COMS W4156: Advanced Software Engineering

Prof. Gail Kaiser

Topics covered in this lecture

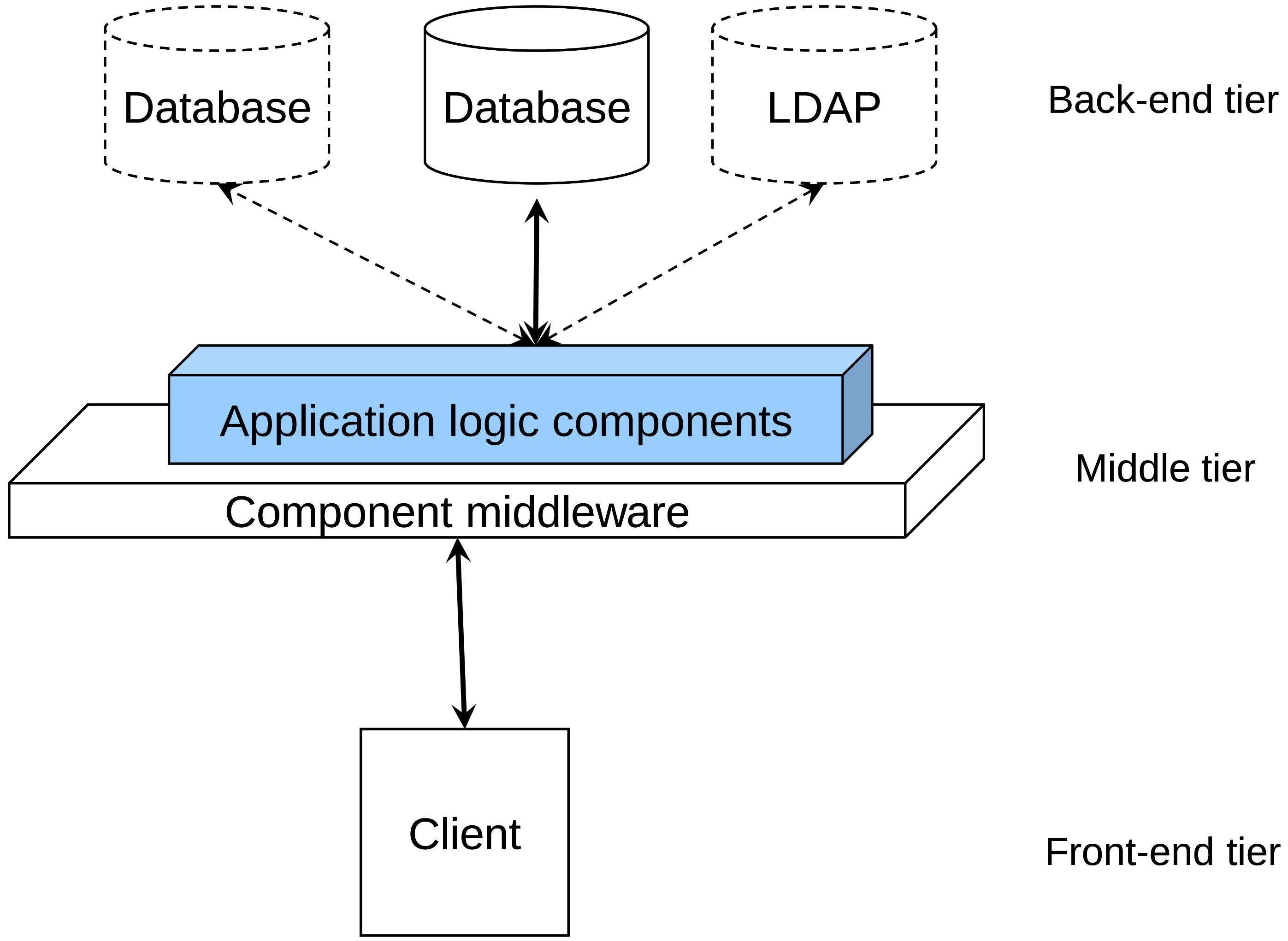
- EJB = Enterprise Java Beans

Enterprise Java Beans (EJB)

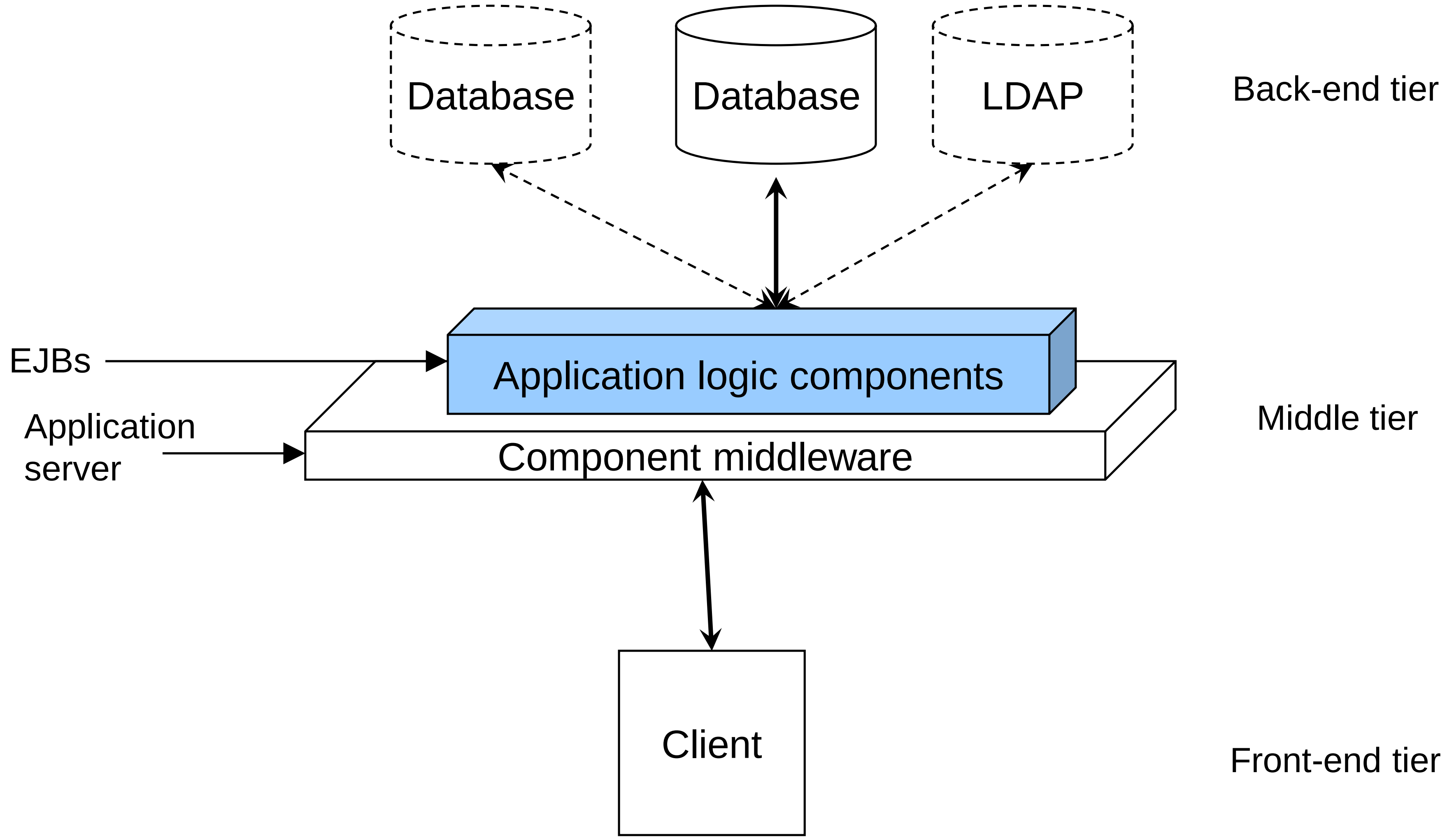
EJB Goals

- Standard component architecture for building distributed business applications in Java
- Interoperability between enterprise beans and other Java Platform Enterprise Edition components, as well as non-Java applications
- Compatible with other Java APIs and with CORBA protocols
- Follow the *Write Once, Run Anywhere* philosophy of Java - an enterprise bean can be developed once and then deployed on multiple platforms without recompilation or source code modification
- Define the “contracts” that enable tools from multiple vendors to develop and deploy components that can interoperate at runtime

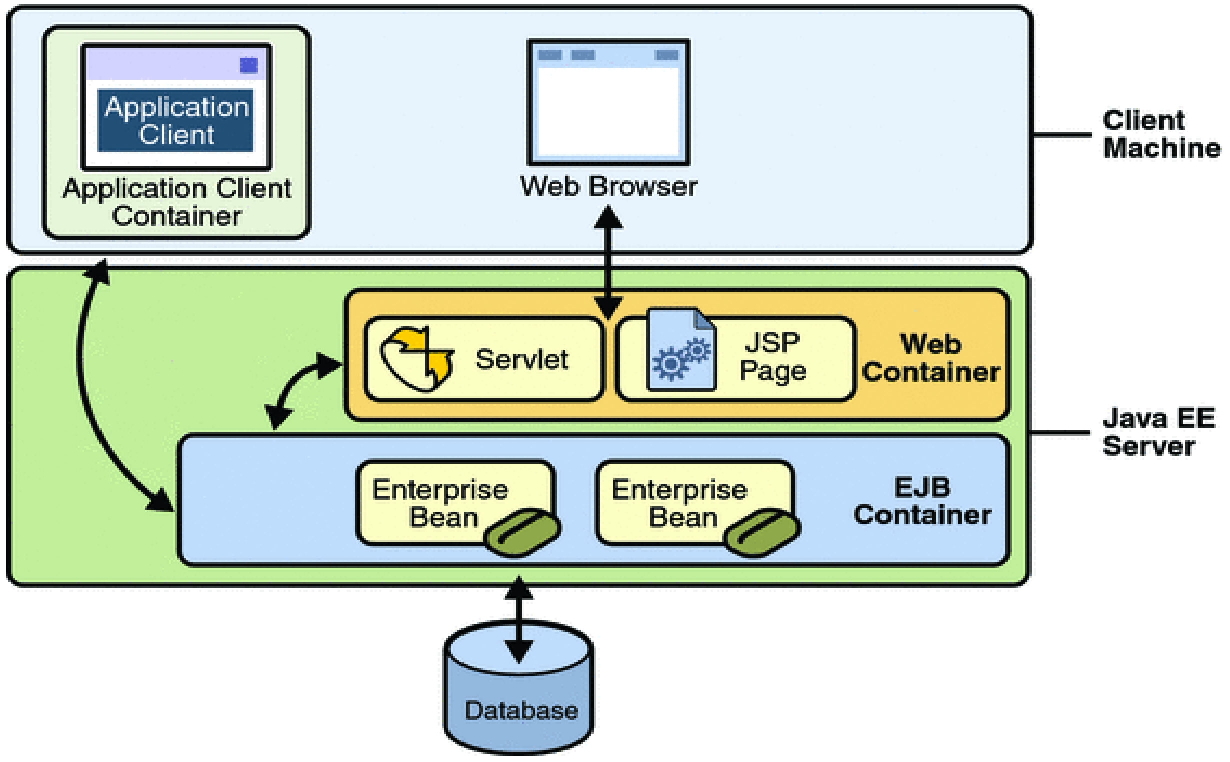
Reprise: 3-tiered architecture



EJB 3-tiered architecture



Java EE 3-Tier Architecture



EJBs as Components

- Enterprise Java Beans are components that provide middle-tier business logic
- And interact heavily with the data layer of the application
- EJB framework conforms to and at the same time induces a 3-tier architecture for distributed applications

EJB as Component Model Framework

- Programming model
- Standardized interfaces
- Runtime environment
- Built-in component services (persistence, transactions, security, etc.)
- Meta-data
- Deployment facilities

EJB Specification

- EJB is an *open* specification - any vendor can develop a runtime environment that complies with the specification
- EJB code intended to be portable across brands (assuming uses only services defined by the spec, not additional vendor facilities)
- EJB specs have evolved:
  - Originated with IBM 1997
  - Later adopted by Sun (1.0 1998, 1.1 1999)
  - Enhanced under Java community process (2.0 2001, 2.1 2003, 3.0 2006)
- EJB 3.0 is a major departure from earlier versions, but backwards compatible (old code works with 3.0 but not vice versa)

Enterprise Beans

- Body of code with fields and methods
- Encapsulates business data or business logic that operates on the enterprise’s data
- Instances are created and managed at runtime by a *Container* (application server)
- Client access is mediated by the bean instance’s Container - isolates the bean from direct access by client applications (and other beans)

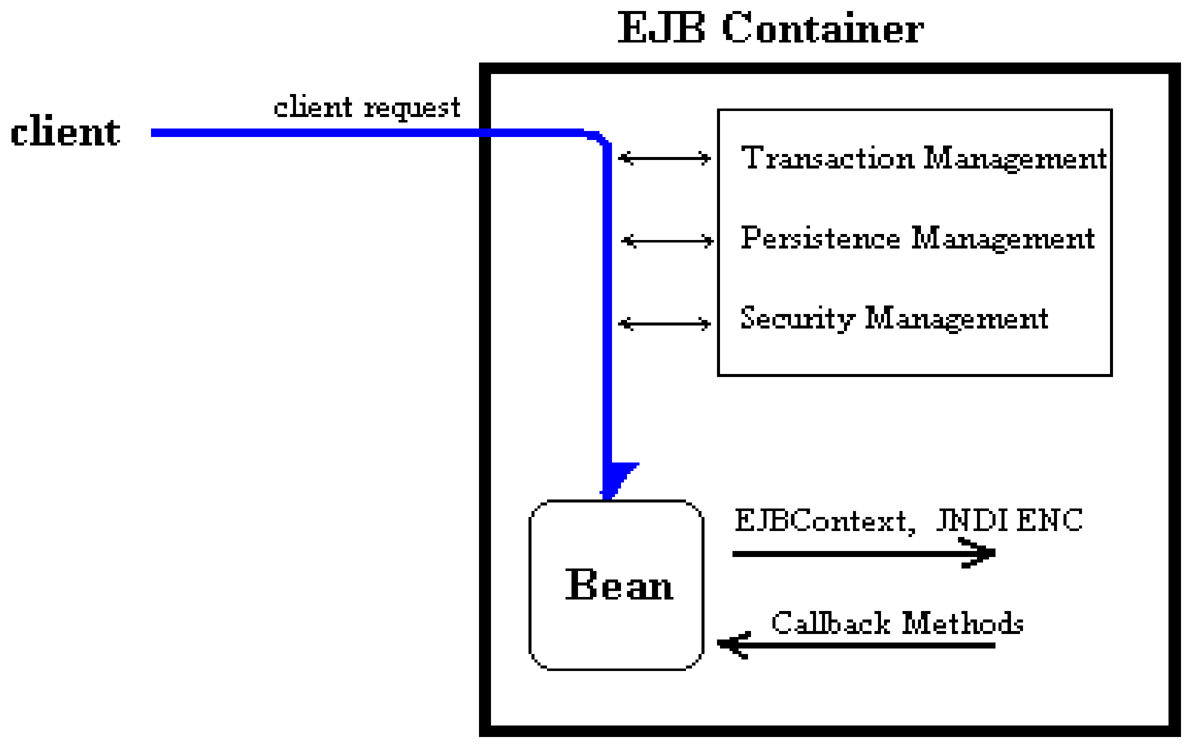
Enterprise Bean Portability

- If an enterprise bean uses only the services defined by the EJB specification (and not additional facilities peculiar to the vendor), the bean can be deployed in any compliant EJB Container
- Can be included in an assembled application without requiring source code changes or recompilation
- Component services information, such as a transaction and security attributes, are separate from the enterprise bean class - this allows the services information to be managed by tools during application assembly and deployment
- Can be customized at deployment time by editing the bean's environment entries and/or deployment descriptor

EJB Container

- Manages every aspect of an enterprise bean at runtime and implements component services
- When a client application invokes a method on an enterprise bean, the container first intercepts the invocation to ensure persistence, transactions and access control are applied properly to every operation a client performs on the bean
- An enterprise bean cannot function outside of an EJB container

EJB Container



EJB Containers manage enterprise beans at runtime

Resource Management

- Containers manage many beans simultaneously
- To reduce memory consumption and processing, containers pool resources
- When a bean is not being used, a container may place it in a pool to be reused by another client
- Or evict it from memory (passivate) and only bring it back (activate) when its needed
- While its reference on the client remains intact
- When the client invokes a method on the reference, the container re-incarnates the bean to service the request

Business Data and Methods

- An *entity* bean (aka persistence entity) represents persistent business data stored in one row of a database table, and may add behavior specific to that data - but the methods are often just getters, setters and finders
- *Session* beans implement business processes and interact with clients
- *Message-driven* beans combine features of a session bean and a message listener, allowing a business component to receive messages (and event notifications) asynchronously

Business Interfaces

- A “business interface” is required for both session and message-driven beans (and for entities prior to EJB 3.0)
- The business interface of a message-driven bean is defined by the messaging type used (typically `MessageListener`), not by the developer

Multiple Interfaces

- If a bean class implements only a single interface (not counting standard interfaces such as java.io.Serializable or any of the javax.ejb interfaces), it is deemed the “business interface” and is by default a local interface unless designated by a @Remote annotation
- A bean class may have multiple interfaces, but one or more must be designated as a business interface by either a @Local or @Remote annotation
- Remote business interfaces support remote clients running on a different JVM or machine, to which the bean’s location is transparent
- If there are only local interfaces, all clients must run in the same JVM as the bean, and the location of the bean is <u>not</u> transparent

Example

```
@Stateless @Remote
public class CalculatorBean implements Calculator {
  public int add (int a, int b) {
      return a + b;
  }
  public int subtract (int a, int b) {
  return a - b;
  }
}
public interface Calculator {
  public int add (int a, int b);
  public int subtract (int a, int b);
  }
```

Remote and Local Interfaces

- To allow remote access, must decorate the business interface with the @Remote annotation

  @Remote public interface InterfaceName { ... }

- OR decorate the bean class with @Remote, specifying the business interface(s)

  @Remote(InterfaceName.class) public class BeanName implements InterfaceName { ... }

- To build an enterprise bean that allows only local access, optionally annotate the business interface of the enterprise bean as @Local

  @Local public interface InterfaceName { ... }

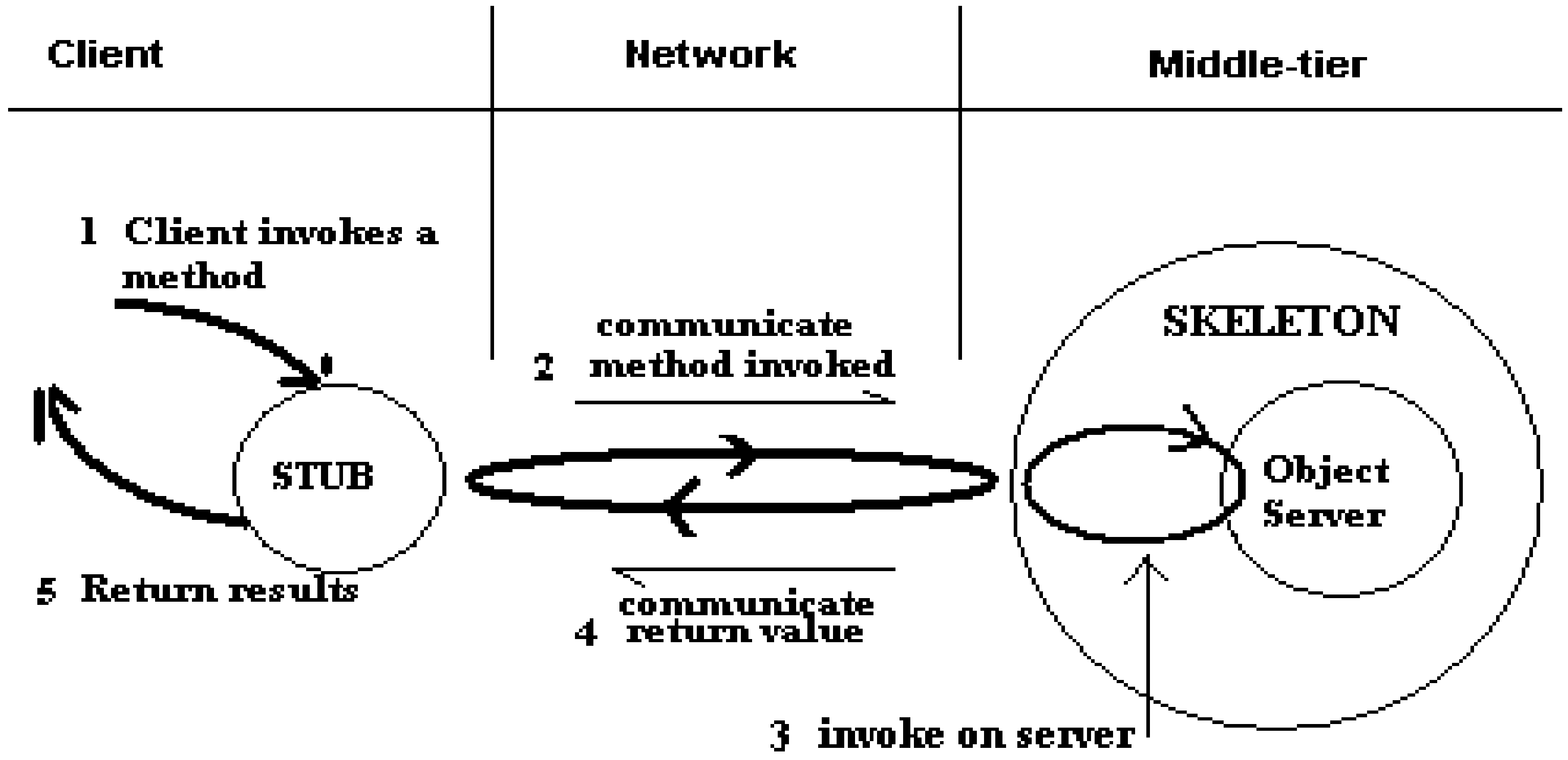
- OR specify the interface by decorating the bean class with @Local and specify the interface name

  @Local(InterfaceName.class) public class BeanName implements InterfaceName { ... }

Enterprise Beans as Distributed Objects

- Business interfaces are types of Java RMI Remote interfaces
- The `java.rmi.Remote` interface is used by distributed objects to represent the bean in a different address space (process or machine)
- An enterprise bean class is instantiated and lives in its container but can be accessed by client applications that live in other address spaces, using skeletons and stubs implemented by the container

Stubs and Skeletons



Deciding on Local vs. Remote: Coupling

- Tightly coupled beans depend on one another
- For example, if a session bean that processes sales orders calls a session bean that emails a confirmation message to the customer, these beans are tightly coupled
- Tightly coupled beans are good candidates for local access
- Because they fit together as a logical unit, they typically call each other often and would benefit from the increased performance that is possible with local access

Deciding on Local vs. Remote: Type of Client

- If an enterprise bean is accessed by application clients, then it should allow remote access
- In a production environment, these clients almost always run on different machines than the Application Server
- If an enterprise bean's clients are web components or other enterprise beans, then the type of access depends on how you want to distribute your components

Deciding on Local vs. Remote: Component Distribution

- Java EE applications are scalable because their server-side components can be distributed across multiple machines
- In a distributed application, the web components may run on a different server than do the enterprise beans they access
- Then the enterprise beans should allow remote access

Deciding on Local vs. Remote: Performance

- Due to factors such as network latency, remote calls are often slower than local calls
- On the other hand, if you distribute components among different servers, you may improve the application’s overall performance
- Actual performance can vary in different operational environments

Deciding on Local vs. Remote

- If you aren’t sure which type of access an enterprise bean should have, choose remote access, which gives more flexibility
- In the future you can distribute your components to accommodate the growing demands on your application
- It is possible for an enterprise bean to allow both remote and local access through different interfaces (the same business interface cannot be both a local and remote business interface)

Session Beans

Session Bean

- Represents a single client (at a time) inside the Application Server
- Client invokes the session bean’s methods to execute business tasks
- When the client terminates, the session bean appears to have terminated and is no longer associated with the client

Stateful vs. Stateless

- There are two basic kinds of session bean: Stateless and Stateful
- *Stateful* session beans encapsulate business logic and state specific to a client
- Stateful beans are called "stateful" because they maintain conversational state between method invocations
- The state is held in instance variables (in memory) and is not persistent across executions
- The state disappears when the client removes the bean or terminates

Stateful Session Beans

- To conserve resources, stateful session beans may be *passivated* when not in use by the client
- Passivation means the bean's conversational-state is written to secondary storage (disk) and the instance is removed from memory
- If the client removes the bean or terminates, the session ends and the state disappears
- The client's reference to the bean is not affected by passivation: it remains alive and usable while the bean is passivated
- When the client invokes a method on a bean that is passivated, the container will activate the bean by instantiating a new instance and populating its conversational-state with the state previously written to secondary storage

Stateless vs. Stateful

- *Stateless* session beans are made up of business methods that behave like functions: they operate only on the arguments passed to them when they are invoked (but can lookup state in a database or file)
- Stateless beans are called "stateless" because they are transient - they do not maintain a conversational state between method invocations
- The bean’s instance variables may contain a state specific to the client during a <u>single</u> method invocation, but not retained when the method is finished

Stateless Session Beans

- Each invocation of a stateless business method is independent from previous invocations
- Because stateless session beans are "stateless" they tend to process requests faster and use less resources
- All instances are equivalent – the EJB container can assign a pooled stateless bean instance to any client, improving scalability

Session Bean Interfaces

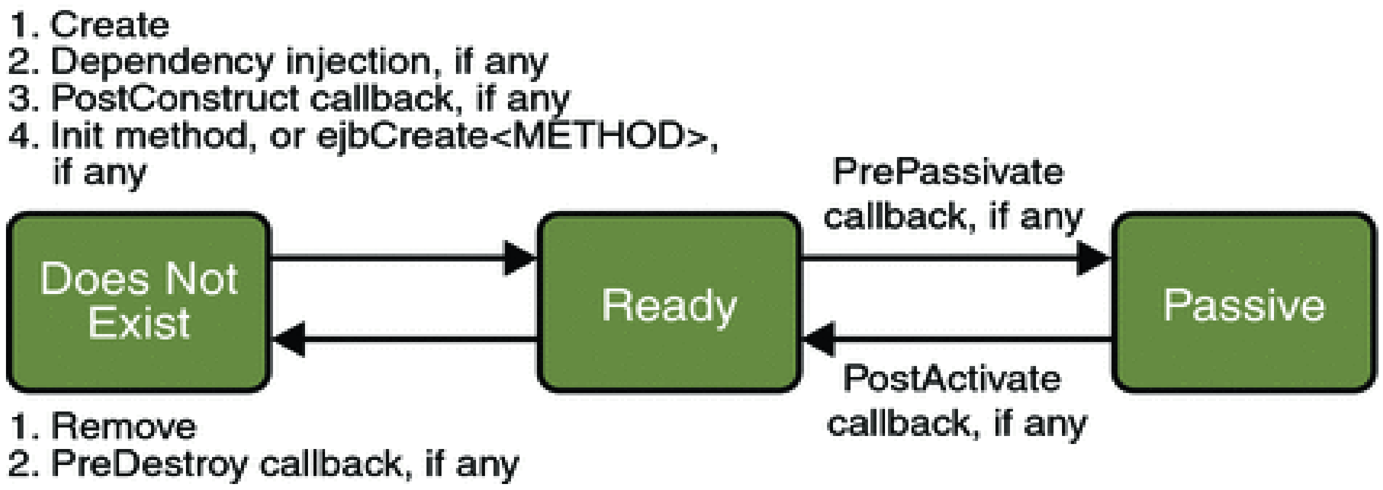
- A client can access a session bean only through the methods in the bean’s business interface
- Can have more than one business interface
- A business interface can be either local or remote (or web service)
- Not required to implement any lifecycle methods, but may optionally do so and annotate as such (prior to EJB 3.0, all enterprise beans had to implement a “home” interface with lifecycle methods)

Lifecycle Methods

- The actual methods can have any names
- `@PostConstruct`: The container immediately calls the annotated method after a bean instance is instantiated
- `@Init`: Designates initialization methods for a stateful session bean
- `@PrePassivate`: Called before the container passivates a stateful bean instance
- `@PostActivate`: Called when a re-activated stateful bean instance is ready
- `@Remove`: Informs the container to remove the bean instance from the object pool after the method executes (not actually a callback)
- `@PreDestroy`: Called before the container destroys an unused or expired bean instance from its object pool

Lifecycle of a Stateful Session Bean

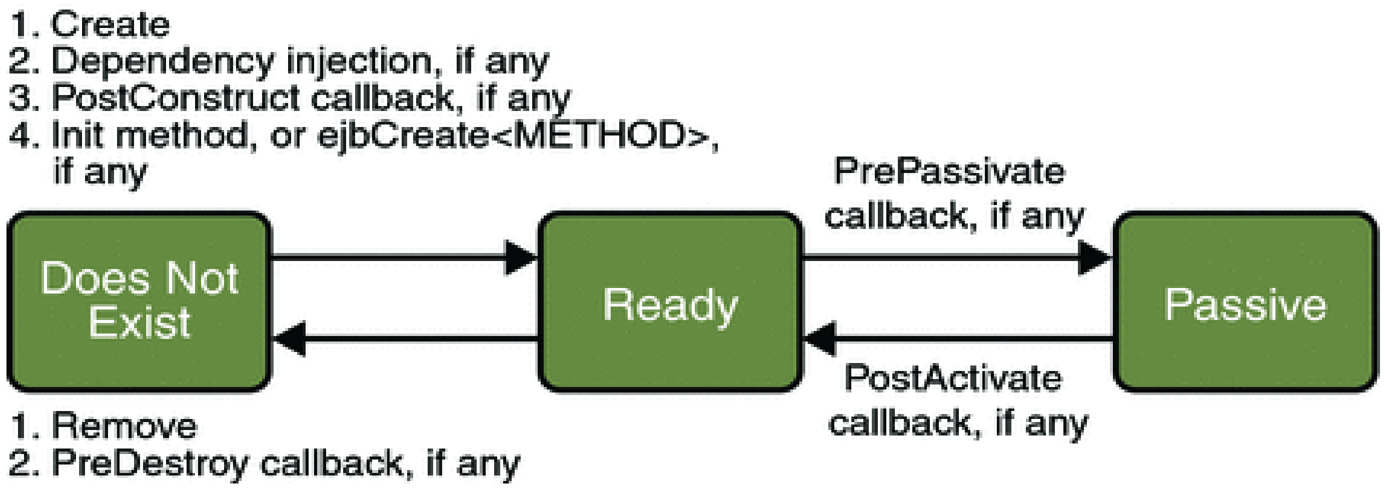
- Client initiates the lifecycle by obtaining a reference
- Container invokes the `@PostConstruct` and `@Init` methods, if any
- Now bean ready for client to invoke business methods



1. Create
2. Dependency injection, if any
3. PostConstruct callback, if any
4. Init method, or ejbCreate<METHOD>, if any
Does Not Exist
Ready
Passive
PrePassivate callback, if any
PostActivate callback, if any
1. Remove
2. PreDestroy callback, if any

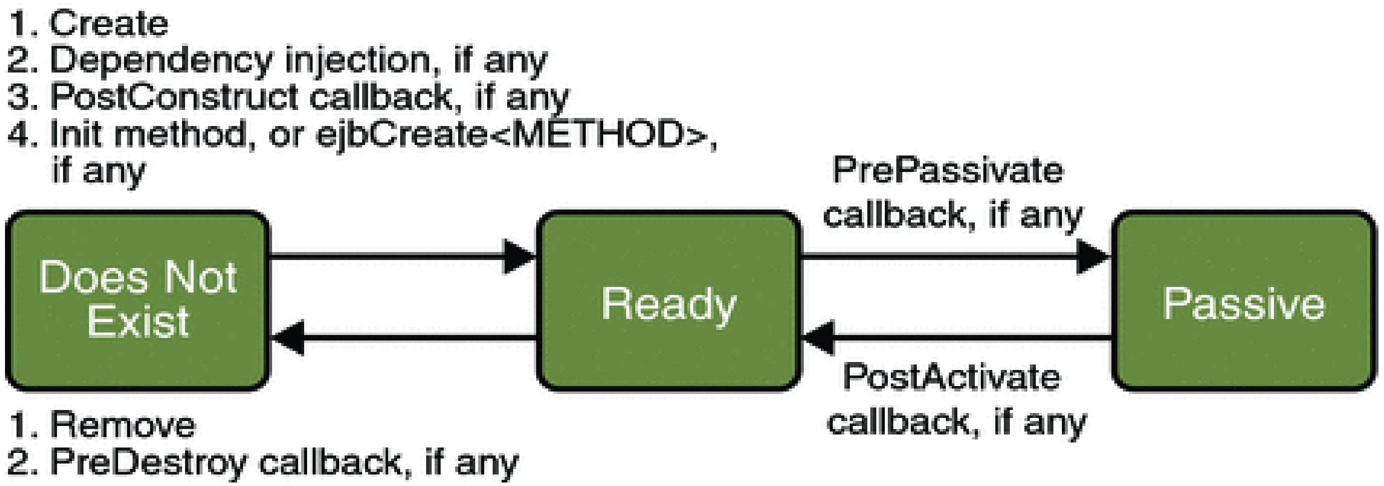
Lifecycle of a Stateful Session Bean

- While in ready state, container may passivate and invoke the `@PrePassivate` method, if any
- If a client then invokes a business method, the container invokes the `@PostActivate` method, if any, and it returns to ready stage


1. Create
2. Dependency injection, if any
3. PostConstruct callback, if any
4. Init method, or ejbCreate<METHOD>, if any
PrePassivate callback, if any
Does Not Exist
Ready
Passive
PostActivate callback, if any
1. Remove
2. PreDestroy callback, if any

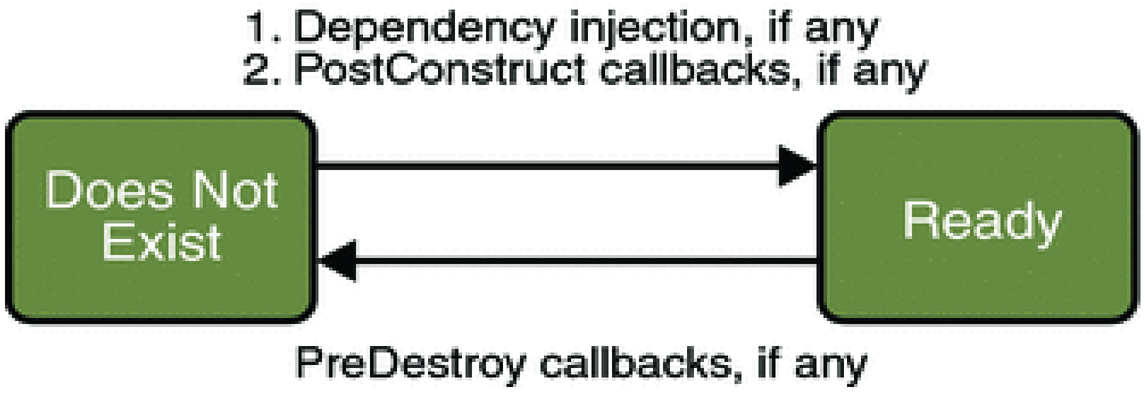
Lifecycle of a Stateful Session Bean

- At the end of the life cycle, the client invokes a method annotated `@Remove`
- The container calls the `@PreDestroy` method, if any


1. Create
2. Dependency injection, if any
3. PostConstruct callback, if any
4. Init method, or ejbCreate<METHOD>, if any
Does Not Exist
Ready
Passive
PrePassivate callback, if any
PostActivate callback, if any
1. Remove
2. PreDestroy callback, if any

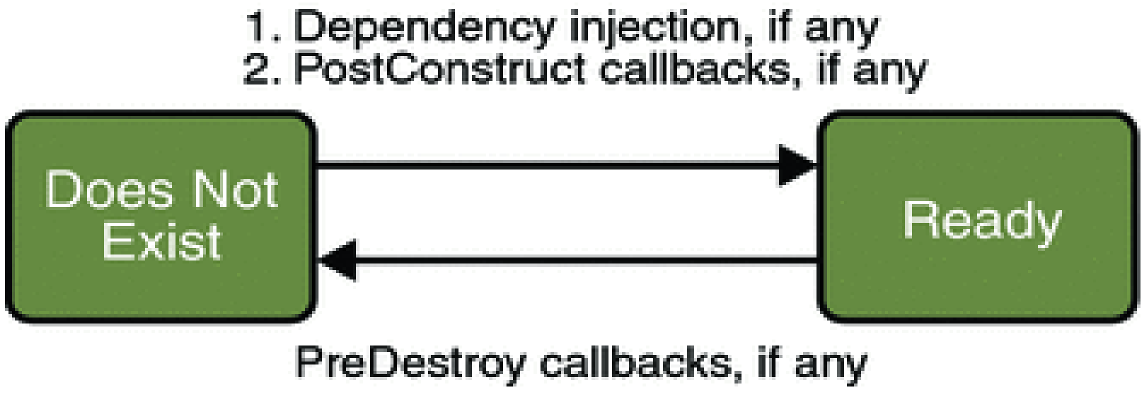
Lifecycle of a Stateless Session Bean

- A client initiates the life cycle by obtaining a reference
- The container invokes the `@PostConstruct` method, if any
- The bean is now ready to have its business methods invoked by clients


1. Dependency injection, if any
2. PostConstruct callbacks, if any
Does Not Exist
Ready
PreDestroy callbacks, if any

Lifecycle of a Stateless Session Bean

- Because a stateless session bean is never passivated, its life cycle has only two stages: nonexistent and ready for the invocation of business methods.
- At the end of the life cycle, the container calls the `@PreDestroy` method, if any


1. Dependency injection, if any
2. PostConstruct callbacks, if any
Does Not Exist
Ready
PreDestroy callbacks, if any

Entity Beans

Entity Beans

- Called entity beans < EJB 3.0, persistence entities (or just entities) >= EJB 3.0
- Provides an object view of data in the database
  - An entity class represents a table in a relational database
  - An entity instance represents a row in that table
- Uses the Java Persistence API
- Annotated with `@Entity`

Entity Beans

- An entity bean provides an object view of data in the database
- Allows shared access from multiple users
- Can be long-lived (as long as data in the database)
- Persistent
  - The entity and its remote reference survive a crash of the EJB Container
  - If the state of an entity was being updated by a transaction at the time the container crashed, the entity’s state is automatically reset to the state of the last committed transaction
  - An application program can create an entity bean, then be stopped and restarted, and again find the entity bean it was working with - and continue using the same entity bean

Instance Variables

- Persistent instance variables can only be accessed through the entity class' methods
- Must only be serializable types (so they can be stored in a database)
- Object/relational mapping must be defined
- An entity may include non-persistent instance variables, annotated as `@Transient`

Entity Beans are Identified by a Primary Key

- Entity Beans must have a *primary key* that uniquely identifies it
- Used to locate the bean’s data in some underlying database
- For example, an “employee” entity bean may have a Social Security number as primary key
- You can only use entity beans when your objects have a unique identifier field, or when you can add such a field (or set of fields)

Primary Keys

- May be either simple or composite
- Simple primary keys annotated `@Id`
- Composite primary keys defined by a primary key class, annotated `@IdClass`
- The simple primary key, or each field of a composite primary key, must be a Java primitive type, string or date
- `EntityManager.find` method used to look up entities by primary key, returns reference to the one specific entity bean (exception if not found)

Queries

- Other finder methods defined using SQL-like queries in Java Persistence Query Language, return a collection of entities that match the request
- `EntityManager.createQuery` method used to create *dynamic queries* defined within business logic

```
public List findWithName(String name) {
return em.createQuery(
    "SELECT c FROM Customer c WHERE c.name
  LIKE :custName")
    .setParameter("custName", name)
    .setMaxResults(10)
    .getResultList();
}
```

Queries

- `EntityManager.createNamedQuery` method used to create *static queries* defined in annotation metadata

```
@NamedQuery(
    name="findAllCustomersWithName",
    query="SELECT c FROM Customer c WHERE c.name LIKE
  :custName"
)
customers =
  em.createNamedQuery("findAllCustomersWithName")
    .setParameter("custName", "Smith")
    .getResultList();
```

Managing Entities

- An Entity Manager is associated with a persistence context corresponding to a particular data store
- State of persistent entities automatically synchronized to the database when the associated transaction commits
- But business logic for transactions resides in session or message-driven beans
- Both Container-Managed Entity Managers (automatic) and Application-Managed Entity Managers

Container-Managed Transactions

- Container sets the boundaries of transactions, cannot use operations like `commit` or `rollback` within code
- Container begins transaction immediately before enterprise bean method starts and commits just before method exits
- Transaction types: `Required, RequiresNew, Mandatory, NotSupported, Supports, Never`

Transactional Attributes

- **Required -** If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container starts a new transaction before running the method.
- **RequiresNew -** If the client is running within a transaction and invokes the enterprise bean's method, the container suspends the client's transaction, starts a new transaction, delegates the call to the method, resumes the client's transaction after the method completes; if the client is not associated with a transaction, the container starts a new transaction before running the method.
- **NotSupported -** If the client is running within a transaction and invokes the enterprise bean's method, the container suspends the client's transaction before invoking the method; after the method has completed, the container resumes the client's transaction.
- **Supports -** If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction; if the client is not associated with a transaction, the container does not start a new transaction before running the method
- **Mandatory -** If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction; if the client is not associated with a transaction, the container throws the `TransactionRequiredException`.
- **Never -** If the client is running within a transaction and invokes the enterprise bean's method, the container throws a `RemoteException`

Application-Managed Transactions

- The code in the session or message-driven bean explicitly marks the boundaries of the transaction
- Useful for implementing multiple transactions within a single method or transactions than span multiple methods
- Can use either Java Database Connectivity (JDBC) or the Java Transaction API (JTA)
- A JTA transaction can span updates to multiple databases from different vendors managed by the Java Transaction Service, but cannot support nested transactions
- JTA supplies `begin`, `commit` and `rollback` methods

Using Transactions in Session Beans

- A stateless session bean with bean-managed transactions must commit or rollback before returning
- A stateful session bean using JTA transactions retains its association with a transaction across multiple client calls, even if the database connection is opened and closed
- A stateful session bean using JDBC transactions loses its transaction association if the connection is closed

Saving a Session Bean’s State in a Database

- Transactions normally concerned with synchronizing the state of persistent entities to databases
- Optional for a stateful session bean to receive transaction synchronization notifications to also store its own data in a database
- Then must implement the `SessionSynchronization` interface, supplying `afterBegin`, `beforeCompletion` and `afterCompletion` methods