Stateful Session Beans

- Conversational State
- Stateful Session Bean Lifecycle

What is a Stateful Session Bean?

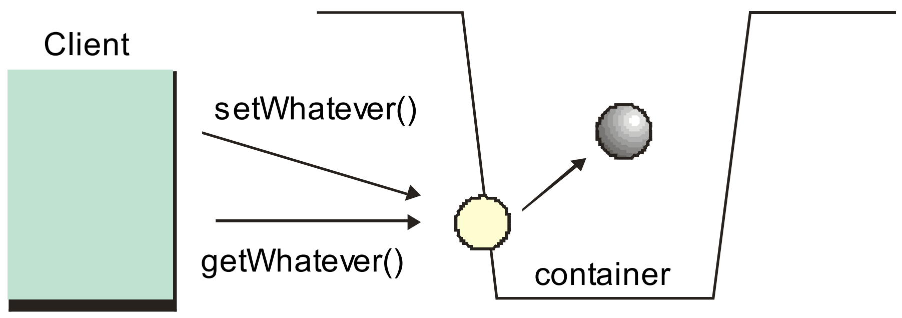
- Stateful session beans let clients perform workflow that spans multiple methods -- the bean "remembers" state from one call to the next
- Stateful session beans can be used in "shopping-cart" applications
- As such, stateful beans are essentially an extension of the client, and let you migrate functionality from the client to the server for easier maintenance
- However, like all session beans, stateful session beans are *not* persistent -- the container does not store them in a database

In some ways, stateful beans are similar to local Java classes -- you can call their methods expecting the methods to modify the bean's internal state. But like all EJBs, they operate within the bounds of a container, which provides transactions and security (we will cover those topics later).

The notion mentioned here about moving functionality to the server is a common theme in client/server programming. By moving common function to the server, you allow for "thinner" clients. It's also easier to maintain the system -- if you decide to change the behavior, you only need to modify code on the server instead of on all of the clients.

Maintaining Conversational State

- Unlike stateless session beans, stateful beans can define instance data that the bean customizes on behalf of a client



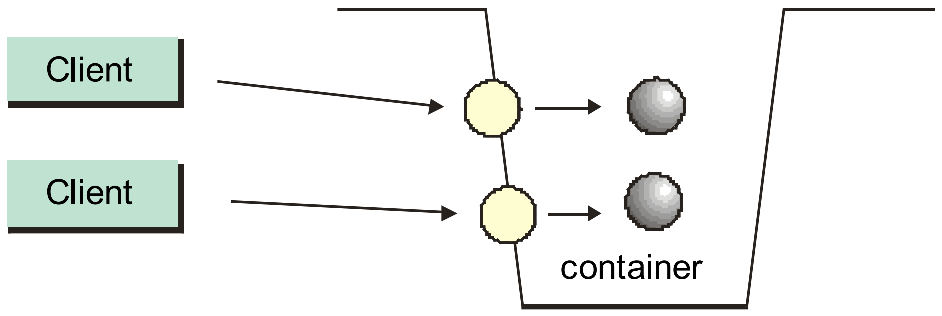
Stateful beans can define fields just like local Java classes do -- the container will guarantee that the bean's state is maintained across method calls. That lets you write the normal "get/set" methods that JavaBean developers are accustomed to writing.

But remember that the container does not store the instance data persistently, so when the client is done with the bean (or the container crashes), the state is lost. This is in contrast with entity beans, which we will cover later.

So the real use of stateful beans is to let your application maintain state in a similar fashion to HTTP sessions. In other words, you can use a series of requests (methods) from the client to accomplish a complex task.

Stateful Session Beans are Dedicated

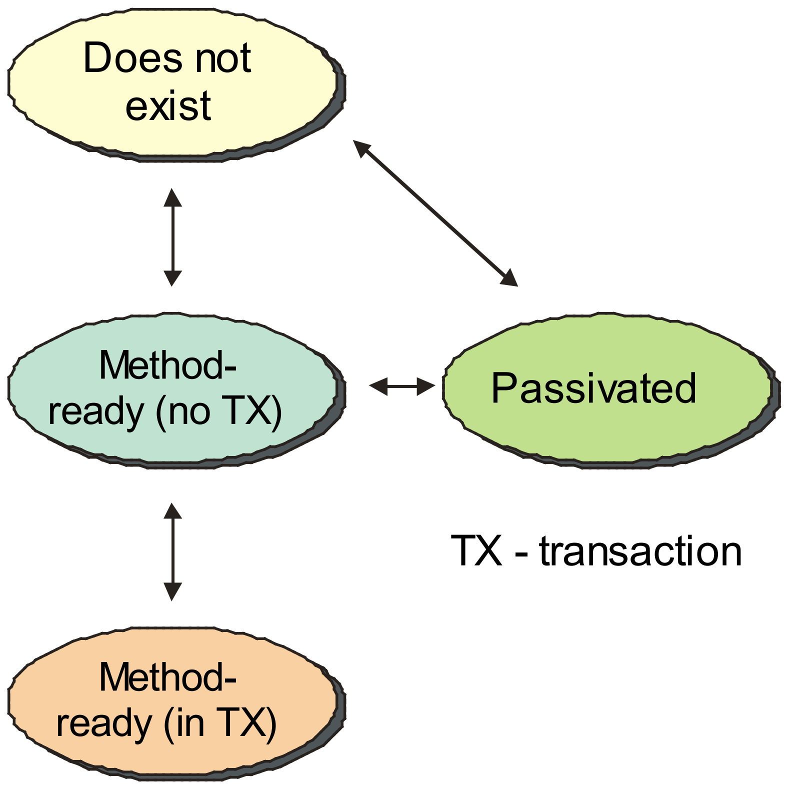
- Unlike stateless session beans, the container assigns each stateful bean EJB object to a single bean instance (no pooling)
- In other words, each stateful bean belongs to a single client



For conversational state and parameterized create methods to work, the container must guarantee that each bean instance (and EJB object) is dedicated to a single client. Thus the container cannot perform instance pooling as it can with stateless beans. That means that containers cannot manage the storage for stateful beans as efficiently as for stateless beans, where a small number of beans can service large numbers of clients.

Another possible bad side-effect is that if you have many clients using stateful beans, the container could eventually run out of memory. In a moment, we will see that the container can essentially temporarily "page out" beans to help manage its working set.

Stateful Session Bean Lifecycle



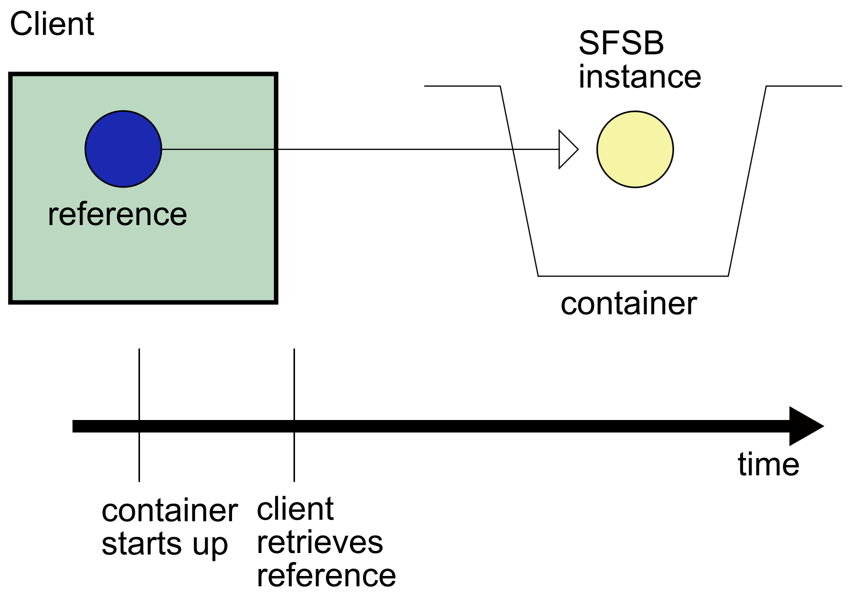
Stateful beans have a different lifecycle from stateless beans. Since they are not pooled, they go directly to the "ready" state at creation and are assigned to a single client. They can return to the non-existent state if the client calls a "remove" method.

If the container decides it's appropriate, it can save the bean's state somewhere and remove the bean from memory. This is referred to as passivation</i>. If the client again accesses the SFSB, the container re-creates the bean and initializes it from the saved state: this is referred to as activation</i>. Note that a container can remove passivated beans permanently -- most containers use a timeout period after which they "reap" beans that have not been accessed in a long time.

We will cover transactions later, but note that the only transition out of the in-transaction state is back to method-ready -- the container will not passivate or remove beans that are involved in transactions.

Client Retrieves Reference

- Clients retrieve stateful bean references in the same fashion as for stateless beans:
  - Using **dependency injection** in "managed" clients
  - Using JNDI in standalone clients



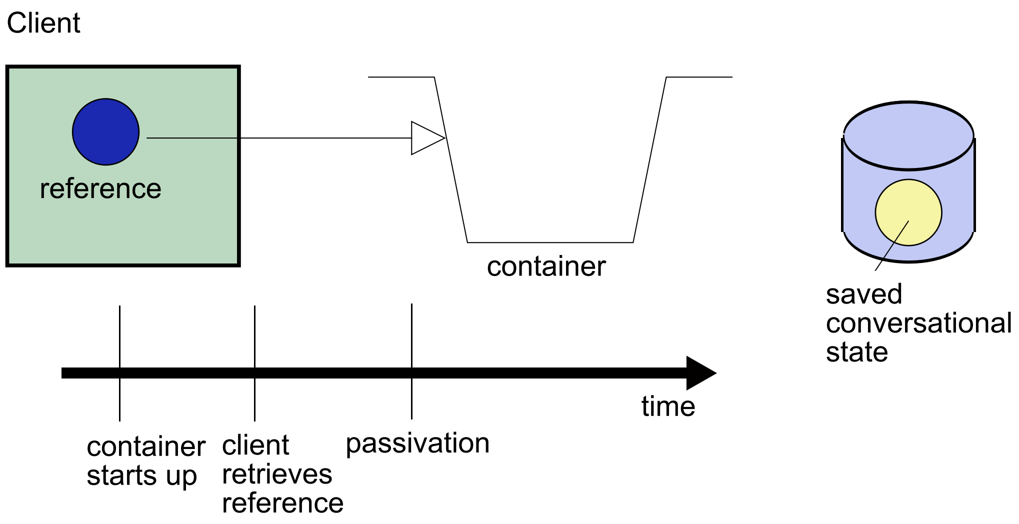
When the client retrieves a reference, the container instantiates the bean implementation, injects resources, and optionally calls any @PostConstruct method. The bean is now in the method-ready state and assigned to that particular client.

It's important to note that the container creates a new SFSB instance each time you use JNDI to look up the EJB's business interface reference. In other words, there's no explicit "create" method -- the act of doing the JNDI lookup creates the instance.

Also note that servlets should not use dependency injection on a field to reference a SFSB instance, since servlets are multithreaded and the single field value would be intermingled across multiple users. Instead, servlets use JNDI look ups to create instances and then store the SFSB references in the HTTP session.

Passivation

- Based on container-specific policies, the container can save a stateful bean's state and temporarily remove it from memory



At the container's discretion, it can manage memory by removing SFSBs from storage, first saving their state in some sort of disk-based storage.

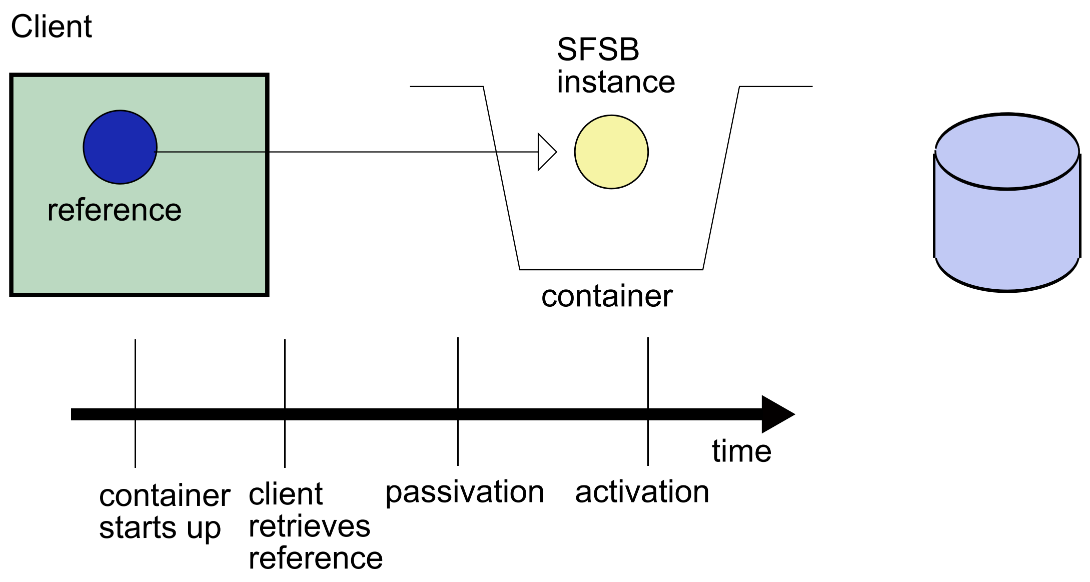
Before passivating, the container optionally calls any method marked with @PrePassivate.

The client's reference is unaffected by passivation-- as far as the client's concerned, nothing's happened to its bean.

Note that the storage used for passivation is not permanent -- when the container shuts down, it's free to delete the passivated objects. Remember that all session beans are non-persistent!

Activation

- When the client later calls a method, the container automatically resurrects the stateful bean and re-dispatches the method



The container tracks which beans are in memory and which are passivated. When the client references a passivated bean, the container creates a new bean instance, attaches it to the client's reference and re-initializes the bean from the passivated store. The container then calls any method annotated with @PostActivate. The container then re-runs the original method.

The client is unaware of this entire process except that the method may take a bit longer than expected.

Client Removes Reference

- A SFSB implementation can mark one or more methods as **remove** methods
- When a client calls a "remove" method, the container can make the instance available for garbage collection

```
@Stateful
public class MyImplementation
  implements MyInterface
{
. . .
  @Remove
  public void removeMe()
  {
  }
}
```

Providing "remove" methods is a best practice since it makes it easier for the container to manage memory.

You can specify "remove" methods either with the annotation or in the EJB deployment descriptor.

Since there is no instance pooling for stateful beans, when the client removes the remote reference, the container can immediately make the bean available for garbage collection. The container does first call any method annotated with @PreDestroy before making the instance available for garbage collection.

Rules for Conversational State

- The rules for what is allowed for conversational state in a SFSB mirror the rules for Java **serialization**, including:

    - A serializable object
    - A null
    - A reference to an enterprise bean's business interface
    - A reference to the SessionContext object
    - A reference to a container-managed EntityManager object
    - A reference to an EntityManagerFactory object obtained via injection or JNDI lookup
    - A reference to a javax.ejb.Timer object

Transient Fields

- When a stateful bean is passivated, the container saves all non-transient fields and restores them when the been is re-activated
- Note that the container may use Java serialization, but this is not mandated

```
@Stateful
public class MyImplementation
        implements MyInterface
{
    private int someInt;
    private transient int otherInt;

    . . .
}
```

2 - 11

The Java language includes the transient</i> keyword which the container examines to determine if a particular field should be saved during passivation. The idea is that the bean may have temporary or calculated fields (especially large ones) that don't need to be saved since they can be re-calculated when the bean is activated. You should mark any such fields as transient.

Java serialization is a standard API that makes it pretty easy to save the state of an object -- we saw an example of it earlier when a client serialized an object handle. The container vendor is free to use serialization, but can use some other technique if it so chooses, perhaps for better performance or to take advantage of existing infrastructure.

Session Timeout

- Using container-specific techniques, the deployer can specify a inactivity timeout period after which the container can remove the session bean from memory

This notion is similar to the session time in HTTP-based applications. The issue is that clients may terminate unexpectedly or "forget" to call the remove method. To protect against runaway memory (or disk) usage, the container can "reap" beans that have not been accessed for a certain period.

Concurrency

- The container will never simultaneously invoke more than one method on a given session bean instance
- Thus the implementation does not have to take steps to be thread safe

This makes writing the EJB much simpler, since you don't need to worry about synchronization.

Lifecycle Notification

- If you want the container to notify your implementation of lifecycle events (e.g., the instance is created), you can use the **@PostConstruct**, **@PreDestroy**, **@PrePassivate** and **@PostActivate** annotations on methods or an **interceptor** class

```
@Stateful
public class MyImplementation implements MyInterface
{
  @PostConstruct
  public void myCreate() {. . .}

  @PreDestroy
  public void myDestroy() {. . .}

  @PrePassivate
  public void myDestroy() {. . .}

  @PostActivate
  public void myCreate() {. . .}
}
```

The PostConstruct callback invocations occur before the first business method invocation on the bean. This is at a point after which any dependency injection has been performed by the container.

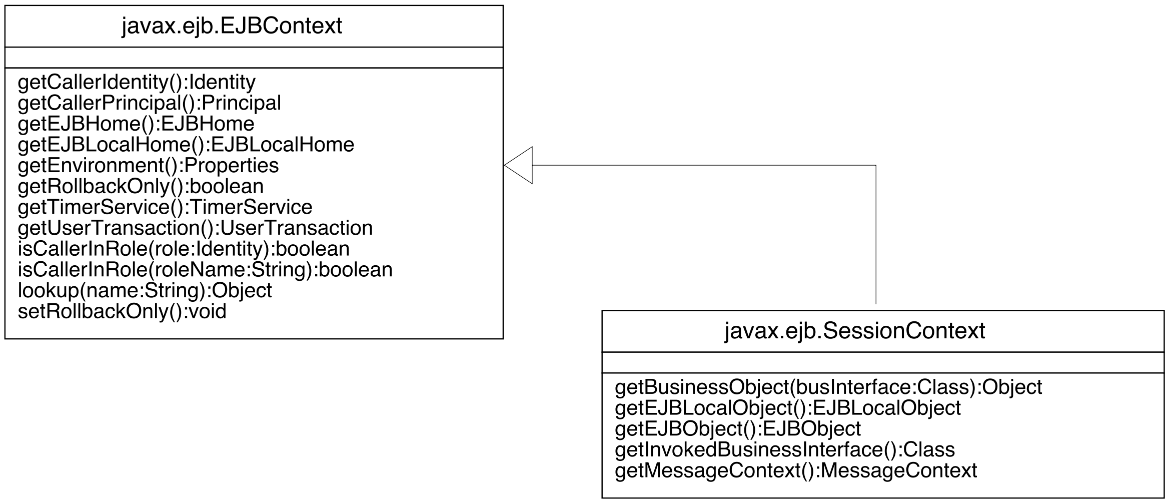
The PreDestroy callback notification signals that the instance is in the process of being removed by the container. In the PreDestroy lifecycle callback interceptor methods, the instance typically releases the resources that it has been holding.

The @PrePassivate and @PostActivate methods let a stateful bean correctly manage resources such as open sockets and database connections, since these should be closed when a bean is passivated.

As usual, instead of using annotations, you can configure these lifecycle methods in the EJB deployment descriptor. We will cover interceptors in a later chapter.

The Session Context

- The **SessionContext** interface provides an API for so that the implementation can interact with the container
- The implementation can obtain a session context reference via dependency injection


javax.ejb.EJBContext
getCallerIdentity():Identity
getCallerPrincipal():Principal
getEJBHome():EJBHome
getEJBLocalHome():EJBLocalHome
getEnvironment():Properties
getRollbackOnly():boolean
getTimerService():TimerService
getUserTransaction():UserTransaction
isCallerInRole(role:Identity):boolean
isCallerInRole(roleName:String):boolean
lookup(name:String):Object
setRollbackOnly():void
javax.ejb.SessionContext
getBusinessObject(busInterface:Class):Object
getEJBLocalObject():EJBLocalObject
getEJBObject():EJBObject
getInvokedBusinessInterface():Class
getMessageContext():MessageContext

Some of the methods in these two interfaces are not often used in EJB3 and are provided for backward compatibility with EJB2. These methods include:

```
getEJBHome()
getEJBLocalHome()
getEJBLocalObject()
getEJBObject()
```

Injecting the Session Context

- You can use the **@Resource** annotation to instruct the container to inject a SessionContext into a field
- @Resource can inject other types of references, for example a JDBC datasource

```
@Stateful
public class MyImplementation
   implements MyInterface
{
    @Resource
    private SessionContext ctx;

    . . .
}
```

Instead of writing the annotation, you can write a "resource-ref" stanza in the EJB deployment descriptor.

The container provides the session-context reference immediately after the instance is created. Thus, the reference is available for any business-interface method.

Stateful Beans vs Stateless Beans

- Stateful beans maintain state between methods, but are less efficient, since the container cannot use instance pooling
- Stateless beans are efficient, but require you to pass all necessary information as arguments (or have the bean do some sort of lookup)
- Stateful beans are more vulnerable to container crashes (conversational state will be lost)

Having a choice between stateless or stateful session beans and entity beans is a good thing. It lets you architect systems that use the best features of each kind of bean. The bottom line is that many applications that use EJBs will use some combination of each bean type.

Chapter Summary

In this chapter, you learned:

- About stateful session bean conversational state
- About the stateful bean lifecycle
- About activation and passivation