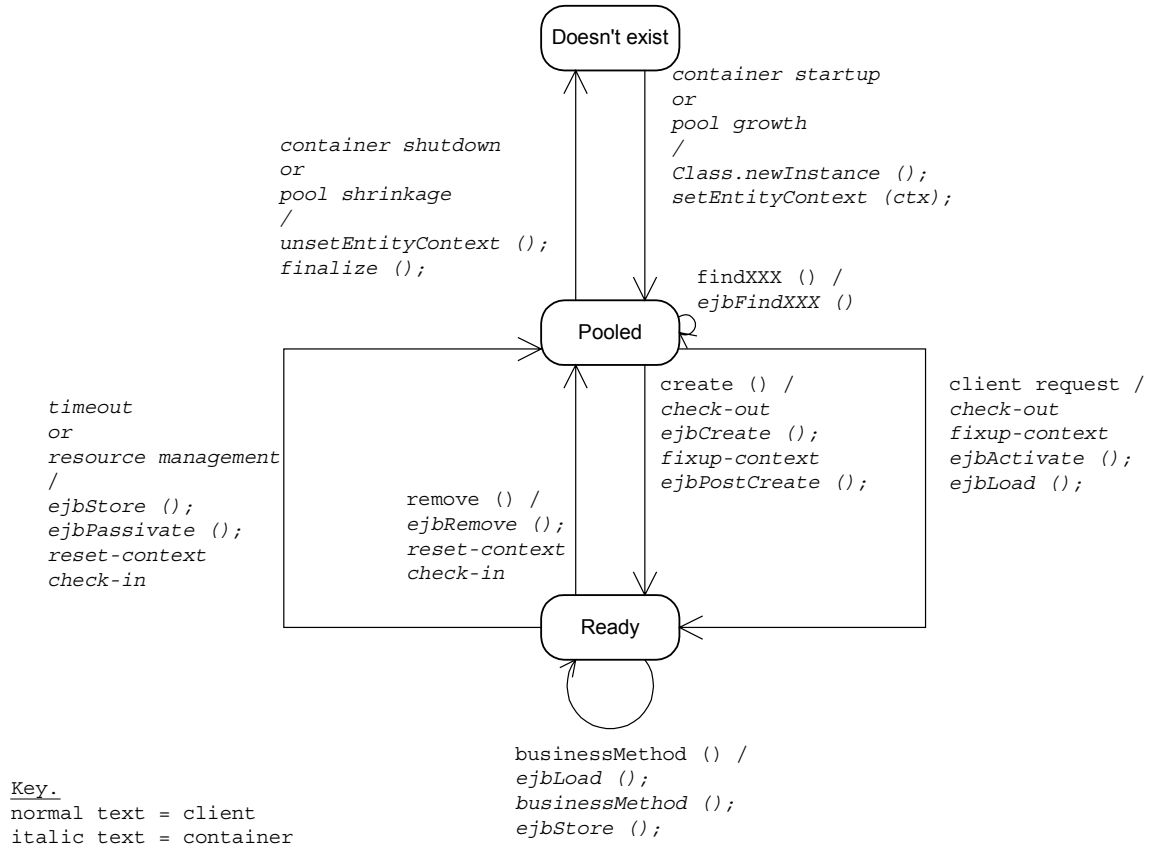
Key.
normal text = client
italic text = container

Notes.

Item	*Description*
`check-out`	marks the bean as in-use
`check-in`	marks the bean as available
`fixup-context`	Associates the bean with the EJBObject which enables the bean to get details of the security context, etc.

`setSessionContext (ctx) or ejbCreate ()`
 	
`ejbRemove ()`
	Be careful not to access a null variable within `setSessionContext (ctx)or ejbCreate ()` – if it doesn't complete, the bean can't be created and pooled.
	Bean generic instance variables can be set/reset using these methods - e.g. a socket client.

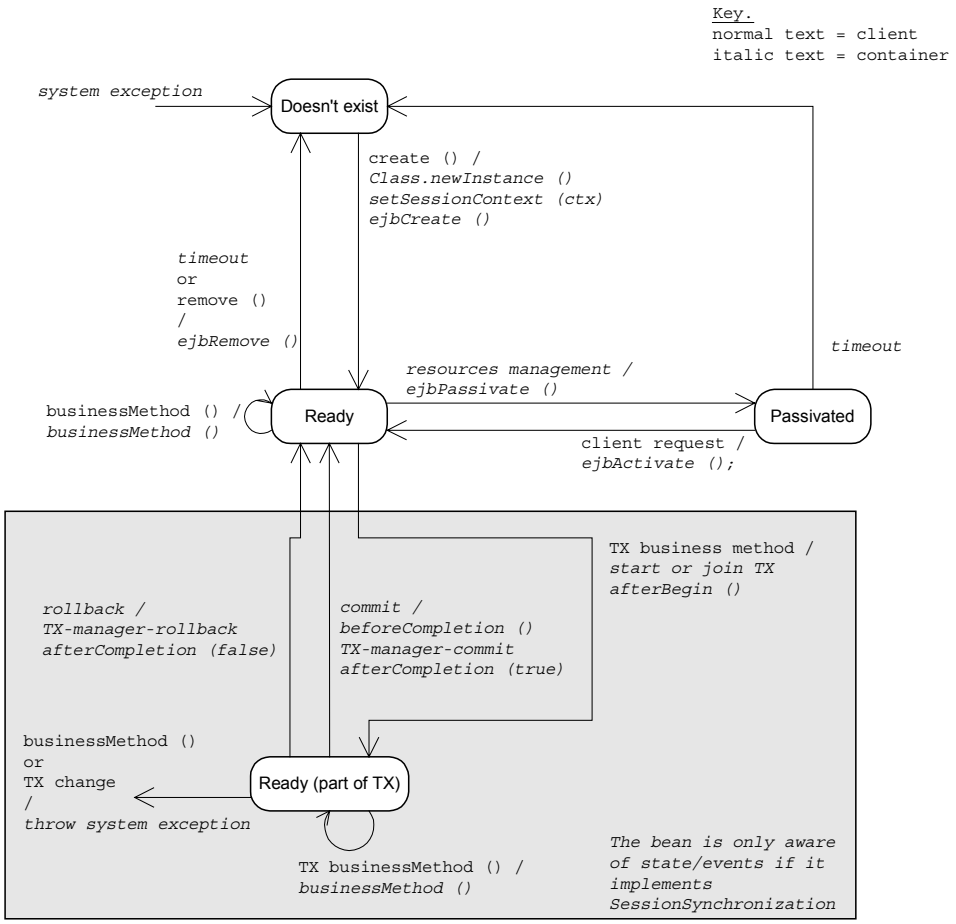
Entity Bean Lifecycle.



Notes.

Item	***Description***
`check-out`	marks the bean as in-use
`check-in`	marks the bean as available
`fixup-context`	Associates the bean with the EJBObject and sets the primary key. Consequently, the EJBObject or primary key can only safely be retrieved from the EntityContext as of the start of ejbPostCreate ()
`reset-context`	disassociates the bean from the EJBObject and clears the primary key
`findXXX`	If the finder returns a single object and it's found, the container returns the found bean. If the finder returns multiple object and some are found, an EJBObject and skeleton are created and the stub returned to the client. If the client decides to use the bean it will be activated on method request; if they don't use the bean, the container has saved making a potentially expensive ejbLoad () call for each bean found.
`ctx.getEJBObject ()` `ctx.getPrimaryKey ()`	If a bean isn't associated with an EJB object and these methods are called (e.g. within a finder method), IllegalStateException will be thrown
`setEntityContext (ctx)` `unsetEntityContent ()`	Be careful not to access a null variable within `setEntityContext (ctx)` – if it doesn't complete, the bean can't be created and pooled. Bean generic instance variables can be set/cleared using these methods - e.g. a Log instance; a bean-wide socket (one that's needed for finder methods)
`ejbActivate ()` `ejbPassivate ()`	Event notification - indicates that the bean is moving out of / into the pooled state. Bean specific instance vars can be set/reset using this event notification – e.g. a socket to a particular server based on the user details
`ejbRemove ()`	Resources should be cleared – e.g. use ejbPassivate ()
`ejbLoad () ejbStore ()`	The operation of ejbLoad/Store () can be configured as required according to the transactional requirements – e.g. load,method,store; load,method, method,store, etc.

Stateful Session Bean Lifecycle.



Notes.

Item	***Description***

`setSessionContext (ctx)` `or ejbCreate ()`
 	
`ejbRemove ()`
	Be careful not to access a null variable within `setSessionContext (ctx)` or `ejbCreate ()` – if it doesn't complete, the bean can't be created and pooled.
	Bean generic instance variables can be set/reset using these methods - e.g. a Log instance.
	N.B. ejbRemove () isn't called if the bean times out while passivated (the container would have to activate the bean just to call ejbRemove () !)

`ejbActivate ()` `ejbPassivate ()`	Event notification – the container is just about to serialize/deserialize the instance fields (on passivation, a serialized object ID or the serialized values themselves will be stored in the EJBObject so that the state can be restored on activation). Only primitives, the SessionContext, home refs, remote refs, the JNDI context and any other non-transient, serializable fields can be passivated – anything else (transient fields excluded) should be set to null and restored when the bean is activated. If there's a failure during passivate/activate the bean will be destroyed. Bean specific instance vars can be set/reset using this event notification – e.g. a database connection that's based on the shopping cart
`ejbRemove ()`	Resources should be cleared – e.g. use ejbPassivate ()
`Doesn't exist`	A SFSB can be destroyed at any time if a method throws a system exception
`Transactions`	As a SFSB maintains client specific state over several method calls, an additional state "Ready (part of TX)" is required. If a transactional method is invoked, the bean is notified of the transition and the state switches to Ready-TX. Once in this state, only other transaction methods (and of the same TX type) are allowed. If a non-TX method is invoked, an exception is thrown and the bean destroyed. Non-TX methods can be invoked once the transaction has finished and the bean has transitioned to "Ready"