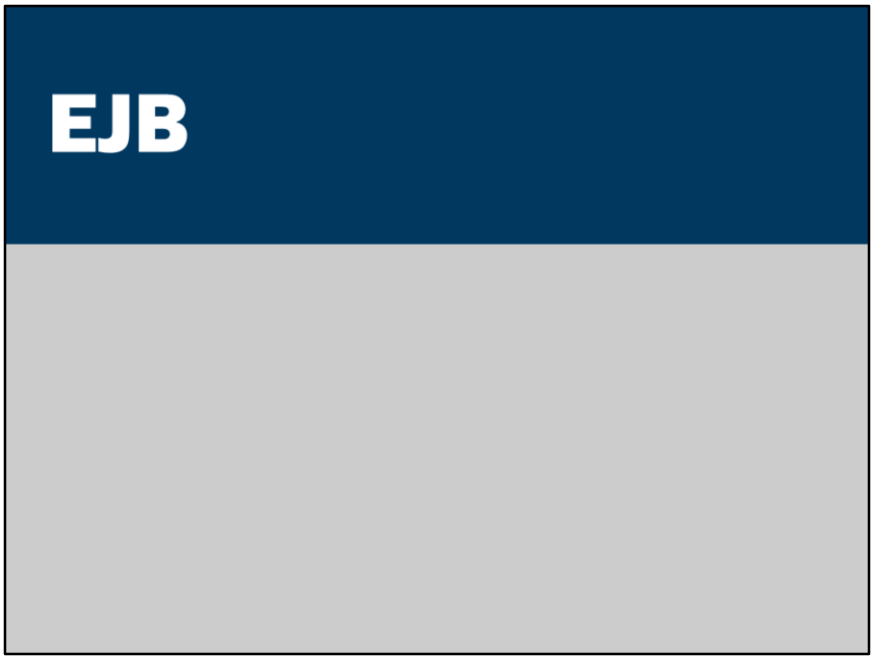
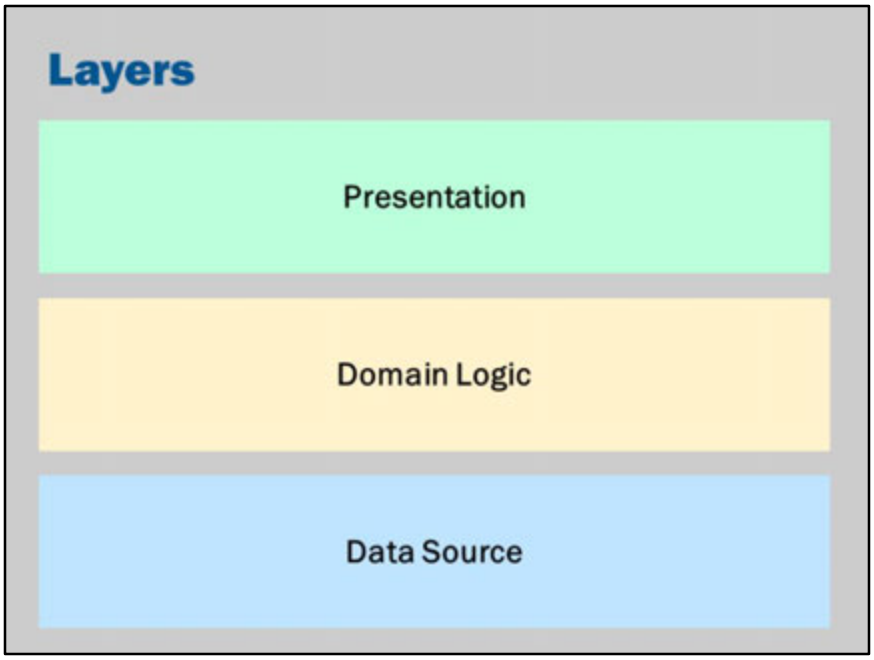


Enterprise Java Beans



Recall that in developing software applications, we can think of three major layers: Presentation, Domain Logic and Data Source.

Even though the borrowing power calculator has no database, it does have presentation and domain logic.

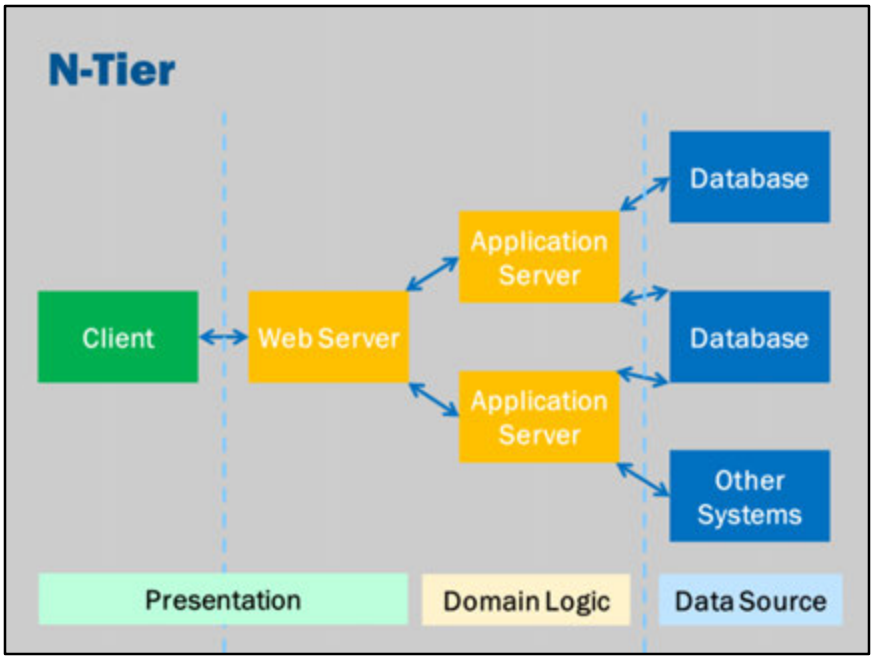
In lecture 3 we looked at various ways of structuring our application to improve the separation between layers.

However, even though our code was well designed, there were a number of things we could not do:

- Our domain logic cannot be run on separate machines
- Our domain logic must be packaged with the presentation technologies
- Our domain logic must be managed in the same environment and configuration of the presentation technologies

That is, so far, we have only been able to *reuse* our domain logic. We cannot *share* and *manage* it.

EJB technologies allow you to do this: share a module of domain logic.



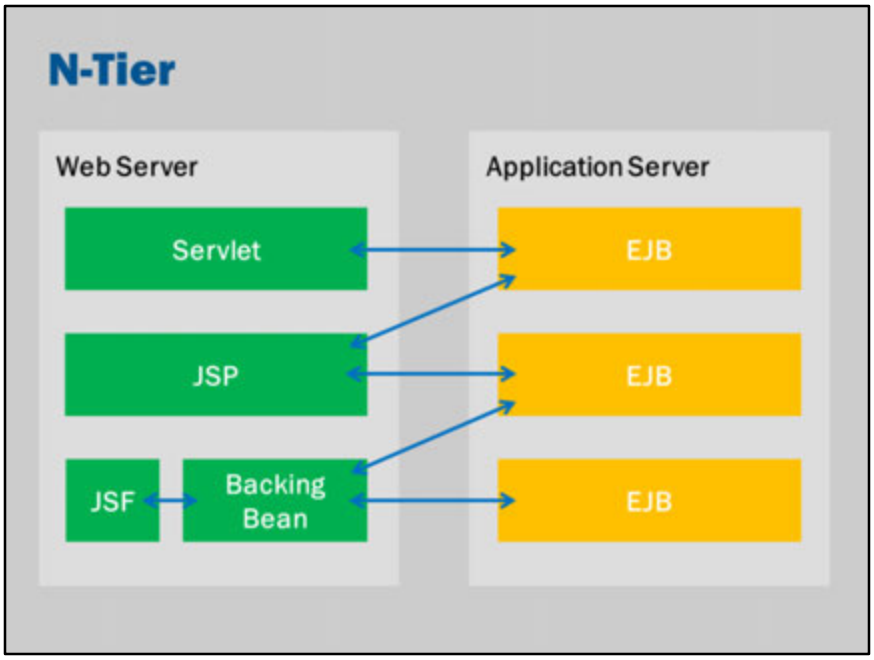
So far we have been building systems where the presentation and domain logic have been executing on a single tier.

For simple applications, this is not a problem.

However, as we want to scale and grow our application, we may need to start running the application over several machines.

The way we have been developing our software in this subject, so far, does not make any concessions for the code being run on multiple machines. We have not been able to call method or objects on other computers. We have only been able to create and call methods directly in the domain layer.

EJBs allow you to separate the presentation logic and domain logic into separate tiers, as in an n-tier architecture.

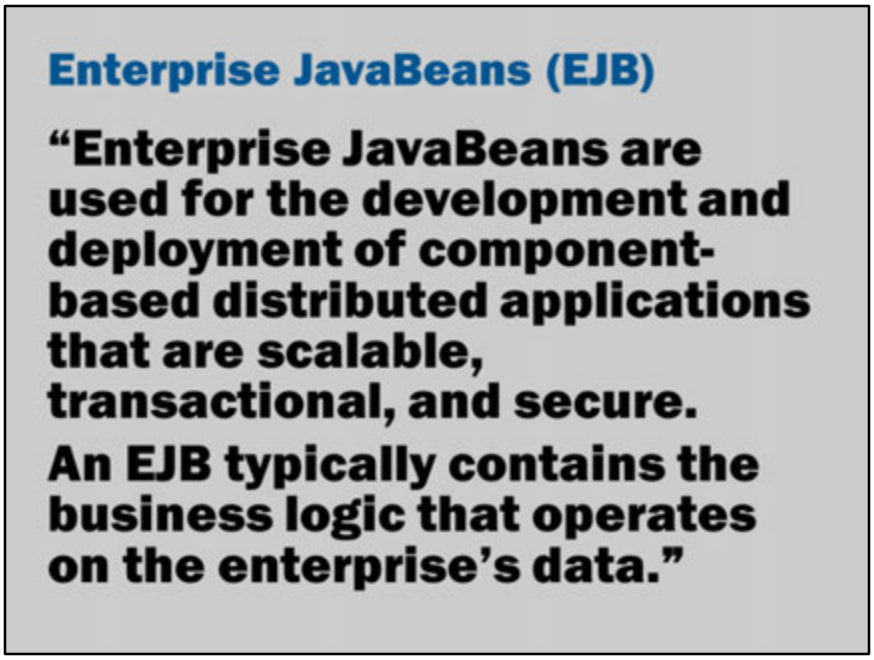


If we "zoom in" on the Web Server and Application Server, we see the situation that EJBs are designed to solve.

We have different presentation technologies in our Web Server.
The presentation technologies access the domain logic in the application server.

We have domain logic in our Application Server.
Our domain logic is implemented in a number of different components.
These components run on the application server.
The web server communicates to the application server via a network.

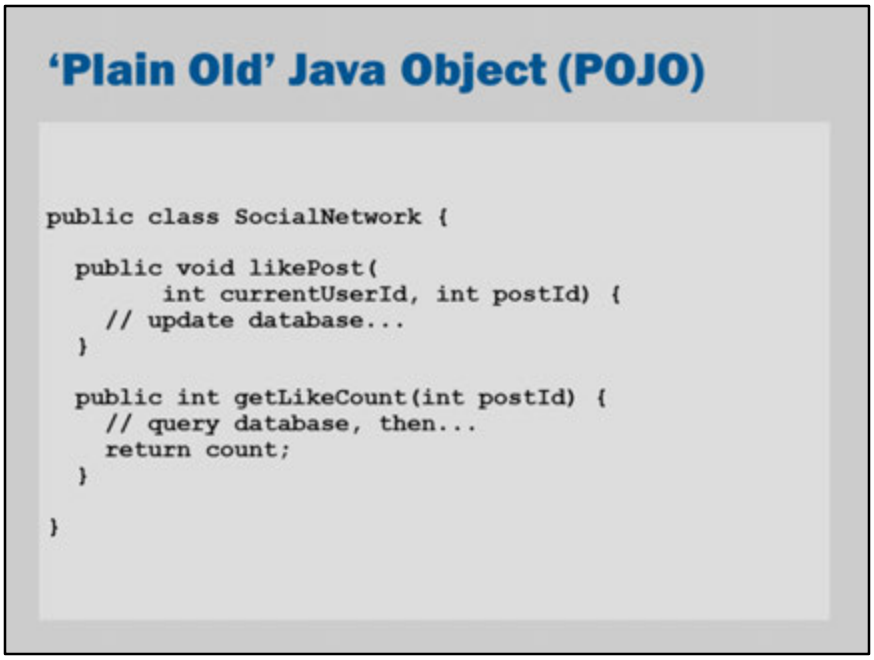
EJB technologies assist with managing the domain logic components and also providing the connectivity between the two tiers.



Arun Gupta (2013) *Java EE 7 Essentials*, p 145.

Key points:

- **Component based**: emphasizes separation of concerns and reusability
- **Distributed**: can be run over a network
- **Scalable**: can grow with more complex situations
- **Transactional**: ensures data correctness and works with database transactions
- **Secure**: it has built-in measures to ensure authentication and security



An ordinary java class is sometimes referred to as a "POJO".
"POJO" stands for "Plain Old Java Object".

In a social network app, this class represents the business logic.
This class lets you "like" and count the "likes" on a post.

EJB

```
import javax.ejb.*;

@Stateful
public class SocialNetworkBean {

  public void likePost(
        int currentUserId, int postId) {
    // update database...
  }

  public int getLikeCount(int postId) {
    // query database, then...
    return count;
  }

}
```

Converting the POJO into an EJB is a simple matter of adding an annotation.

This one annotation automatically gives you all the benefits of an Enterprise Java Bean.

With the annotation the class is now transactional, distributable and so on.

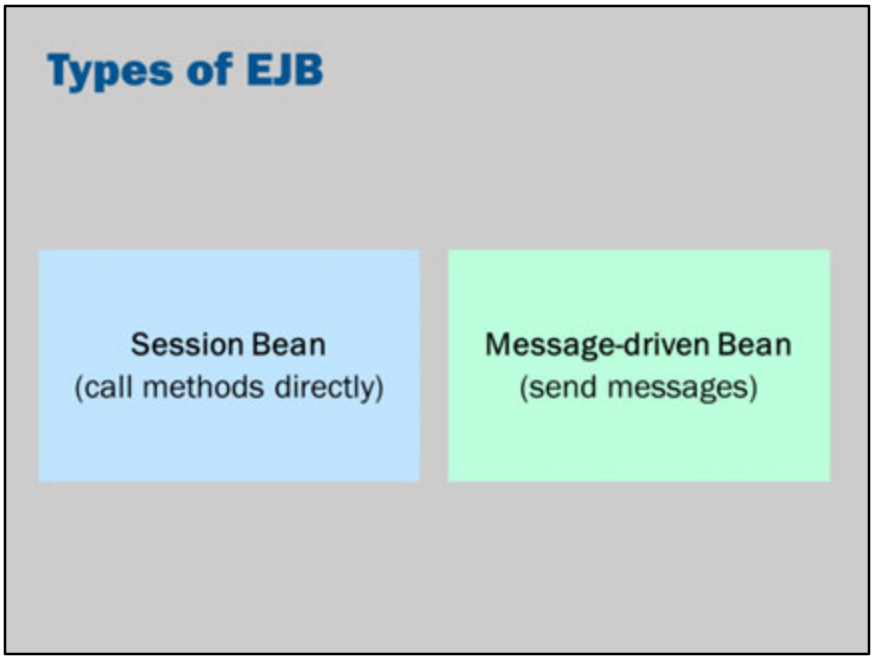
Note: I also renamed the class. This is not required. However, it is convention that the name of an EJB implementation should end with "Bean".

Once you have declared a session bean you can use it by dependency injection (you can also use JNDI lookup as well).

i.e., we might use the SocialNetworkBean in a backing bean that looks something like this:

```
@Named
@RequestScoped
public class SocialNetworkController {

  @Inject
  private SocialNetworkBean sn; // here we inject the bean

  … other private variables, setters and getters …

  public String like() {
    sn.likePost(currentUserId, postId); // now we use it
    return "liked_successful";
  }
```

```
}
```



There are two types of Enterprise Java Bean.

They have different modes of operation.

Session Beans work like ordinary Java classes and methods.
The Java EE application server adds all the EJB capabilities to your domain logic.

Message-driven beans do not work like ordinary method calls.
Instead, you send a message to a message-driven bean.
The message-driven bean will process the messages one-by-one.
You do not wait for the response.

The difference is analogous to making a phone call versus sending a letter.
Imagine you are trying to get a new credit card.
If you call up the bank, you will ask (and be asked) a number of questions, over the phone.
The application process will be completed while you are on the phone.
You can only call when telephone banking is available.

If, instead, you were to apply by post, you only need to put the letter in the letterbox.
The bank will receive the letter in a few days at some unknown time.
They will process the information on the letter and then your credit card will be approved.
You can post your letter at any time. You can drop it in the postbox at 1am if you wish.
It doesn't matter if the bank is open or closed.
You don't need to wait around for the response.
You can keep working on other things.

Key points

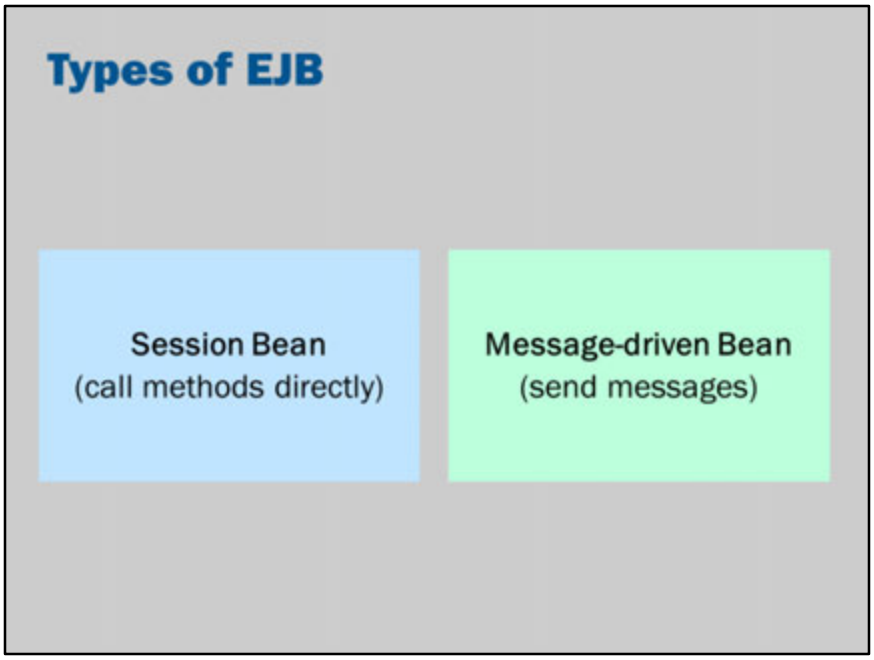
- An EJB is executed inside an application server that provides:
  - *Transactions*
  - *Scalability*
  - *Persistence*
  - *Security*
  - *Synchronization and Locking*
  - *Distributed object invocation*
  - *Monitoring and management*
- You focus on the domain logic, the container deals with the details!
- Two kinds of EJB:
  - *Session beans: executed immediately when called*
  - *Message driven beans: sent messages and not executed immediately*

Session beans

Characteristics

- An EJB is a component that implements the domain logic that operates on enterprise data
- An EJB is executed inside an application server that provides:
  - *Remote access*
  - *Concurrency*
  - *Transactions*
  - *etc...*

You focus on the domain logic, the container deals with the details!



There are two types of Enterprise Java Bean.

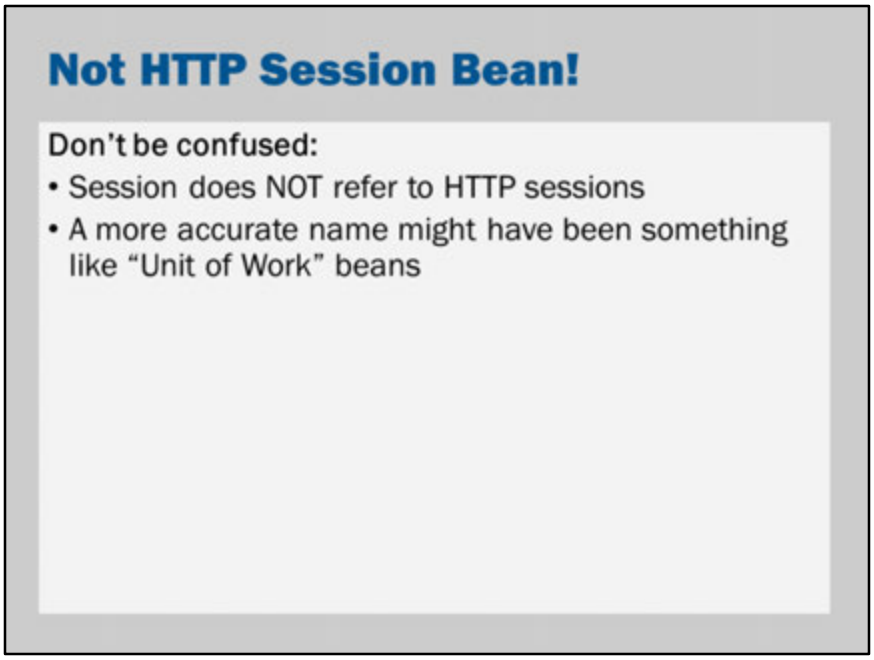
They have different modes of operation.

Session Beans work like ordinary Java classes and methods.
The Java EE application server adds all the EJB capabilities to your domain logic.

Message-driven beans do not work like ordinary method calls.
Instead, you send a message to a message-driven bean.
The message-driven bean will process the messages one-by-one.
You do not wait for the response.

The difference is analogous to making a phone call versus sending a letter.
Imagine you are trying to get a new credit card.
If you call up the bank, you will ask (and be asked) a number of questions, over the phone.
The application process will be completed while you are on the phone.
You can only call when telephone banking is available.

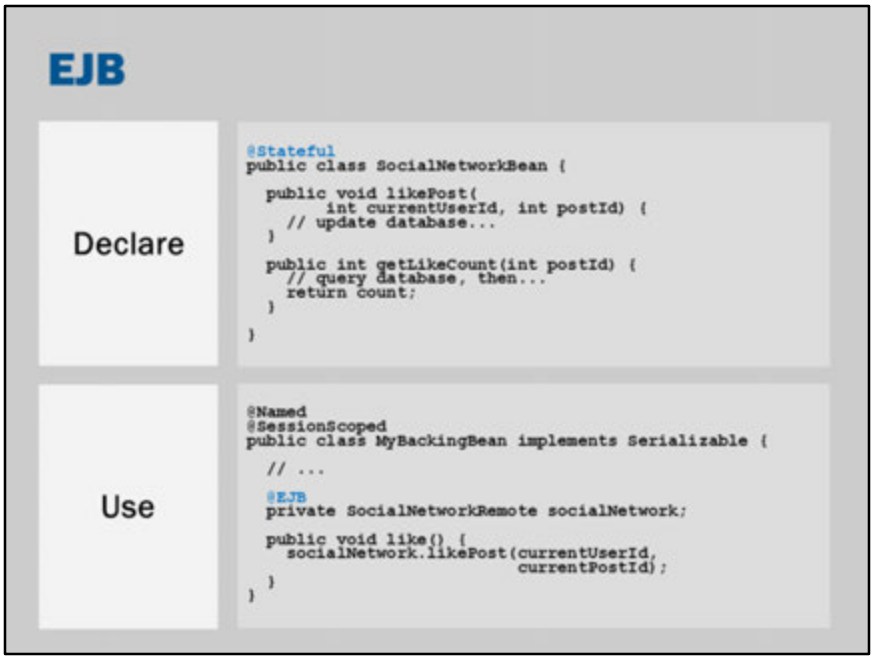
If, instead, you were to apply by post, you only need to put the letter in the letterbox.
The bank will receive the letter in a few days at some unknown time.
They will process the information on the letter and then your credit card will be approved.
You can post your letter at any time. You can drop it in the postbox at 1am if you wish.
It doesn't matter if the bank is open or closed.
You don't need to wait around for the response.
You can keep working on other things.



Session beans have nothing to do HTTPSession or @SessionScoped or Cookies.

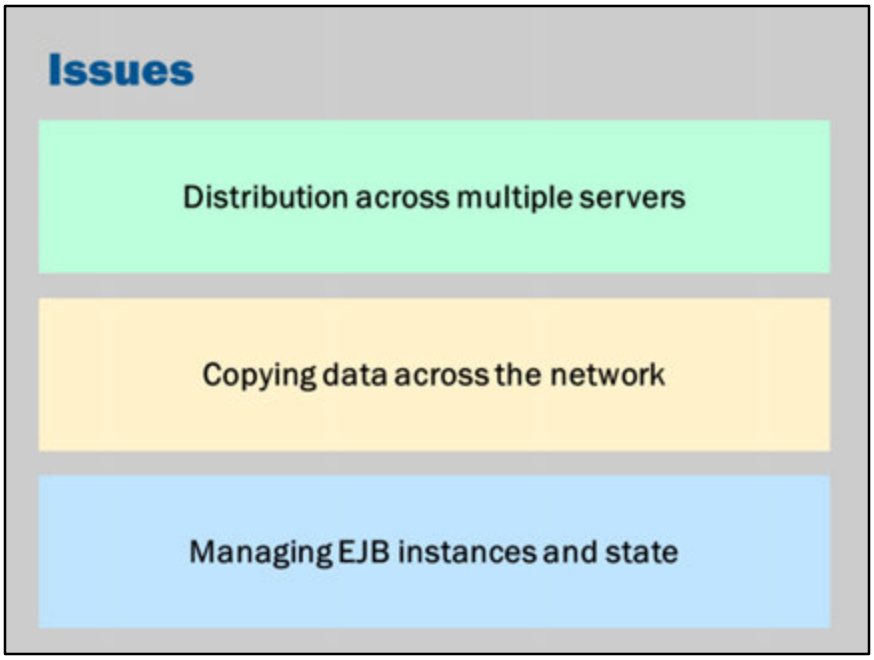
Think of it as a server-side unit of work.
Or, think of it as a connection between the EJB client code and the EJB server code.

Session beans



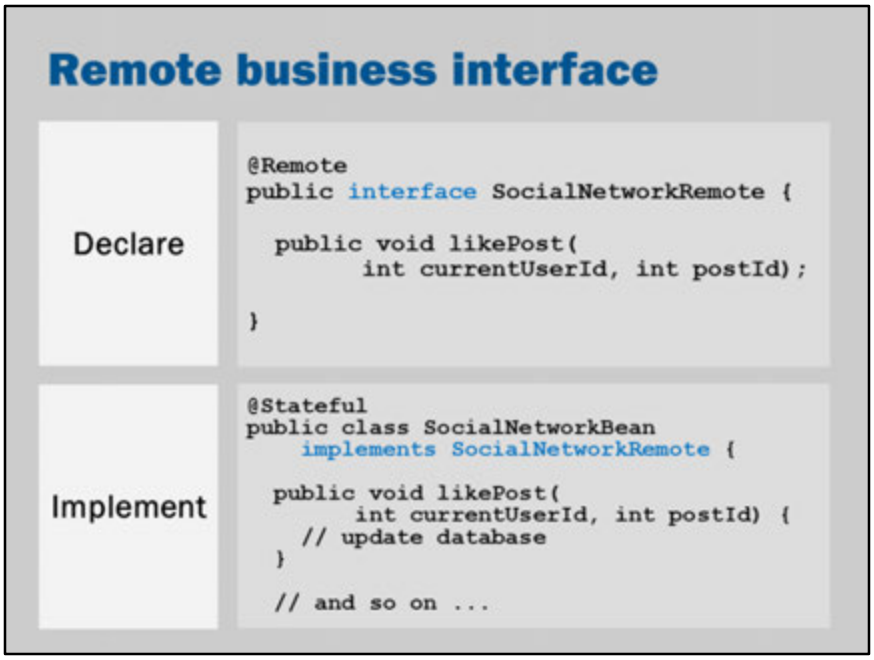
Declare a session bean with an EJB session bean annotation (@Stateful, @Stateless or @Singleton).

Then you can use an EJB simply by injecting it into your code using the @EJB annotation.



In our simple example of declaring and using an EJB, there are a number of issues that will come up when we try to deal with large-scale systems.

How can we the code by distributed?
How can the declaration and usage be handled from separate computers?
How can data be copied across the network?



In Java, an interface is a class without any implementation code. It is just a collection of method names.

You annotate an interface with @Remote to tell Java EE that it can be used from a remote machine.

Java EE will automatically generate a class that works something like the following:

```
public class BorrowingPowerRemote Proxy implements BorrowingPowerRemote {
  public double getLoanAmount(int loanTerm, double income, boolean isCouple) {
    open network connection to domain logic tier
    send method name "getLoanAmount"
    send parameters: loanTerm, income, isDouble
    receive response
    return response
  }
}
```

So that, when you use injection on the client…

@EJB private BorrowingPowerRemote remote;

…then Java EE will do something similar to setting:
remote = new BorrowingPowerRemoteProxy();

… and, on the system that runs the domain logic layer, Java EE will have a class that acts as a server for those network connections:

```
public class BorrowingPowerRemoteServer {

  private BorrowingPowerBean bean = new BorrowingPowerBean();

  public void run() {
    while true:
      wait for network connection
      receive method name
      receive parameters
      look up method name on bean
      call bean and get result
      send result back to client via network connection
  }
}
```

Using a remote interface

```
@Named
@SessionScoped
public class MyBackingBean implements Serializable {

  private int currentUserId;
  private int currentPostId;

  @EJB
  private SocialNetworkRemote socialNetwork;

  public void like() {
    socialNetwork.likePost(currentUserId,
                           currentPostId);
  }

  // rest of the class ...
}
```

From the client side, a remote interface works just like any other EJB, and works just like any other Java objects.

You inject the interface using @EJB, then you can call methods on the injected object as though it were an ordinary Java object.

Example remote proxy

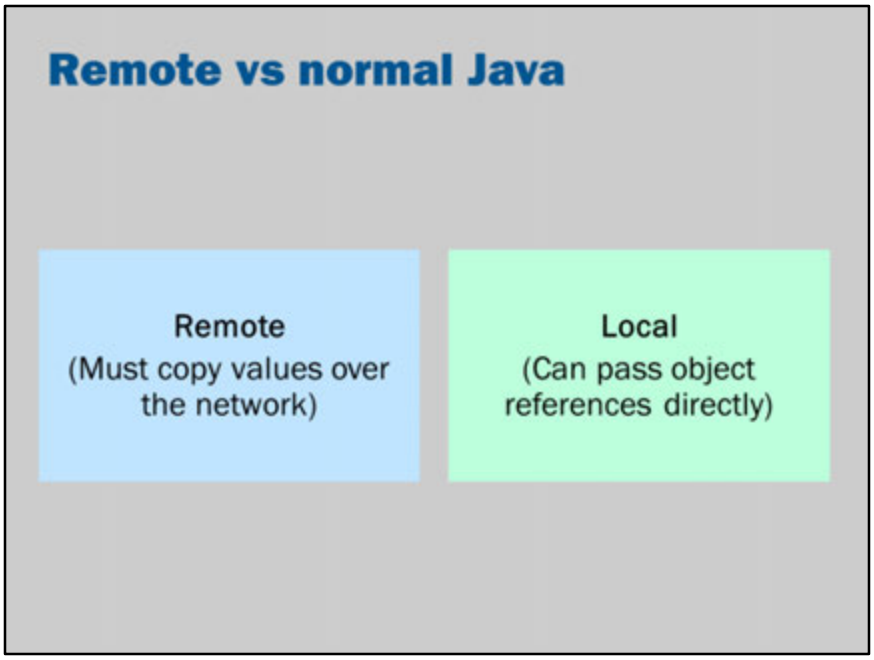
```
public class SocialNetworkNetworkProxy
    implements SocialNetworkRemote {

  private NetworkConnection server;

  public void likePost(
        int currentUserId, int postId) {
    server.send("likePost");
    server.send(currentUserId);
    server.send(postId);
    server.receive();
  }

}
```

Behind the scenes, the application server creates proxies.

Proxies are fake classes that implement the remote interface.
When you call a method on a proxy, then it will forward the data to the remote server.

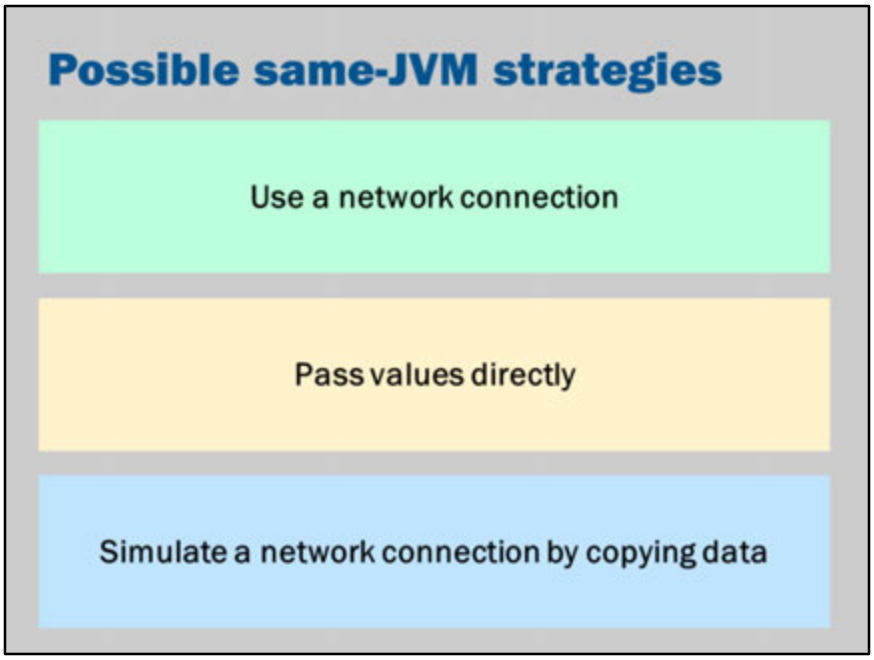
So, this code above is a highly simplified example of what a remote proxy might look like.

When you call likePost on the proxy, it connects to the remote server, and sends a command to call "likePost" on the real object, passing along the parameters currentUserId and postId.



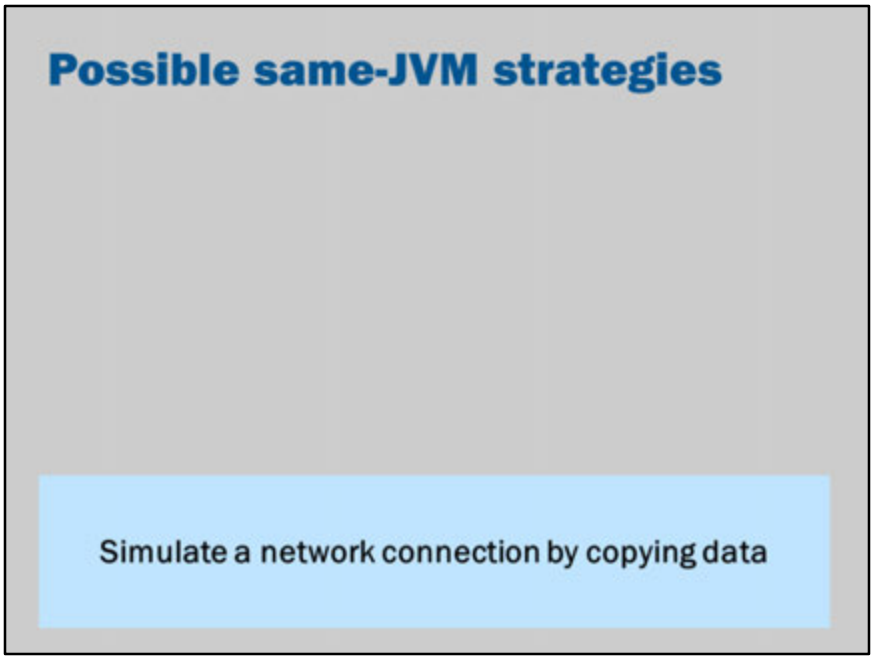
There's an important distinction between when you call a method on the local computer (e.g., in ordinary Java) versus calling a method over the network.

When you pass parameters to a method on the remote computer, the data needs to be transmitted and copied over to the other machine.
When you pass a parameter, you need to save the value of the object, and then reconstruct the object on the other end.

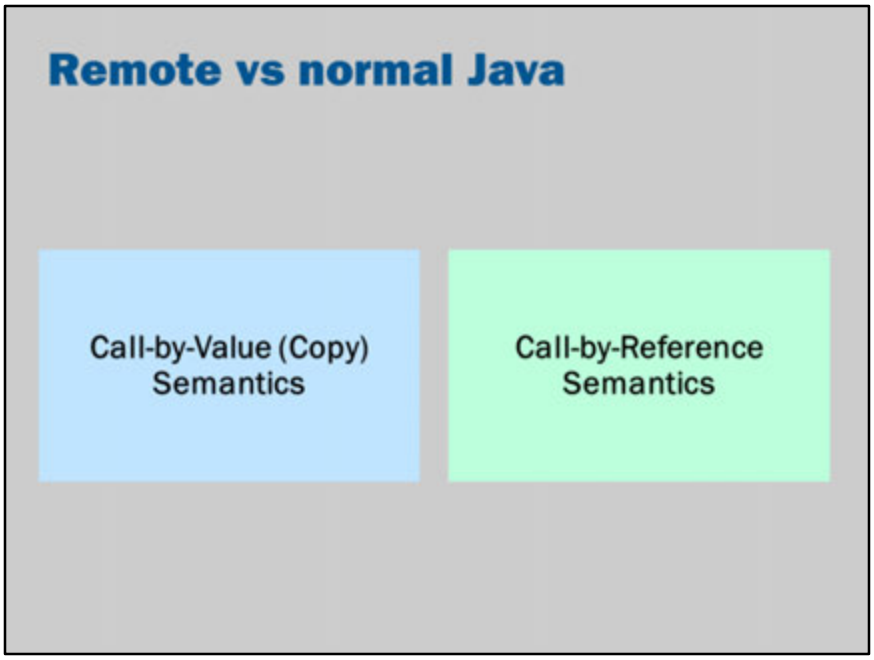


When calling remote interfaces from a local machine the EJB designers had three possibilities:

1. Use a network connection to connect to yourself.
   This guarantees that a remote connection on a remote machine works exactly the same as a remote connection from a local machine.
2. Bypass the proxy and pass the object directly.
   This is very efficient because there are no network overheads. However the downside is that passing values directly is very different to copying them (serialization) and creating new objects that are copies (deserialization).
   This could cause bugs: if you assume that the data has been copied over a network, then you might decide that it is safe to modify and otherwise corrupt the object that was sent (because it is just a copy). If the object was, in fact, passed directly, then the original object will be unexpectedly corrupted.
3. Bypass the actual creation of a network connection, but simulate the connection by copying data.



For the best compromise between reliability and performance, the EJB designers chose to require a Java EE application server to essentially simulate a network connection when you use a remote interface to call an object on the same machine.



When we send data over the network, it must be a copy (or a proxy).
You can't simply pass a pointer or reference to an object.

Consider the situation where I have a Data Transfer Object:

```
public class Person {
  private String firstName;
  private String lastName;
  … setters and getters …
}
```

And I pass this object to an EJB:

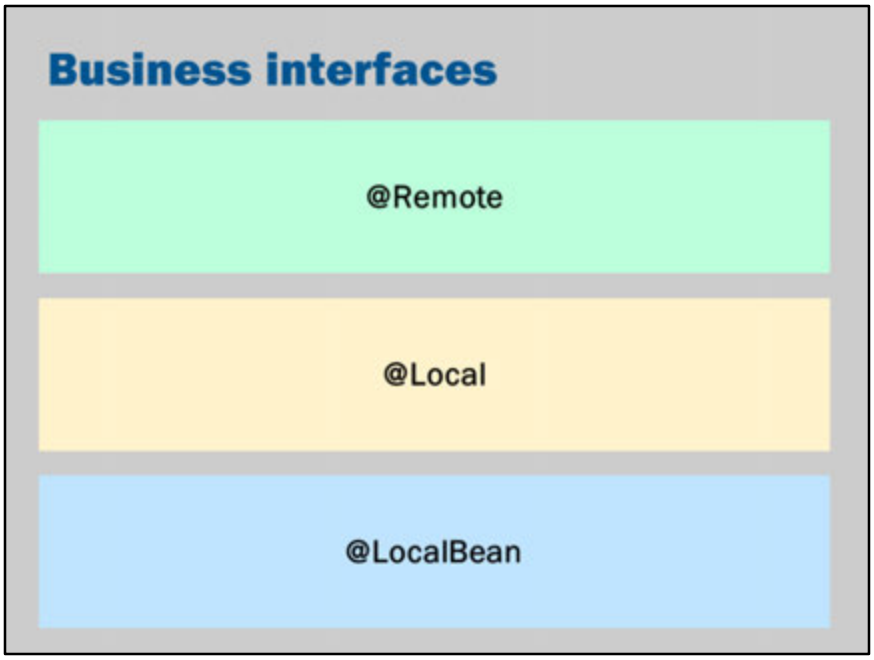
```
@EJB
private WaitingListBean waitingList;


Person myPerson = new Person();
myPerson.setFirstName("Carol");
```

```
myPerson.setLastName("Brady");
waitingList.addPerson(myPerson);
```

Then, to send myPerson over the network, Java EE must read the internal data, send it over the network and create a new object on the other end.
When the WaitingListBean receives the Person object, it will be a new version of the object.
The value of that object matters. Not specific reference to the instance.
That is, it uses call-by-value semantics.
If the WaitingListBean were to make changes to the Data Transfer Object (e.g., myPerson.setFirstName("Mike")), this information is not automatically reflected back on the client.
(If you wanted the client to see the change, you'd need to send it back in the return value.)

Call-by-value semantics apply even if the @Remote interface is running on the same machine.
Java EE pretends that the machines are separate, even if they are not.
Objects passed to @Remote interfaces will be copied, even if both sides are running on the same machine.

@Local and @LocalBean must always runs on the same machine.
This means that there is no need to copy data.
@Local and @LocalBean always use call-by-reference semantics.
i.e., @Local and @LocalBean work like ordinary Java classes.



The easiest way to understand EJBs is to imagine that the presentation logic and domain logic are running on separate computers.

However, in practice, actually opening the network connections is slow and resource intensive.

The performance costs of this network connection are unnecessary when both the caller and implementation are running on the same computer (i.e., the same Java Virtual Machine).

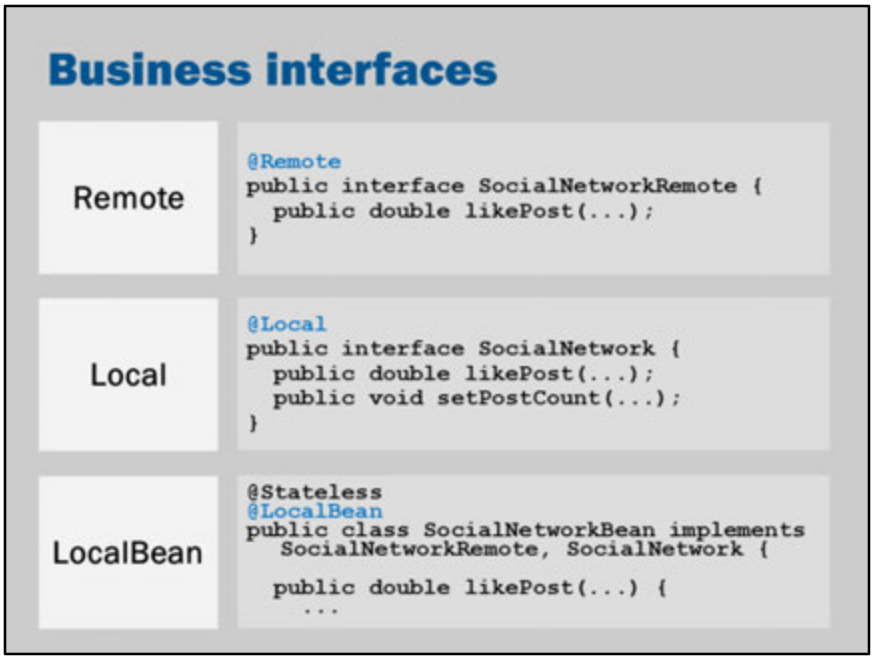
This creates a potential for optimization that Java EE exploits with @Local and @LocalBean annotations.

An interface annotated with @Local works similar to @Remote interfaces.
However, both the caller and implementation need to be running on the same machine. This is assumed and required for local interfaces.

@LocalBean is similar to @Local.
However, an @LocalBean means that the implementation class can be used directly.

There is no need to access the implementation via interfaces.
Behind the scenes, Java EE will subclass your bean to achieve the effect of creating a proxy.
Thus, using @LocalBean essentially saves the hassle of needing to implement separate interfaces.
However, @LocalBean only makes sense in simple scenarios.
Implementing a full interface (@Local or @Remote) does tend to result in a better design.

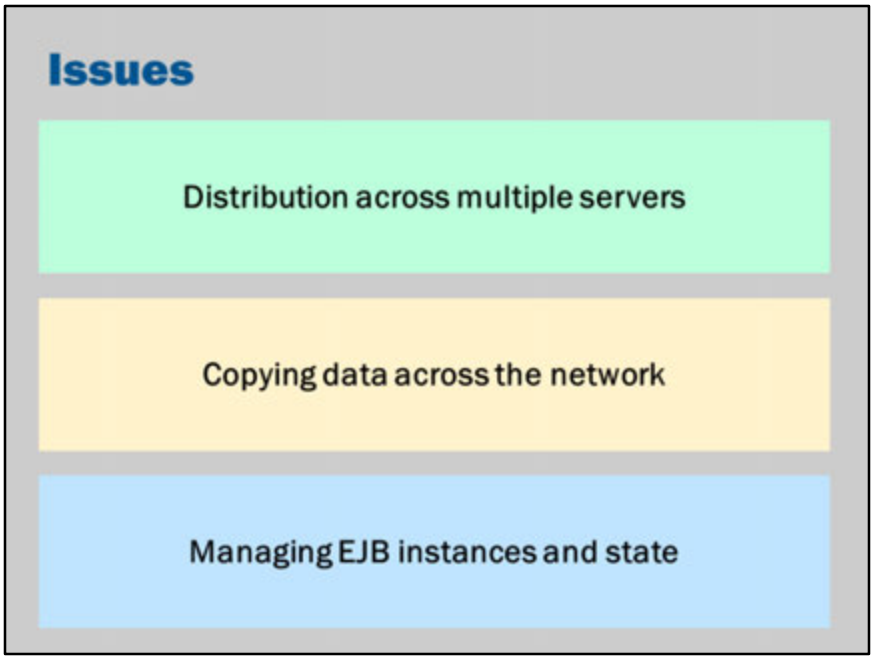
Finally, @Remote *can* also be used on the same machine.
Java EE knows that the remote network connection isn't required.
It will bypass the costs of establishing that connection.
However, it will still do its best to "pretend" to have a network connection.
This means that when you pass an object to a function, it will not pass the object directly.
It will copy the object and give the copy to the target function.
This copy is necessary because when you send an object over the network, it generally must also be copied.
Within a virtual machine this effect is simulated by copying.



The annotations are used by adding them to the start of an interface (in the case of @Remote or @Local) or the start of a concrete class (in the case of @LocalBean).

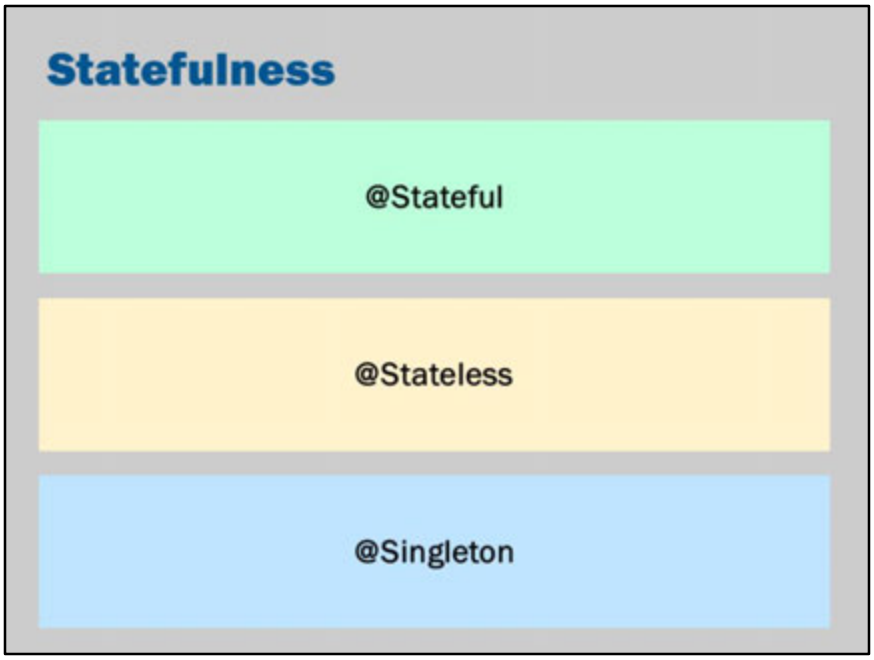
You can combine the annotations.
A bean can have a local interface and a remote interface, as well as also being a local bean. You can use all three annotations, or just the ones that you need.

Note: If you do not have @Remote, @Local or @LocalBean, then the default is just @LocalBean.



So, @Remote solves the problem of distributing and copying data across the network. @Local and @LocalBean also help manage the complexity and performance impacts of using remote interfaces.

However, we haven't addressed how the application server manages the instances of the server-side Enterprise Java Beans themselves.
That's what the next slides will cover…



In computing, state usually refers to systems that have some kind of "memory" of the past.
In other words, the result of an action depends on the previous actions that have occurred in the past.

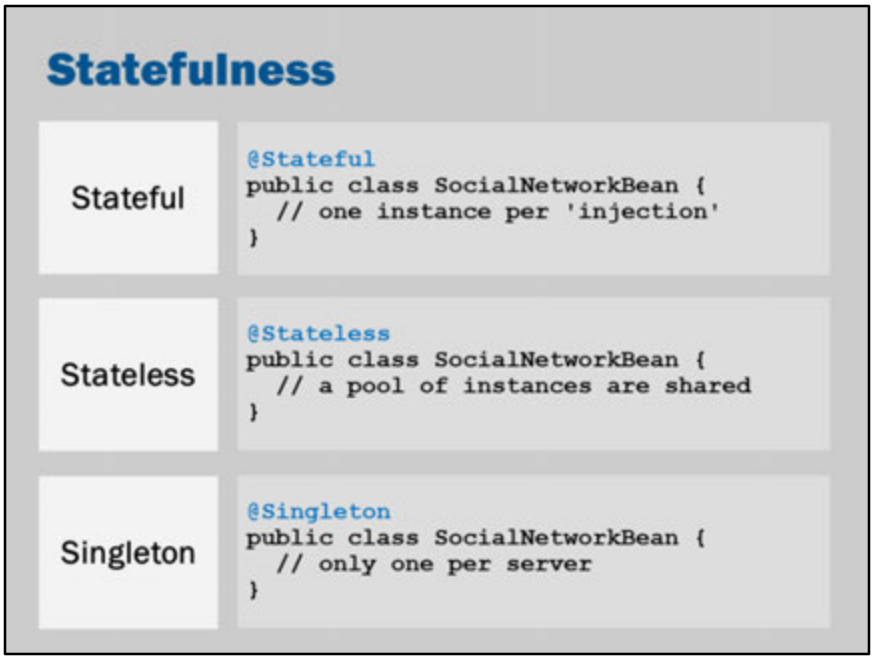
In the case of EJBs, state refers to whether the bean stores information between requests.

A Stateful EJB is like an ordinary Java class.
A separate instance is created each time an EJB is created by injection or JDNI lookup.
The local variables of the instance can be used to store ongoing information.

A Stateless EJB is more like a collection of independent methods
The Java EE server might create a handful of stateless EJBs.
However, any given request can go to any of the stateless EJBs.
In fact, invoking the same method on an "EJB" twice in a row, could be directed to two completely separate instances of the implementation bean.

A Singleton EJB exists only once in an application.

All requests go to the one instance of the bean.



The different kinds of beans correspond to different annotations.

Statefulness

Stateful session beans:

- Most similar to ordinary Java programming
- Inefficient

Stateless session beans:

- All calls are handled by a small pool of reusable beans
- Efficient but requires more care when programming

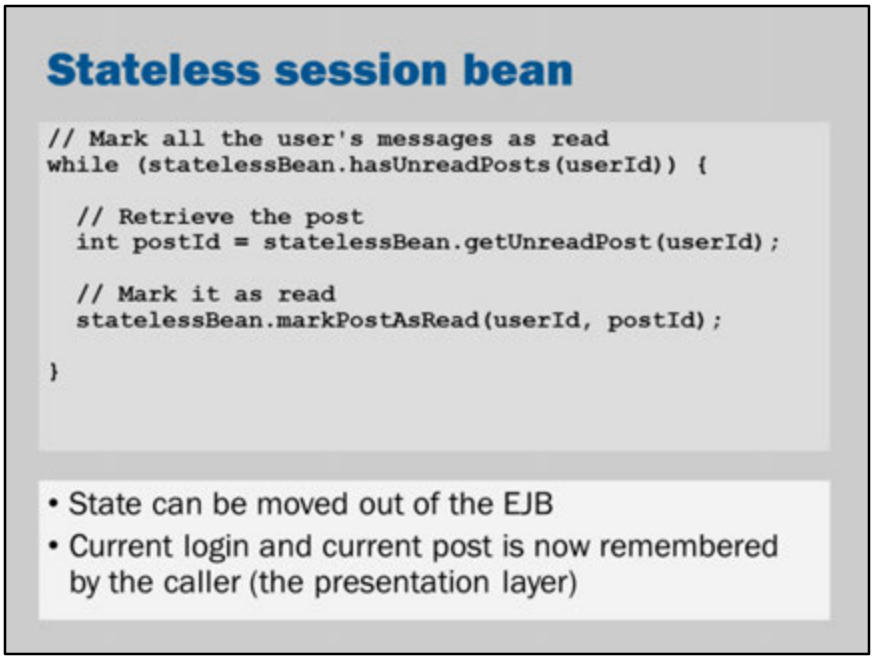
Singleton session beans:

- All calls are handled by just one bean instance
- Efficient but best suited to situations where one instance is explicitly required

A stateful session bean has an ongoing state (i.e., an ongoing connection to the server).
This means that it is possible design your bean so that it needs to remember some state.

In the example above, the EJB needs to remember who the current user is and what the current post is.
That way, when you call markCurrentPostAsRead, it will mark the currently selected post as being read by the currently selected user.

This means that the state on the server needs to include: current user and current post.



We can eliminate the need for state to be stored on the server.
Instead, we can expect that the presentation logic (i.e., the client) needs to remember the state.

So, the EJB has methods to check if there are unread posts for a user, to retrieve the unread posts by the user and the mark a particular post as being read by a particular user.

Notice that this time there's no dependencies between the methods.
The methods are self-contained.
There's no need to set the current user or set the current post because the required information (i.e., the required state) is passed along with every request.

That stateless EJB could also be simplified even more, by just moving the logic into the domain logic in a method of the EJB that does multiple actions.

Singleton EJB

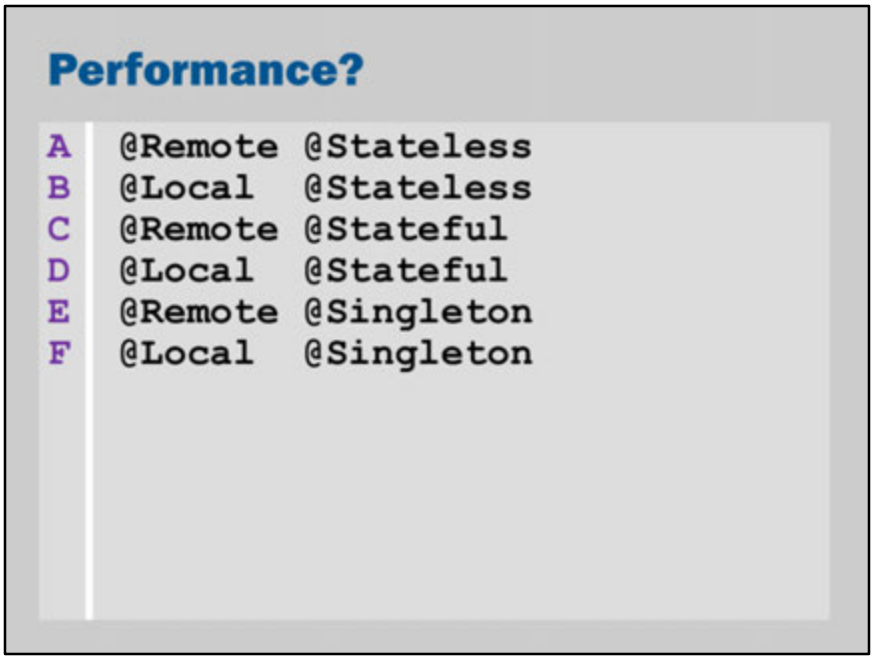
Some use cases:

- Single state shared across the entire application
- Connections to an external service that can only be used sequentially
- Running code on startup (`@Startup` annotation)

Choosing a session bean type

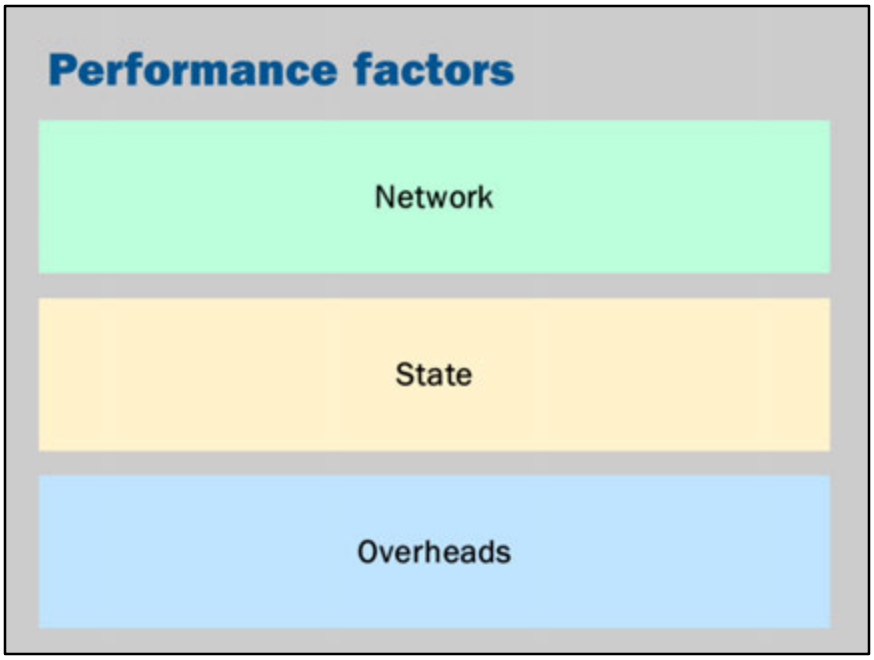
- Try stateless session beans first
- Use singletons if you need code to be run on start up
- Use stateful session beans with caution

Performance



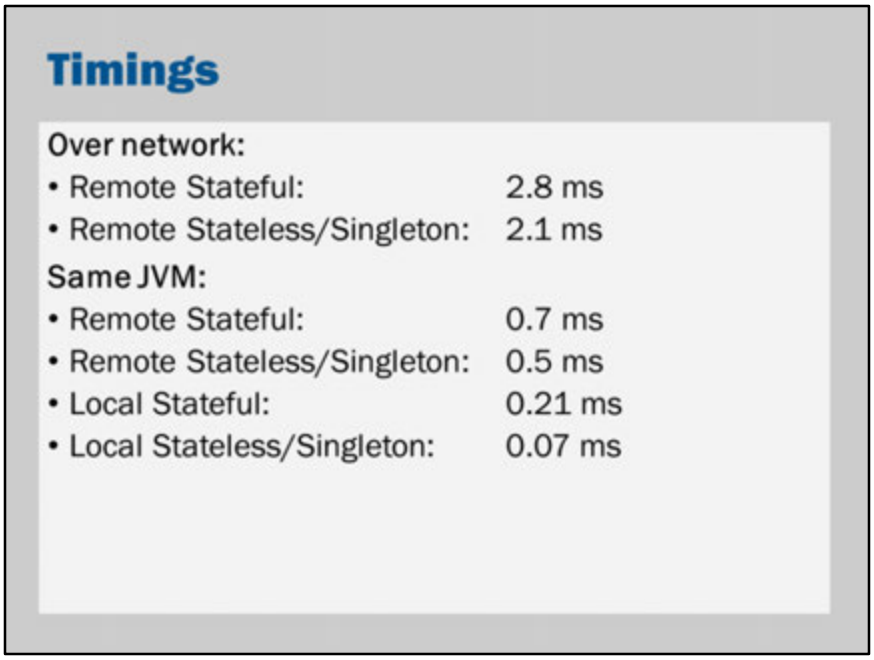
Which do you think is fastest?

Can you put them into order from slowest to fastest?



Factors that affect performance:

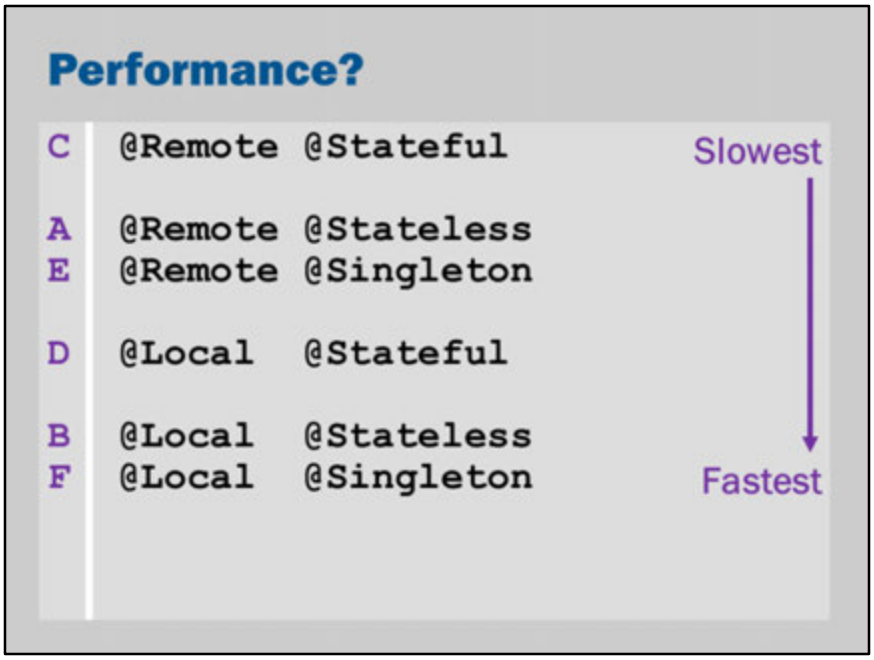
- **Network** (the costs of establishing the network connection and sending the data over the network: throughput, latency)
- **State** (the need for the application server to remember the individual details of each separate stateful bean, this takes memory/disk space)
- **Overheads** (other overheads relating to transactions, locks and also things like emulating call-by-value semantics when using a @Remote interface on a local machine)



Here are some timings calculated on my computer. The timings are calculated from many trials involving:

- Getting a reference to an EJB
- Calling two functions, the second being a "close" (@Remove) function in the case of a stateful bean.

Clearly, even though the it was running on the one machine, the operating system overhead associated with establishing a network connection takes a fair bit of time!



Which do you think is fastest?

Here's a typical ranking of the performance:

- C (slowest)
- A/E (slow)
- D (fast)
- B/F (fastest)

Stateless and singleton beans are roughly equivalent in speed.

When I last tested the performance, the *stateless* beans appeared to be just slightly faster than singleton. However, the difference was very small. It could have just been measurement error.

Key points

Remote interfaces:

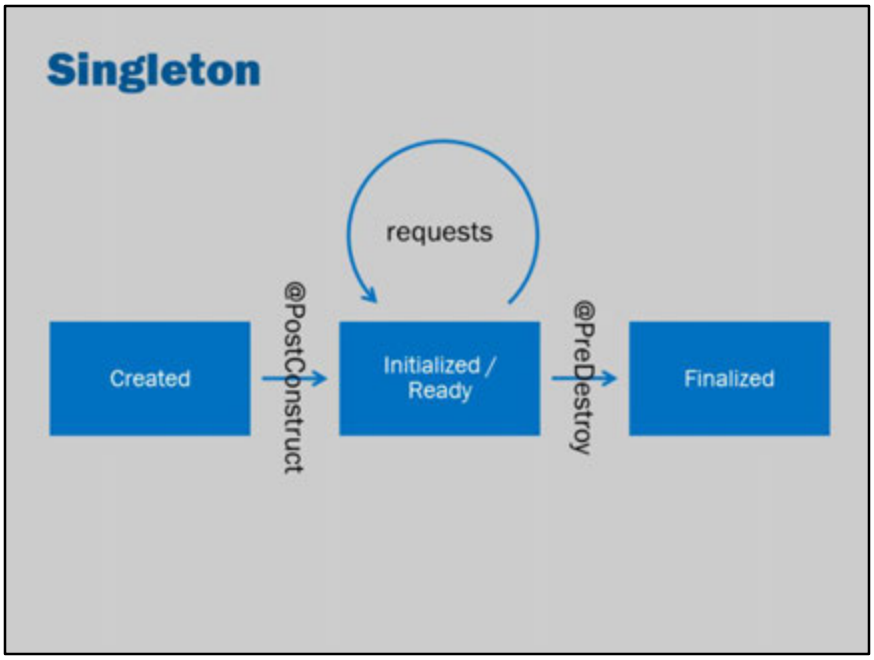
- Use call-by-value semantics
- Can be used remote or locally

Local and LocalBean interfaces:

- Use call-by-reference semantics
- Can only be used locally

Prefer using stateless session beans

EJB Lifecycles



The lifecycle of an EJB is controlled by the container.

1. An EJB is constructed (i.e., new MySessionBean()) but it will not receive any requests until after it has been initialized.
2. It is initialized by calling the method annotated with @PostConstruct (if one exists).
3. After @PostConstruct, the EJB will be able to respond to calls from clients.
4. The container will call @PreDestroy when it does not need to use the EJB anymore.
5. It will not receive any requests after it has been destroyed.

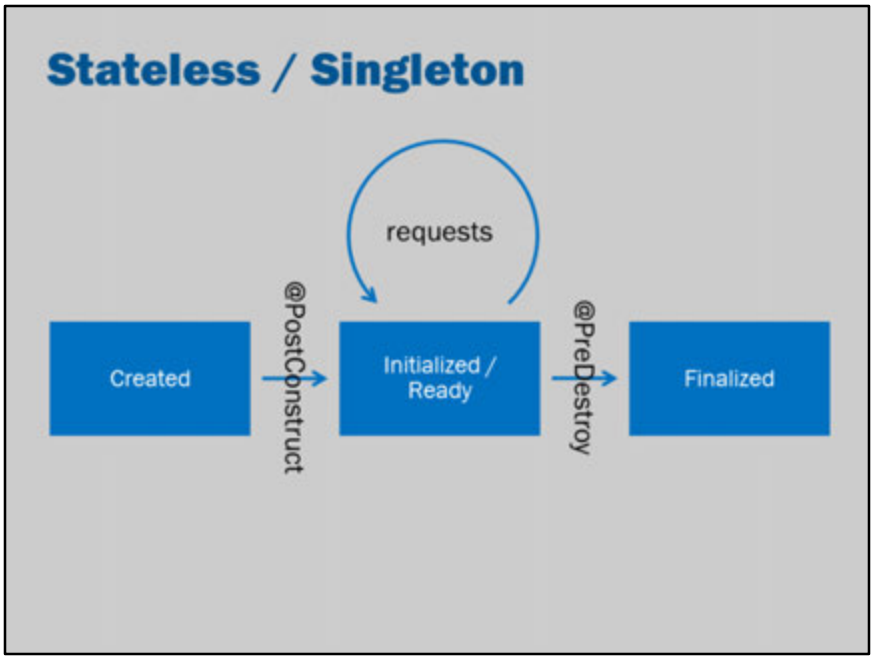
These states are depicted in the diagram above.

Why do we need a lifecycle?
Why not just use the constructor and finalizer that can be used on any class in Java?

The reason is that the application container needs to start up cleanly and safely.
It may need to instantiate the object and inspect the object early.
However, it can only initialize the session bean when other services in the container

are ready (e.g., it may need to wait for the database to be ready before allowing the session bean to initialize).

You cannot assume that any of the application server's services are available in the constructor – any initialization that depends on databases or other application server services should be done in a method annotated with @PostConstruct.



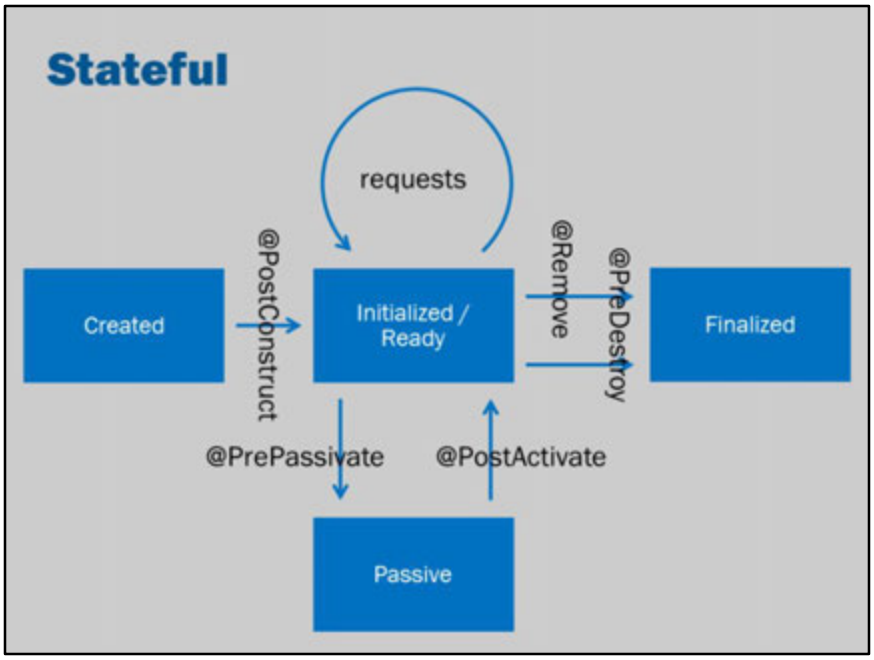
The same lifecycle applies to stateless session beans. The only difference is that the application server might create more than one instance

Lifecycle example

```
@Stateless
public class SocialNetworkBean {

  @PostConstruct
  public void init() {
    // set up and create connections
  }

  public void likePost(...) {
    ...
  }

  @PreDestroy
  public void close() {
    // clean up and close resources
  }


}
```

Methods annotated with @PostConstruct are called just after the SocialNetworkBean has been created.
This is where you would your own code to initialize your EJB.
Use @PostConstruct instead of the object's constructor because you be sure that the execution environment has been set-up properly yet in the constructor.

The same thing applies to @PreDestroy.
Annotate a method with @PreDestroy when you need to write clean-up code that is executed when the application server is finished with your EJB.



A Stateful bean has a few extra states and operations.

Stateful beans use up memory. If the application server needs to make space, it can save the state of some of those stateful beans to disk. This is called *passivation*.

The additional "Passive" state refers to beans that have been stored on disk or a database.
Methods that have been annotated with @PrePassivate or @PostActivate are called by the Java EE server before being saved and after being loaded from disk (respectively).

The @Remove annotation can be added to any method.
If a method has this annotation, then the Stateful EJB will be destroyed after such a method has been called.

Lifecycle example

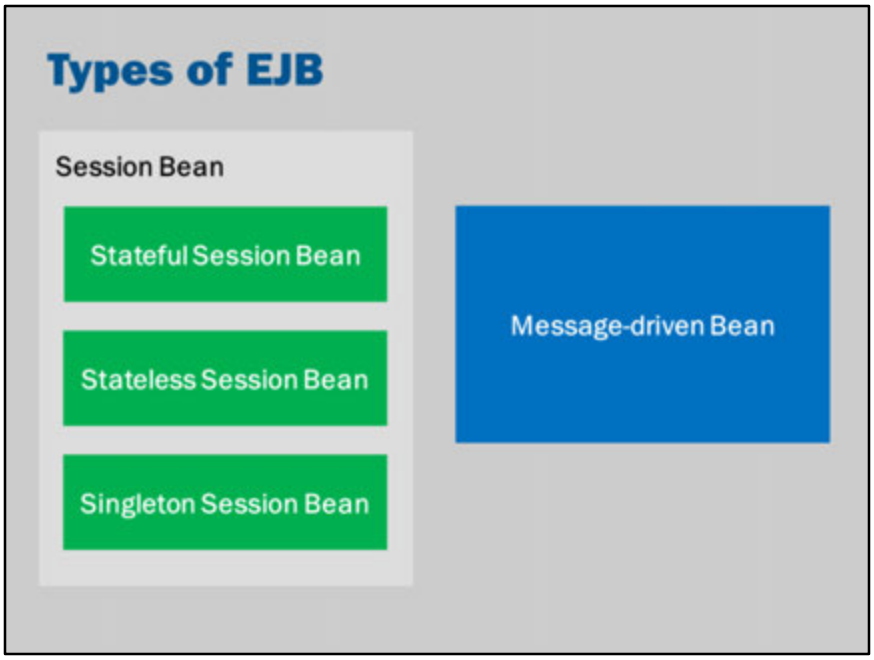
```
@Stateful
public class SocialNetworkBean {

  @PostConstruct
  public void init() {
    // set up and create connections
  }

  public double likePost(...) {
    ...
  }

  @Remove
  public void end() {
    // instance is now no longer needed
  }

  @PreDestroy
  public void close() {
    // clean up after user
  }

}
```

Bonus slides

Characteristics

- An EJB is a component that implements the domain logic that operates on enterprise data
- An EJB is executed inside an application server that provides:
  - *Transactions*
  - *Scalability*
  - *Persistence*
  - *Security*
  - *Synchronization and Locking*
  - *Distributed object invocation*
  - *Monitoring and management*

You focus on the domain logic, the container deals with the details!



Looking at the Session Bean…

There are three types:

- **Stateful**: Stores state associated with each instance. This is similar to a network connection. One instance of a stateful session bean will be created for each user.
- **Stateless**: Stores no state associated with users/instances. This means that Stateless session beans can be reused by multiple users.
- **Singleton**: Has only one instance in the entire application. This is similar to how a Servlet works: one instance handles all requests.

Benefits

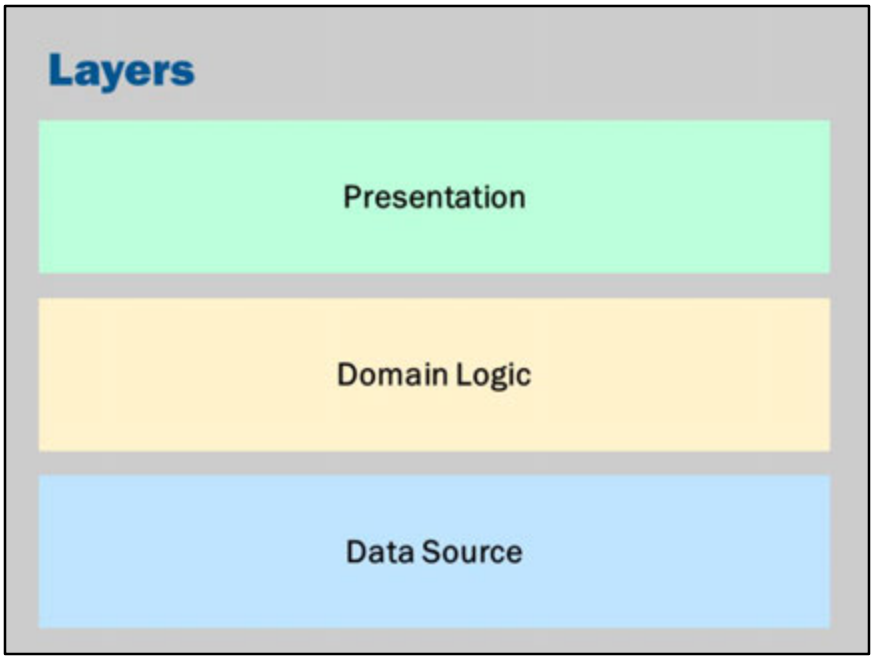
Transactions:
- Container automatically starts a (configurable) transaction with each invocation

Scalability:
- Instances automatically pooled and reused

Security:
- Methods can be configured for access only by authenticated clients

Synchronization:
- Various locking modes can be configured (locked by default)

Monitoring and management:
- Configure and control from administration console



The easiest way to understand EJBs is to imagine that the presentation logic and domain logic are running on separate computers.

The domain logic is implemented in an EJB.
The domain logic implements an interface.
The interface defines the functions that the EJB makes available.

The presentation logic then accesses the domain logic *via* that interface.

On the client, Java EE automatically generates a class to implement the interface.
That class runs on the presentation tier.
That class has code to automatically connect to the domain logic tier.
The domain logic tier decodes the request and passes it to the appropriate instance of the EJB.

This means that the presentation tier only needs access to the interface.
The implementation of the business logic is executed and stored only in the domain logic tier.

@Stateful or @Stateless?

Stateful session beans:

- The bean represents an interaction with a specific client
- The bean *needs* to hold information about the client across method invocations
- The bean mediates between the client and the other components of the application, presenting a simplified view to the client
- Behind the scenes, the bean manages the work flow of several enterprise beans

@Stateful or @Stateless?

Stateless session beans:

- The bean's state has no data for a specific client
- In a single method invocation, the bean performs a generic task for all clients
- The bean implements a web service

@Stateful or @Stateless?

Singleton session beans:

- State needs to be shared across the application
- One bean needs to be accessed by multiple threads concurrently
- Tasks need to be performed upon application startup and shutdown (@Startup)
- The bean implements a web service

Get current inventory levels
This is probably stateless. If we are checking the number of copies of the book "Beginning Java EE 7" in a book store, it probably does not depend on who we are or what the previous requests were.

Submit expense report
This *might* be stateful. Submitting an expense report could consist of several stages that need to be performed in sequence. Alternatively, if you need to submit many expense reports in a small amount of time, it might make sense to be able to remember common information such as the username and reimbursement details.

However, it is more likely to be stateless. You would construct an expense report in the user interface but submitting it is done in one command.

Add new customer
This is probably stateless. Adding a customer to a database could be done in a single command without needing to refer to previous information.

Reserve flight

This might be stateless or stateful.

Perhaps reserving a flight is a two-stage process. First, you check the seat availability and this results in the seat be reserved for 10 minutes. Then you have 10 minutes to pay for that seat before it is released for booking to another user.

Alternately, perhaps checking seat availability does not guarantee availability. Checking availability and reserving may be completely independent. This might mean that it makes sense to keep them as entirely independent, stateless operations (thus freeing up any resources required to maintain state).

Finally, even if the availability and reservation is completely separate, in some circumstances it might make sense to retain statefulness. Perhaps sending the flight and time details uses lots of data on a slow network. This data would be the same for checking availability and making the reservation. Retaining state on the server could help minimize some of the network costs.

Shopping cart
Shopping carts are often given as an example for why @Stateful beans are needed. Adding items to a shopping cart is certainly an ongoing process that does involve state.

However, in practice this may not be ideal.

The shopping cart concept may be a presentation layer issue. Tracking shopping carts over time should perhaps not be part of the domain logic. If we consider another, radically different, user interface we might not have a shopping cart. For example, we may just be buying a list of items that are sent in one order. That is, the relevant domain logic command is a *buy* or *order* function, not the creation of a shopping cart. The state is stored in the presentation layer (i.e., using cookies and @SessionScope) and the domain logic is stateless.

If the shopping cart is part of the domain logic, then it may still make the most sense to make it stateless. Consider shopping carts on amazon.com. You can add an item to your cart today but come back in a year's time to check-out your purchase. This wouldn't make sense as an EJB because the original EJB-client would be long gone and the Stateful bean would have timed out long before the year passes. Adding to a shopping cart might be a simple stateless operation: the real state is being stored in the database (not in a stateful bean).

This is ultimately a design decision you will need to make for yourself. Consider the trade-offs between simplicity and the cost of storing state. Consider also whether

building a shopping cart relates to an ongoing but time-limited stateful unit of work, whether the shopping cart is a presentation layer issue or if the shopping cart operations are simply changes to a database but that require no ongoing state in the domain logic layer.

Screen-by-screen checkout process

Much the same discussion with the shopping cart applies to screen-by-screen checkout processes.

The choice of individual screens may be a presentation layer issue: the actual transaction may make most sense as a single stateless action at the end of a sequence of presentation-layer actions.