Software Architecture Design II
Java EE Middleware

Matthew Dailey

Computer Science and Information Management
Asian Institute of Technology

Readings

Readings for these lecture notes:

- Burke, B. and Monson-Haefel, R. *Enterprise JavaBeans 3.0*, 5th edition, O'Reilly, 2006.
- Jendrock, E., Ball, J., Carson, D, Evans, I., Fordin, S., and Haase, K. *The Java EE 5 Tutorial*, 3rd edition, Addison-Wesley, 2006, Chapter 31.

Outline

For large-scale enterprise application development and integration, the main choices are Java EE and Microsoft .NET.

The two technologies offer similar capability. .NET is simpler but single-platform; Java EE is cross-platform and multi-vendor.

Here we will discuss how Java EE technology supports the design techniques we have been learning.

Java EE's main pieces:

- Servlets: an API for processing HTTP requests and generating Web pages dynamically.
- JavaServer Pages (JSP): an extension of servlets allowing embedding of Java into HTML documents.
- Enterprise JavaBeans (EJB): a server-side framework for distributed components.

Java EE also dictates standards for messaging, naming and directory services, database connectivity, and more.

We will focus on EJB 3.0 support for the domain and data source layers.

Enterprise JavaBeans (Burke and Monson-Haefel, 2006), p 5

Enterprise JavaBeans is a standard server-side component model for distributed business applications.

For persistence, EJB 3.0 maps entity beans (POJOs with a bit of metadata) to a database via the Java Persistence API (JPA).

Session beans are components with remote interfaces:

- Session beans implement synchronous (RMI-style) interaction.
- Session beans can be stateful or stateless.
- Session beans implement most of the taskflow aspects of the business logic.

Message-driven beans (MDBs) provide processing of asynchronous messages delivered through the Java Messaging Service (JMS).

Connectivity with other Enterprise Information Systems is provided through the Java EE Connector Architecture (JCA) and Web services (JAX-WS).

Lookup and naming of remote objects is provided through the Java Naming and Directory Interface (JNDI).

Java EE-compliant application servers provide EJBs with a run-time environment called the EJB container.

There are many EJB container vendors but they must all implement a common set of standards.

The EJB container provides EJBs with 8 primary services:

- Concurrency: the container manages threads and concurrent access to entity beans and other resources.
- Transactions: the container supports atomic isolated distributed transactions.

- Persistence: the container automatically maps entities to persistent storage with the help of developer-supplied metadata.
- Distribution: the container provides transparent synchronous invocation of remote EJBs.
- Asynchronous messaging: the container supports reliable messaging from external clients to MDBs.
- Naming: the container provides an API (JNDI) for naming, lookup, and referencing of remote objects.
- Timer: the container allows MDBs, stateles session beans, and entity beans to register for notification messages at specific times.
- Security: the container provides client authentication (identification), authorization (access control), and privacy (encryption).

Outline

Bean types

Entity beans

Entity beans are POJOs whose persistence is managed through the Java Persistence API.

To illustrate, we'll use Burke and Monson-Haefel (2006)'s Titan cruise ship booking application.

Example of the entity for a cabin on a ship:

```
package com.titan.domain
import javax.persistence.*;

@Entity
@Table(name="CABIN")
public class Cabin implements java.io.Serializable{
   private int id;
   private String name;
   private int deckLevel;
   private int shipId;
   private int bedCount;
```

Bean types

Entity beans

```
    @Id
    @Column(name="ID")
    public int getId( ) { return id; }
    public void setId(int pk) { id = pk; }

    @Column(name="NAME")
    public String getName( ) { return name; }
    public void setName(String str) {name = str; }

    @Column(name="DECK_LEVEL")
    public int getDeckLevel( ) { return deckLevel; }
    public void setDeckLevel(int level) { deckLevel = level; }

    @Column(name="SHIP_ID")
    public int getShipId( ) { return shipId; }
    public void setShipId(int sid) { shipId = sid; }

    @Column(name="BED_COUNT")
    public int getBedCount( ) { return bedCount; }
    public void setBedCount(int bed) { bedCount = bed; }
}
```

The annotations marked with '@' are directives telling the container how to use the bean. This metadata can be provided through XML "deployment descriptor" files instead of annotations.

In the example, only the `@Entity` and `@Id` annotations are strictly needed. JPA can define default table and column names.

Other than annotating the POJO as an entity, we need to tell the container what EntityManager to use and what database the EntityManager should connect to.

The EntityManager is defined in an XML file called `persistence.xml`, for example:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence>
   <persistence-unit name="titan">
      <jta-data-source>java:/DefaultDS</jta-data-source>
      <properties>
         <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      </properties>
   </persistence-unit>
</persistence>
```

This descriptor tells the container to use the default EntityManager (in this case Hibernate on JBoss) and to automatically create and drop tables as needed.

Bean types

Session beans

Session beans have three parts:

- A remote interface denoted by the `@Remote` annotation, for clients outside the EJB container.
- A local interface denoted by the `@Local` annotation, for clients inside the EJB container. The local interface is lower overhead.
- A bean class implementing the interfaces, denoted as `@Stateless` or `@Stateful`.

It's not necessary for a session bean to have both local and remote interfaces.

Example: TravelAgent session bean for the Titan cruise ship reservation application.

First, the TravelAgent remote interface:

```
package com.titan.travelagent;
import javax.ejb.Remote;
import com.titan.domain.Cabin;

@Remote
public interface TravelAgentRemote {
   public void createCabin(Cabin cabin);
   public Cabin findCabin(int id);
}
```

Other than the `@Remote` interface, it is a normal Java interface.

Here is the TravelAgent bean class:

```
package com.titan.travelagent;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import com.titan.domain.Cabin;

@Stateless
public class TravelAgentBean implements TravelAgentRemote{
   @PersistenceContext(unitName="titan") private EntityManager manager;

   public void createCabin(Cabin cabin) {
      manager.persist(cabin);
   }

   public Cabin findCabin(int pKey) {
      return manager.find(Cabin.class, pKey);
   }
}
```

The `@Stateless` annotation indicates this is a stateless session bean.

The `@PersistenceContext` annotation tells the container we want access to the EntityManager persistence service (defined in `persistence.xml`).

To test the session bean and entity bean, we need to create a client application that invokes the session bean's remote interface.

Let's write a simple client that creates a new Cabin entity and makes it persistent:

```
package com.titan.clients;
import com.titan.travelagent.TravelAgentRemote;
import com.titan.domain.Cabin;
import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import java.util.Properties;
import javax.rmi.PortableRemoteObject;
```

Bean types

Client application

```
public class Client {
   public static void main(String [] args) {
      try {
         Context jndiContext = getInitialContext( );
         Object ref = jndiContext.lookup("TravelAgentBean/remote");
         TravelAgentRemote dao = (TravelAgentRemote)
            PortableRemoteObject.narrow(ref,TravelAgentRemote.class);

         Cabin cabin_1 = new Cabin( );
         cabin_1.setId(1);
         cabin_1.setName("Master Suite");
         cabin_1.setDeckLevel(1);
         cabin_1.setShipId(1);
         cabin_1.setBedCount(3);

         dao.createCabin(cabin_1);
```

Bean types

Client application

```
            Cabin cabin_2 = dao.findCabin(1);
            System.out.println(cabin_2.getName( ));
            System.out.println(cabin_2.getDeckLevel( ));
            System.out.println(cabin_2.getShipId( ));
            System.out.println(cabin_2.getBedCount( ));
        } catch (javax.naming.NamingException ne){ne.printStackTrace( );}
    }

    public static Context getInitialContext( )
        throws javax.naming.NamingException {

        Properties p = new Properties( );
        // ... Specify the JNDI properties specific to the vendor.
        return new javax.naming.InitialContext(p);
    }
}
```

[Run the code]

Outline

EJB container

EJB philosophy: developers should focus on business logic and avoid worrying about concurrency, transactions, O/R mapping, and so on.

To use the framework effectively we need to know what the container is doing for us.

The container mediates an EJB's interaction with the outside world and manages the EJB's lifecycle.

Clients of a session bean never invoke bean class instances directly. Instead they call methods on a proxy stub generated automatically from the `@Remote` or `@Local` interface definition.

The proxy stub requests forwards method calls to the EJB container, which takes care of security and transactions then invokes the session bean class, collects the response, and sends that response back to the client.

EJB container

Instance pooling

To increase resource utilization efficiency, stateless session bean and message driven bean instances are shared between clients.

For stateless session beans, the EJB container maintans a pool of bean instances that can be allocated to any EJB object representing a client method call.

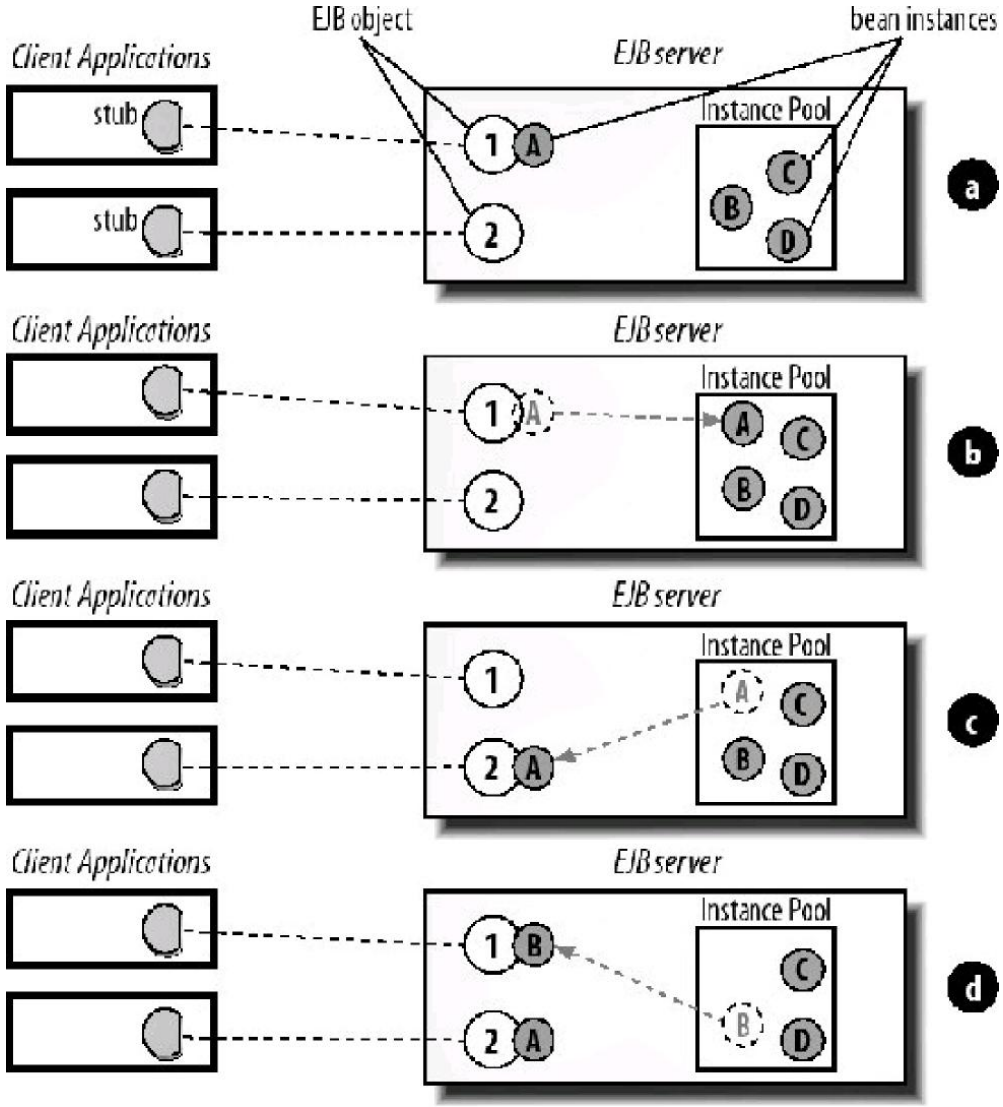
For MDBs, the EJB container maintains a pool of bean instances for each message destination.

Stateful session beans are not pooled. When the EJB needs to conserve resources, it may swap stateful session bean instances to disk (passivation) then restore them (activation) when a new request comes from the client.

Stateful session beans need to close any connection-oriented resources before passivation and restore them on activation. This is accomplished through annotations marking methods as passivation/activation callbacks.

EJB container

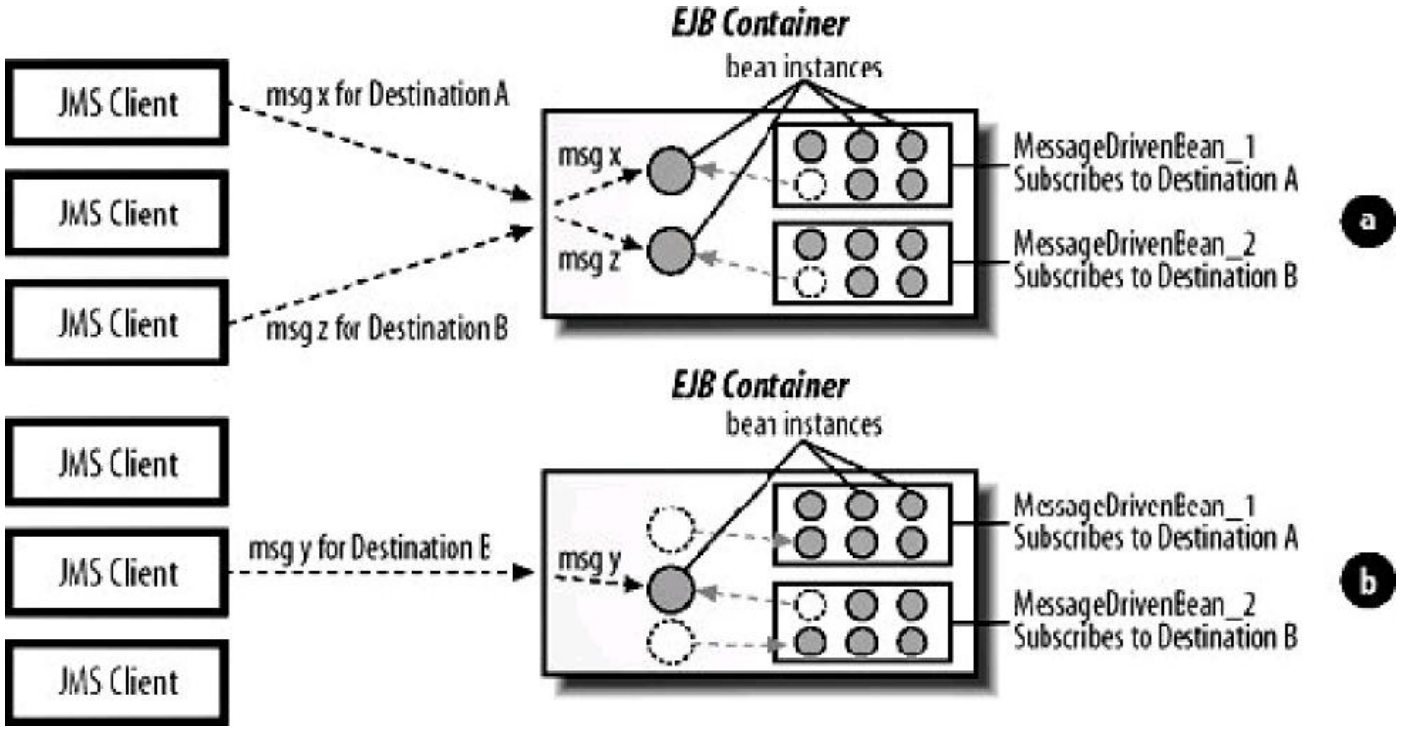
Instance pooling (stateless session beans)



Burke and Monson-Haefel (2006), Fig. 3-1

EJB container

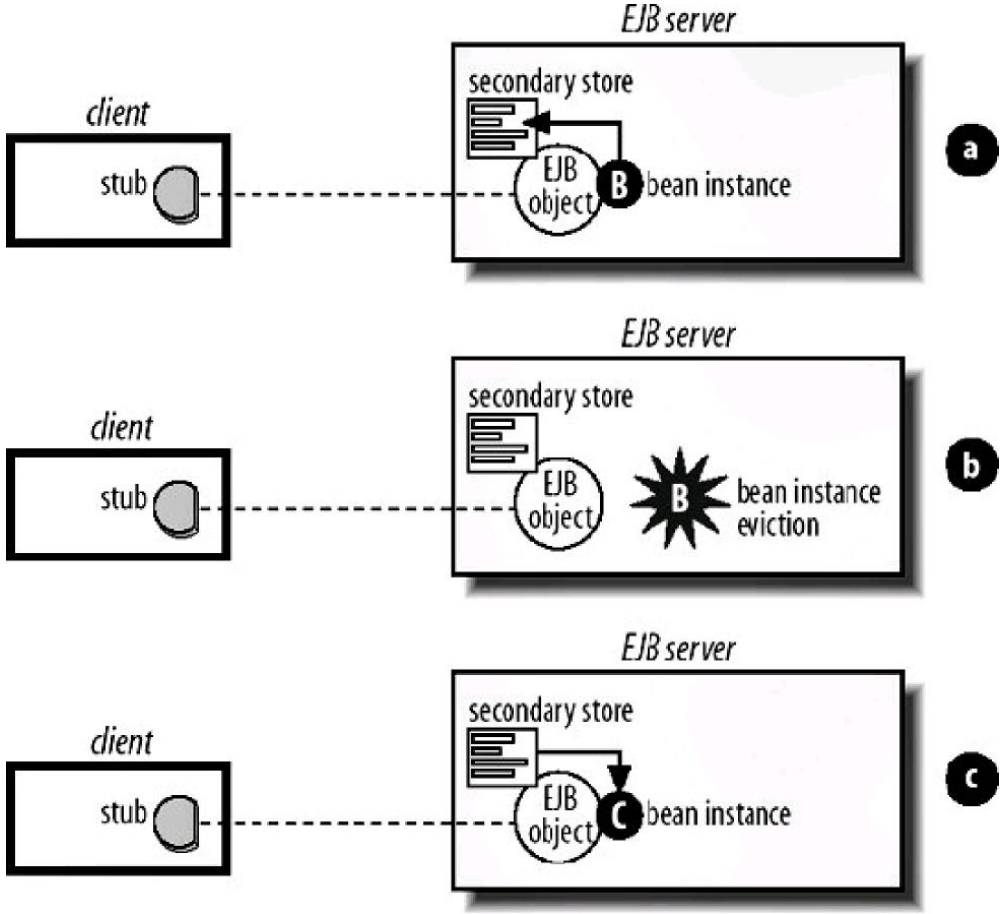
Instance pooling (message-driven beans)



Burke and Monson-Haefel (2006), Fig. 3-2

EJB container

Stateful session bean passivation and activation



Burke and Monson-Haefel (2006), Fig. 3-3

Outline

Java Persistence 1.0 is the persistence specification for Java EE 5:

- It is an abstraction on top of JDBC.
- It is a specification for a object-to-relational mapping engine (ORM).
- It provides a query language (EJB QL) similar to SQL but oriented towards querying Java objects rather than a relational schema.

The `javax.persistence.EntityManager` is responsible for mapping, queries, and the Unit of Work.

To make your POJO persistent, you have to explicitly interact with the EntityManager.

The EntityManager works well with EJB but can also run outside the container, in standalone Java SE programs, for example.

Example: a Customer entity:

```
import javax.persistence.*;

@Entity
public class Customer {
   private int id;
   private String name;

   @Id @GeneratedValue
   public int getId( ) {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }
```

```
    String getName( ) {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

There is no magic — if we create a Customer, it is a POJO only until we ask the EntityManager to create it in the database.

Before a POJO is mapped to the database, we say it is detached from or unmanaged by any EntityManager.

State changes to a detached entity are not persistent.

When we attach the object to an EntityManager, the entity becomes managed.

The current set of entities managed by an `EntityManager` is called a persistence context.

The persistence context is the Unit of Work in JPA.

The `EntityManager` watches the persistence context for changes and flushes state to the database as necessary.

When a persistence context is closed, all entities in the persistence context become detached and unmanaged.

A persistence context can be transaction-scoped or extended.

A transaction-scoped persistence context is one managed by an application server. When a transaction completes, the context is destroyed and the managed entity objects are detached.

Example transaction-scoped entity manager and persistence context:

```
@PersistenceContext(unitName="titan")
EntityManager entityManager;

@TransactionAttribute(REQUIRED)
public Customer someMethod() {
   Customer cust = entityManager.find(Customer.class, 1);
   cust.setName("new name");
   return cust;
}
```

Because of the `REQUIRED` transaction attribute, `someMethod()` runs in a transaction. Since instance `cust` came from the EntityManager, it remains attached until the transaction completes and commits. When the transaction completes, the change will be synchronized.

Extended persistence contexts live longer than a transaction. They are created and managed by application code or stateful session beans.

Detached instances can be serialized and sent over a network, changed, sent back to the server, and reattached.

[More on transactions later!]

A persistence unit is a set of classes mapped to a particular database by an EntityManager.

Persistence units are defined in the file `persistence.xml` file, a deployment descriptor required by JPA and packaged in the JAR, EJB-JAR, or WAR file.

A persistence unit is mapped to one and only one data source.

Java EE persistence contexts default to using Java EE Transactions (JTA), allowing the persistence manager to interact with transactions.

Java SE defaults to "resource local" management of persistence units.

If using JTA we define in `persistence.xml` the vendor-specific `jta-data-source`, usually a JNDI name for the referencing data source.

The application server can provide a default, e.g. the Hypersonic SQL database for JBoss or the Derby SQL database for Glassfish.

The set of classes to be included in a persistence unit is determined by:

- Classes annotated with `@Entity` in the `persistence.xml` file's JAR file.
- `@Entity` classes inside JAR files mentioned in the `<jar-file>` elements of `persistence.xml`.
- Classes mapped in the `META-INF/orm.xml` file.
- Classes mapped in any XML files referenced by a `<mapping-file>` element.
- Classes listed with any `<class>` elements in `persistence.xml`.

Once you package and deploy your persistence units, your application code needs to obtain and access an EntityManager associated with that persistence unit.

The simple way to do this is to inject an EntityManager with the `@PersistenceContext` annotation:

```
@Stateless
public class MySessionBean implements MySessionRemote {
   @PersistenceContext(unitName="titan")
   private EntityManager entityManager;
   ...
}
```

The default is a transaction-scoped persistence context. For an extended persistence context, add the `type`:

```
@Stateful
public class MyStatefulBean implements MyStatefulRemote {
   @PersistenceContext(unitName="titan", type=PersistenceContextType.EXTENDED)
   private EntityManager entityManager;
   ...
}
```

Once you have an EntityManager, you can interact with it, as follows.

Persisting an entity means inserting it into a database. Allocate, set properties, wire up relationships, then call `persist()`:

```
Custom cust = new Customer();
cut.setName("Bill");

entityManager.persist(cust);
```

If `persist()` is called within a transaction, the insert can be immediate or at the end of the transaction, depending on the flush mode.

If the entity has relationships with other entities, the insert can cascade depending on cascade policies.

Finding entities uses two techniques:

- Find by primary key:

```
public interface EntityManager {
<T> T find(Class<T> entityClass, Object primaryKey);
<T> T getReference(Class<T> entityClass, Object primaryKey);
}
```

  using Java generics means we don't need type casting. `find()` returns `null` if the entity does not exist; `getReference()` throws an exception.

- Queries using EJB QL, e.g.:

```
Query query = entityManager.createQuery("from Customer c where id=2");
Customer cust = (Customer)query.getSingleResult();
```

Updates to managed entities are automatically synchronized depending on the flush mode.

Merging entities is necessary when a detached entity has been modified and the changes need to be persisted:

```
@PersistenceContext EntityManager entityManager;

@TransactionAttribute(REQUIRED)
public void updateCabin(Cabin cabin) {
   Cabin copy = entityManager.merge(cabin);   // copy is now managed
}
```

Removing an entity from persistent storage uses `remove()`.

An attached entity can be refreshed using `refresh()`.

To check if an entity is managed, use `contains()`.

To detach all entities from a persistence context, use `clear()`. You might call `flush()` first.

To synchronize changes to attached entities with the database, use `flush()`. To switch between `AUTO` flush and `COMMIT` flush, use `setFlushMode()`. `COMMIT` flushing decreases the amount of time locks are held after updates.

Entities can register for notification of a variety of lifecycle events.

The events are `PrePersist`, `PostPersist`, `PostLoad`, `PreUpdate`, `PostUpdate`, `PreRemove`, and `PostRemove`.

When an entity registers for its own lifecycle events, it provides callbacks.

To prevent entity class coupling to the data source, we can use helper classes called listeners that intercept entity callback events.

This facility is useful for things like logging.

Outline

Object-relational mapping

Basic mapping

JPA maps simple objects to single tables.

Each basic property of an object is mapped to a column in a table.

The basic annotations are `@Entity` and `@Id`. If tables and column names are not specified, the container uses unqualified object and field names.

Object-relational mapping

Basic mapping

```
package com.titan.domain;
import javax.persistence.*;

@Entity
public class Customer implements java.io.Serializable {
   private long id;
   private String firstName;
   private String lastName;

   @Id
   public long getId( ) { return id; }
   public void setId(long id) { this.id = id; }

   public String getFirstName( ) { return firstName; }
   public void setFirstName(String firstName) { this.firstName = firstName; }

   public String getLastName( ) { return lastName; }
   public void setLastName(String lastName) { this.lastName = lastName; }
}
```

In the example we are relying on the default mapping:

```
create table Customer(
   id long primary key not null,
   firstName VARCHAR(255),
   lastName VARCHAR(255)
);
```

Mapping metadata can be expressed with annotations in the Java code or an external XML deployment descriptor, e.g. `orm.xml`.

In addition to mapping an object schema to a new relational schema, we can map to an existing relational schema using additional annotations.

If we don't like the default DDL we can customize it by parameterizing the annotations:

Object-relational mapping

Basic mapping

```
package com.titan.domain;
import javax.persistence.*;

@Entity
@Table(name="CUSTOMER_TABLE")
public class Customer implements java.io.Serializable {
   private long id;
   private String firstName;
   private String lastName;

   @Id
   @Column(name="CUST_ID", nullable=false, columnDefinition="integer")
   public long getId( ) { return id; }
   public void setId(long id) { this.id = id; }

   @Column(name="FIRST_NAME", length=20, nullable=false)
   public String getFirstName( ) { return firstName; }
   public void setFirstName(String firstName) { this.firstName = firstName; }

   public String getLastName( ) { return lastName; }
   public void setLastName(String lastName) { this.lastName = lastName; }
}
```

The relational primary key is important because it is used by the `EntityManager.find()` method.

There are many ways to map the primary key. It can be autogenerated from a table or sequence. It can be a non-numeric column or even composite.

Properties that shouldn't be mapped to the database can be marked `@Transient`.

To precisely specify mapping of Java data and time objects, use `@Temporal`.

For large objects, we can use `@Lob`.

Large objects should normally be lazy-loaded, which is specified through the annotation `@Basic(fetch=FetchType.LAZY)`.

Java enumeration types can be mapped to strings or integers with `@Enumerated`.

To map a single Java object to a join of two tables in the relational schema, use `@SecondaryTable`.

To map properties of a simple enclosed Java object to the same table as the fields of the enclosing object, use `@Embedded`.

Object-relational mapping

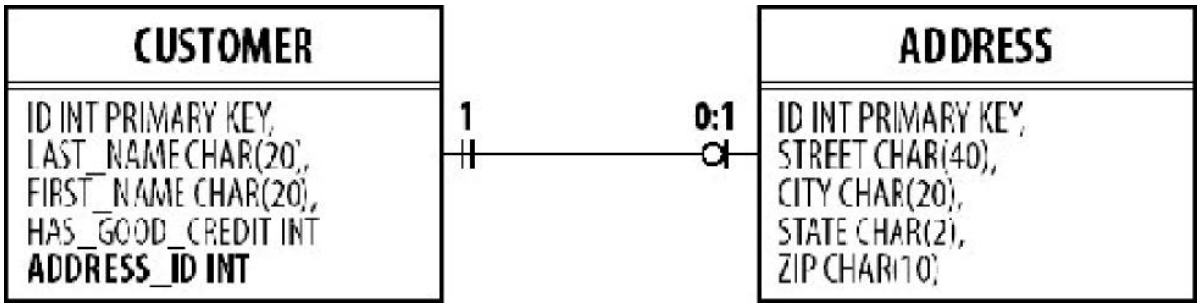
Entity relationships

In the object world, the possible entity relationships are one-to-one, one-to-many, many-to-one, and many-to-many.

Each relationship can be unidirectional or bidirectional in the object schema, giving us 7 kinds of relationships (many-to-one bi $\leftrightarrow$ one-to-many bi).

Note that relations don't have directionality but object references do.

One-to-one unidirectional example: Customer → Address. The Customer table needs a foreign key reference to the Address table.



Burke and Monson-Haefel (2006), Fig. 7-2

The mapping is accomplished with the `@OneToOne` annotation:

```
package com.titan.domain;

@Entity
public class Customer implements java.io.Serializable {
   ...
   private Address address;
   ...

   @OneToOne(cascade={CascadeType.ALL})
   @JoinColumn(name="ADDRESS_ID")
   public Address getAddress( ) {
      return homeAddress;
   }
   public void setAddress(Address address) {
      this.homeAddress = address;
   }
}
```

The cascade type `CascadeType.ALL` specifies that an Address should be persisted, updated, merged, refreshed, and so on whenever the containing Customer is.

The `@JoinColumn` indicates the desired name of the foreign key reference column in the Customer table.

We have similar approaches for the other 6 relationships.

One-to-one bidirectional example: Customer $\leftrightarrow$ CreditCard.

The relational schema will have a foreign key reference to the CreditCard table in the Customer table. The object schema will have references in the Customer and CreditCard objects.

Object-relational mapping

Entity relationships

```
@Entity
public class CreditCard implements java.io.Serializable {
   private int id;
   private Date expiration;
   private String number;
   private String name;
   private String organization;
   private Customer customer;
   ...

   @OneToOne(mappedBy="creditCard")
   public Customer getCustomer( ) {
      return this.customer;
   }
   public void setCustomer(Customer customer) {
      this.customer = customer;
   }
   ...
}
```

The `mappedBy` attribute of the `@OneToOne` annotation:

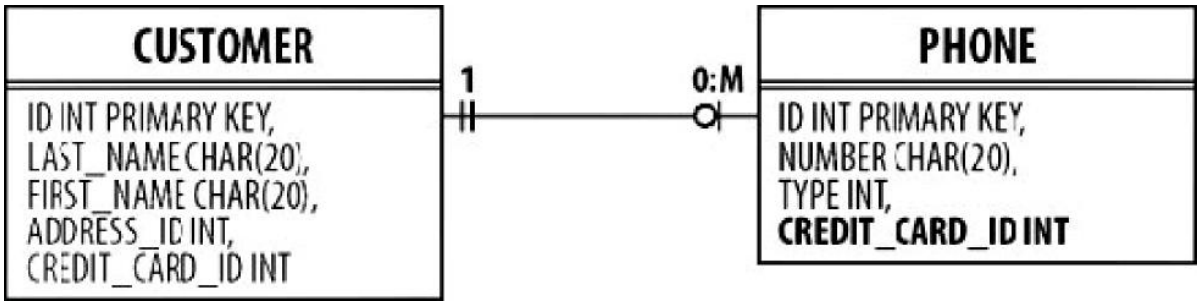
- Says this is a bidirectional object association
- Tells the persistence manager to use the mapped `creditCard` field of the `Customer` class as the foreign key reference.

The Customer class uses `@OneToOne` and `@JoinColumn` for the `creditCard` field, similar to the `address` field.

One-to-many unidirectional example: `Customer` $\longrightarrow$ `Phone`.

In the object schema, `Customer` contains a collection of `Phone` objects.

In the relational schema, one choice would be a foreign key reference to the `Customer` table in the `Phone` table:



Burke and Monson-Haefel (2006), Fig. 7-5

However, Toplink requires us to use a join table (as in a many-to-many association). One-to-many instead of many-to-many is enforced by a uniqueness constraint on the many side.

In the entity definition, we use the `@OneToMany` annotation:

```
@Entity
public class Customer implements java.io.Serializable {
   ...
   private Collection<Phone> phoneNumbers = new ArrayList<Phone>( );
   ...

   @OneToMany(cascade={CascadeType.ALL})
   public Collection<Phone> getPhoneNumbers( ) {
      return phoneNumbers;
   }
   public void setPhoneNumbers(Collection<Phone> phones) {
      this.phoneNumbers = phones;
   }
}
```

In the example, the persistence manager uses the Java generic declaration of the collection type (`Phone`) to determine what class we are mapping to.

Since the association is one-to-many unidirectional, `Phone` does not need any annotations related to `Customer`.

Toplink automatically creates the join table `CUSTOMER_PHONE` with columns `CUSTOMER_ID` and `PHONE_ID` and a uniqueness constraint on `PHONE_ID`.

The mappings work similarly for the rest of the entity relationship types.

JPA allows Sets, Lists, and Maps to be used for the "many" end of entity relationships.

`@JoinColumn` specifies the name of a foreign key reference column used to implement an association in the database.

Default join table behavior can be overridde using `@JoinTable`.

As long as an entity is attached, lazy loading is transparent to the application developer.

But be careful with detached entities!

- Detached entities may have properties that are `null` when the data actually exists but was not loaded.

Object-relational mapping

Cascading

We can control how persistence events like `persist()`, `remove()`, `merge()`, and `refresh()` cascade to related entities.

We use the `cascade` attribute of the mapping annotations (e.g. `@OneToMany(cascade={CascadeType.ALL})`) to specify the behavior.

Be careful, the correct behavior is application specific. Composition relationships will normally cascade, aggregation relationships will normally not cascade.

Example: when a Customer is removed, the related Address object should also be removed.

Example: when a Reservation is removed, the related Cabin objects should not be removed.

JPA provides annotations for the standard inheritance mapping strategies (single table per hierarchy, table per concrete class, table per subclass, and a mixture).

Outline

EJB QL

Introduction

EJB QL is a query language analagous to SQL, but for Java objects rather than the underlying database tables.

EJB QL is vendor neutral, so vendor-specific features like stored procedures need to use native (SQL) queries.

The EJB 3.0 version of EJB QL is also known as JPQL (Java Persistence Query Language). It adds bulk updates and deletes, named parameters, and other features.

EJB QL

API

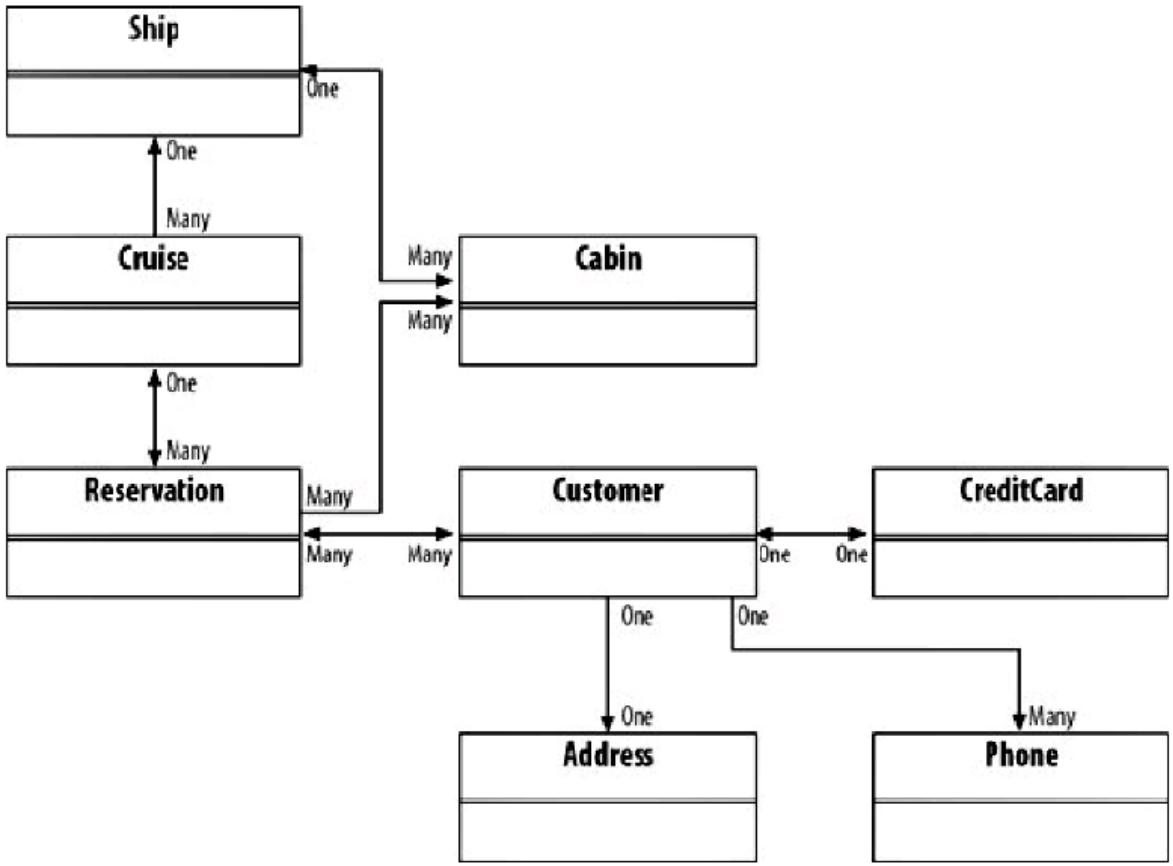
The API offers single result and list result retrieval, bulk updates, paging, named parameters, and vendor-specific hints.

```
package javax.persistence;
public interface Query {
  public List getResultList( );
  public Object getSingleResult( );
  public int executeUpdate( );
  public Query setMaxResults(int maxResult);
  public Query setFirstResult(int startPosition);
  public Query setHint(String hintName, Object value);
  public Query setParameter(String name, Object value);
  public Query setParameter(String name, Date value, TemporalType temporalType);
  public Query setParameter(String name, Calendar value,
                            TemporalType temporalType);
  public Query setParameter(int position, Object value);
  public Query setParameter(int position, Date value, TemporalType temporalType);
  public Query setParameter(int position, Calendar value,
                            TemporalType temporalType);
  public Query setFlushMode(FlushModeType flushMode);
}
```

Queries are created through the `EntityManager`:

```
package javax.persistence;
public interface EntityManager {
  public Query createQuery(String ejbqlString);
  public Query createNamedQuery(String name);
  public Query createNativeQuery(String sqlString);
  public Query createNativeQuery(String sqlString, Class resultClass);
  public Query createNativeQuery(String sqlString, String resultSetMapping);
}
```

Schema for some example queries:



Burke and Monson-Haefel (2006), Fig. 9-1

EJB QL

Simple example

```
SELECT c FROM Customer AS c
```

The `FROM` clause specifies what entity bean types are to be considered. By default, the schema name is the same as the class, but it can be changed in the `Entity` annotation (e.g. `@Entity(name='"Cust")`.

The `AS` clause associates the identifier `c` with each `Customer` entity. Identifiers are case insensitive and cannot clash with schema names.

Suppose the Customer class looks like this:

```
@Entity
public class Customer {
  private ind id;
  private String first;
  private String last;
  @Id
  public int getId( ) { return id; }
  public String getFirstName( ) { return first; }
  public String getLastName( ) { return first; }
```

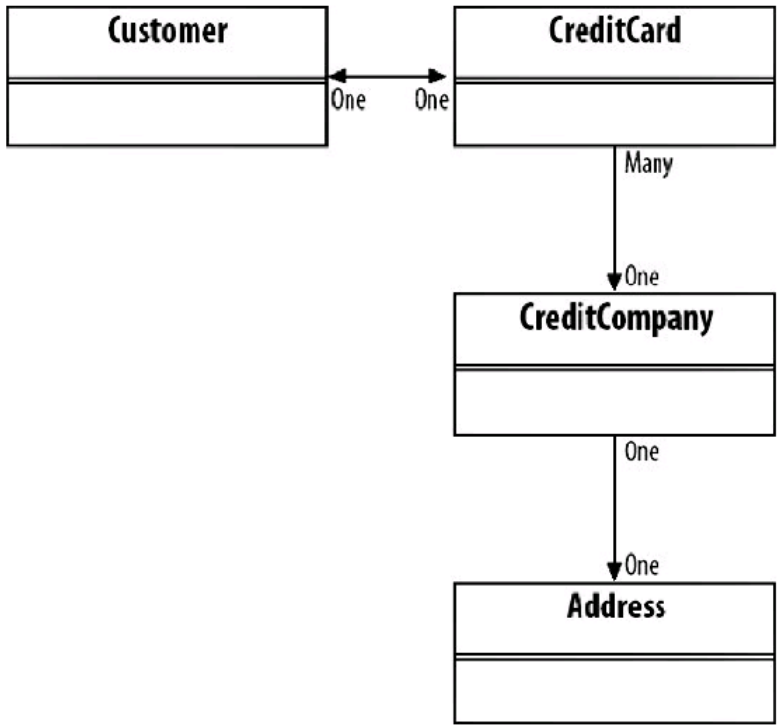
```
SELECT c.firstName, c.lastName FROM Customer as c
```

Property names are derived from the field name or getter name, depending on how you annotate the class declaration.

To execute and use the results of this query:

```
Query query = manager.createQuery(
              "SELECT c.firstName, c.lastName FROM Customer AS c");
List results = query.getResultList( );
Iterator it = results.iterator( );
while (it.hasNext( )) {
   Object[] result = (Object[])it.next( );
   String first = (String)result[0];
   String last = (String)result[1];
}
```

Now let's extend the object schema:



Burke and Monson-Haefel (2006), Fig. 9-2

EJB QL

Navigating properties

We can traverse any single-valued association:

```
SELECT c.creditCard FROM Customer AS c
SELECT c.address.city FROM Customer AS c

SELECT c.creditCard.creditCompany.address FROM Customer AS c
```

But collection-valued assocations require joins (coming up).

EJB QL

Navigating properties

Also, we are only allowed to navigate persistent properties:

```
public class ZipCode {
  public int mainCode;
  public int codeSuffix;
  ...
}

@Entity
public class Address {
    private ZipCode zip;
}
```

`SELECT c.address.zip.mainCode FROM Customer AS c` [illegal!]

Adding the `@EmbeddedValue` annotation allows us to query subfields.

Constructor expressions are powerful:

```
public class Name {
  private String first;
  private String last;
  public Name(String first, String last) {
    this.first = first;
    this.last = last;
  }
  public String getFirst( ) { return this.first; }
  public String getLast( ) { return this.last; }
}
```

```
SELECT new com.titan.domain.Name(c.firstName, c.lastName) FROM Customer c
```

For collection-valued properties, we use joins rather than property navigation.

The `IN` operator specifies an inner join:

```
SELECT r FROM Customer AS c, IN( c.reservations ) r
```

or, equivalently, using the `INNER JOIN` syntax:

```
SELECT r FROM Customer AS c, INNER JOIN c.reservations r
```

A more complex example:

```
SELECT cbn.ship FROM Customer
AS c, IN( c.reservations ) r, IN( r.cabins ) cbn
```

A left outer join pairs elements on the left of the join with the value `null` when there is no corresponding element on the right.

```
SELECT c.firstName, c.lastName, p.number
FROM Customer c LEFT JOIN c.phoneNumbers p
```

This query might return the results

```
David Ortiz 617-555-0900
David Ortiz 617-555-9999
Trot Nixon 781-555-2323
Bill Burke null
```

EJB QL

Joins

Sometimes when we have a lazy-loaded collection association:

```
@OneToMany(fetch=FetchType.LAZY)
public Collection<Phone> getPhones( ) { return phones; }
```

But in some transactions, we might need to access each of the associated objects:

```
Query query = manager.createQuery("SELECT c FROM Customer c");
List results = query.getResultList( );
Iterator it = results.iterator( );
while (it.hasNext( )) {
   Customer c = (Customer)it.next( );
   System.out.print(c.getFirstName( ) + " " + c.getLastName( ));
   for (Phone p : c.getPhoneNumbers( )) {
      System.out.print(" " + p.getNumber( ));
   }
   System.out.println("");
}
```

To avoid the N extra queries, we use a fetch join:

```
SELECT c FROM Customer c LEFT JOIN FETCH c.phones
```

EJB QL

WHERE clause

As in SQL, EJB QL allows limiting the scope of a query using `WHERE`.

Literals in `WHERE` clauses have the form:

- `'Matt'` (single quotes) for string literals.
- `321`, `-89`, `5E3`, `-8.9`, etc. for numeric literals.
- `TRUE` and `FALSE` for Boolean literals.

Standard relational operators (=, =, >, <, >=, <=, <>, `LIKE`, `BETWEEN`, `IS NULL`, `IS EMPTY`, `MEMBER OF`) are supported.

The arithmetic operators are +, -, *, and /.

The logical operators `AND`, `OR`, and `NOT` are also allowed.

EJB QL

WHERE clause

Some interesting comparison operators:

```
...  WHERE s.tonnage BETWEEN 8000 and 10000

...  WHERE a.state IN ( 'FL', 'MI' )

...  WHERE a.state IS NULL

...  WHERE c.phoneNumbers IS EMPTY
```

```
SELECT crs
FROM Cruise AS crs,
IN( crs.reservations ) as res, Customer AS cust
WHERE cust = :myCust AND cust NOT MEMBER OF res.customers
```

EJB QL

Functions in a WHERE clause

Inside the WHERE clause, we can use LOWER, UPPER, TRIM, CONCAT, LENGTH, LOCATE, SUBSTRING, ABS, SQRT, MOD, CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP.

EJB QL

Aggregate functions in a SELECT clause

Within the SELECT clause, we can use COUNT, MAX, AVG, and DISTINCT.

EJB QL

Other features

Similar to SQL, EJB QL supports `ORDER BY`, `GROUP BY`, and `HAVING`.

As in SQL, subqueries are allowed:

```
SELECT COUNT(res) FROM Reservation res
WHERE res.amountPaid > ( SELECT avg(r.amountPaid) FROM
Reservation r )
```

The operators `ALL`, `ANY`, `SOME`, and `EXISTS` can be applied to the result of a subquery.

It is also possible to avoid writing update and delete code in Java:

```
UPDATE Reservation res SET res.amountPaid =
(res.amountPaid+10)
WHERE EXISTS ( SELECT c FROM res.customers AS c WHERE
c.firstName='Bill' AND c.lastName='Burke' )
```

(DELETE is analagous)

But be careful to note that these operations are on the database, so it might cause in-memory instances to become inconsistent.

This inconsistency can be avoided by using `EntityManager.clear()` to detach all active instances in the persistence context.

EJB QL

Named queries

```
@NamedQueries({
   @NamedQuery(name="getAverageReservation",
          query=
           "SELECT AVG( r.amountPaid )
            FROM Cruise As c, JOIN c.reservations r
            WHERE c = :cruise"),
   @NamedQuery(name="findFullyPaidCruises",
          query=
           "FROM Cruise cr
            WHERE 0 < ALL (
              SELECT res.amountPaid from cr.reservations res
            )")
})
@Entity
public class Cruise {...}
```

Such named queries are simple to use:

```
Query query = em.createNamedQuery("getAverageReservation");
Query.setParameter("cruise", cruise);
```

Outline

Consider this business logic for `TravelAgent.bookPassage()`:

```
public TicketDO bookPassage(CreditCardDO card, double price)
   throws IncompleteConversationalState {

   if (customer == null || cruise == null || cabin == null) {
      throw new IncompleteConversationalState( );
   }
   try {
      Reservation reservation = new Reservation(customer, cruise, cabin, price);
      entityManager.persist(reservation);

      this.processPayment.byCredit(customer, card, price);

      TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
      return ticket;
   } catch(Exception e) {
      throw new EJBException(e);
   }
}
```

Transactions

Motivation

We would like the three steps to execute as a transaction.

Another example: when we withdraw money from an ATM, the deduction from the account and the spitting out of money should be a transaction.

The `bookPassage()` method should be ACID:

- Atomicity: if payment processing fails, we need to abort and remove the reservation. If ticketing fails, we need to remove the reservation and refund the money.
- Conistency: all foreign key references and other constraints need to be satisfied when the transaction finishes.
- Isolation: other bean instances should not be able to manipulate our new reservation until `bookPassage()` is finished.
- Durability: the database updates should be committed to disk before the transaction can be considered completed.

EJB provides two forms of transaction control:

- Declarative transaction managment means the container implicitly handles transactions based on the `@javax.ejb.TransactionAttribute` annotation or XML deployment descriptors.
- Explicit transaction management means the application code is responsible for starting, committing, and/or rolling back transactions).

Declarative transaction management tracks the transaction scope.

The scope of a transaction is the set of EJBs participating in the transaction.

In `bookPassage()`, we have TravelAgent, the EntityManager service, and ProcessPayment.

An ongoing transaction's scope can be propagated or extended to another EJB when one of its methods is invoked.

An exception might stop a transaction and may or may not cause rollback.

By default, all EJB methods are transactional.

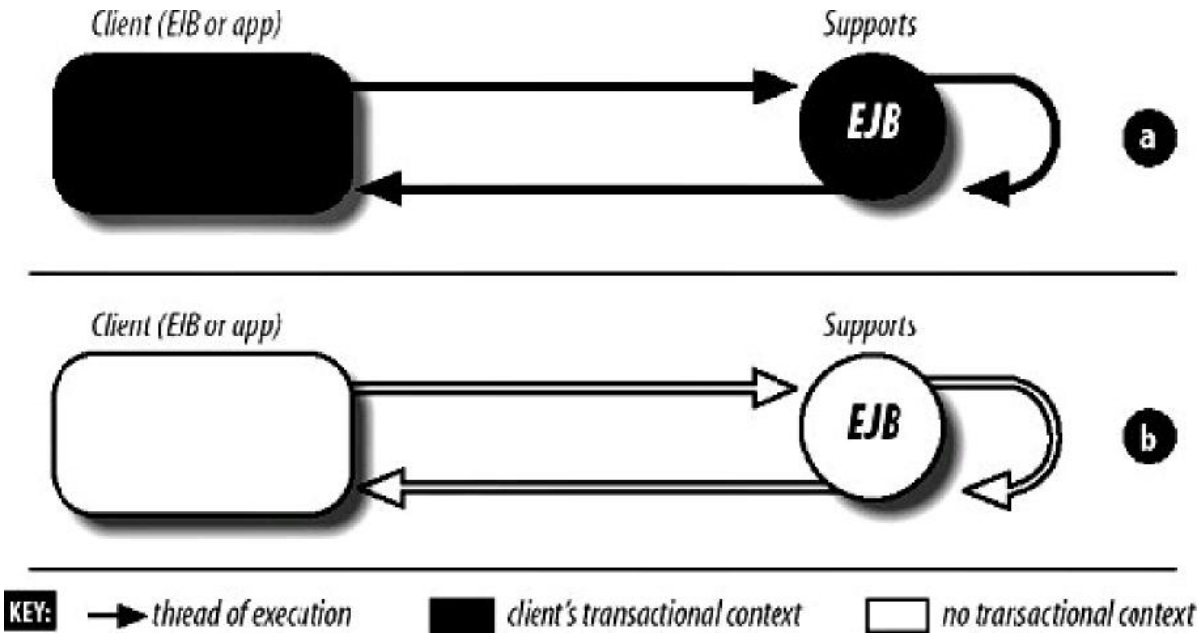
There are 6 possible transaction attributes:

- NotSupported: any ongoing transaction is suspended.
- Supports: if there is already a transaction scope for the client, it is extended. Otherwise no transaction is started.
- Required (the default): if there is already a transaction scope for the client, it is extended. Otherwise, a new transaction is started.
- RequiresNew: we start a new transaction. If the client has an ongoing transaction, it is suspended.
- Mandatory: if the client provides a transaction, it is extended. Otherwise an exception is generated.
- Never: the method can never be transactional. If the client provides a transaction, an exception is generated.

The NotSupported attribute:

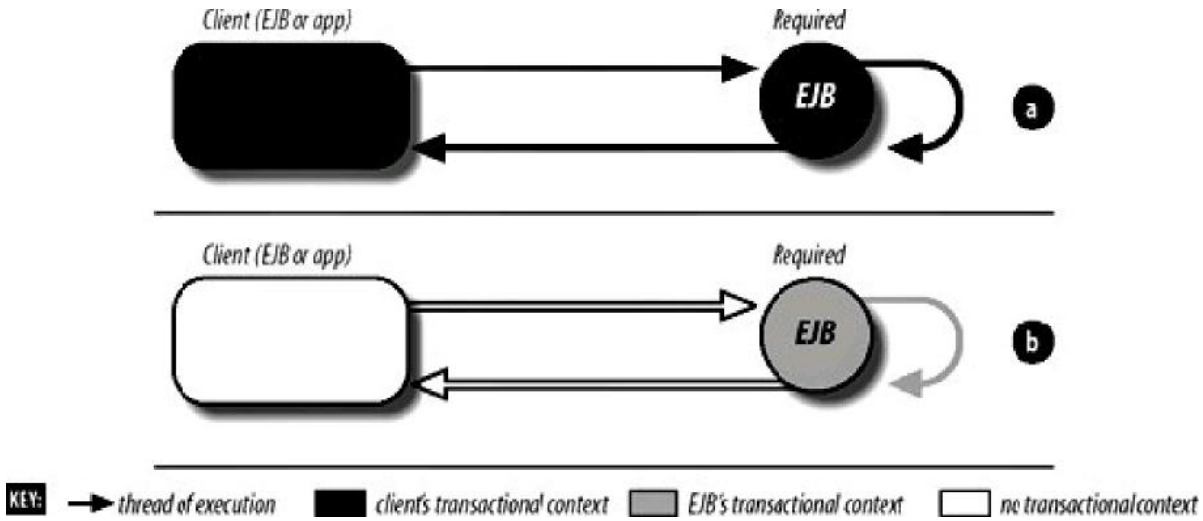
Burke and Monson-Haefel (2006), Fig. 16-1

The Supports attribute:



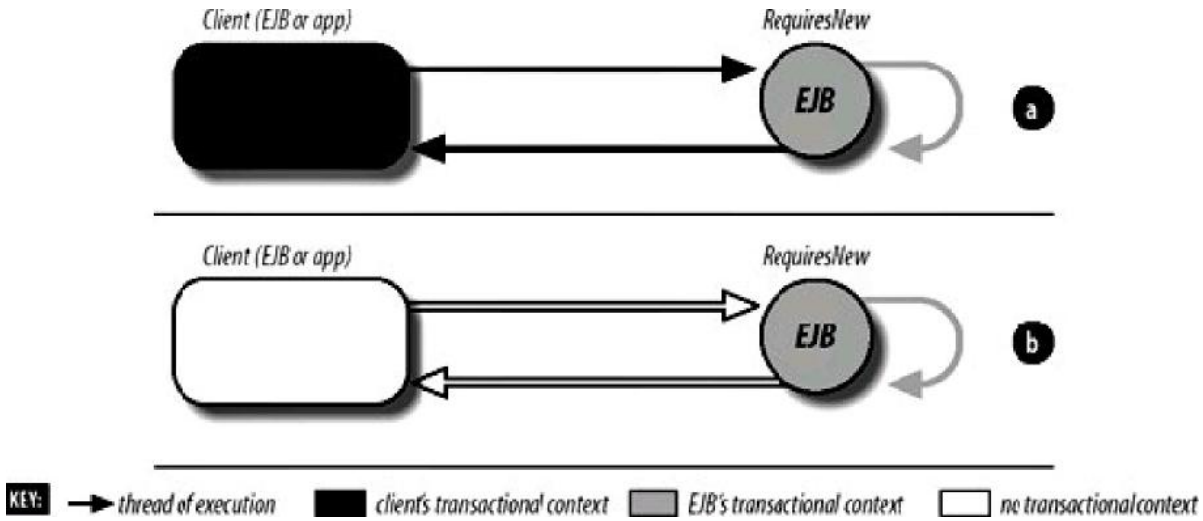
Burke and Monson-Haefel (2006), Fig. 16-2

The Required attribute:



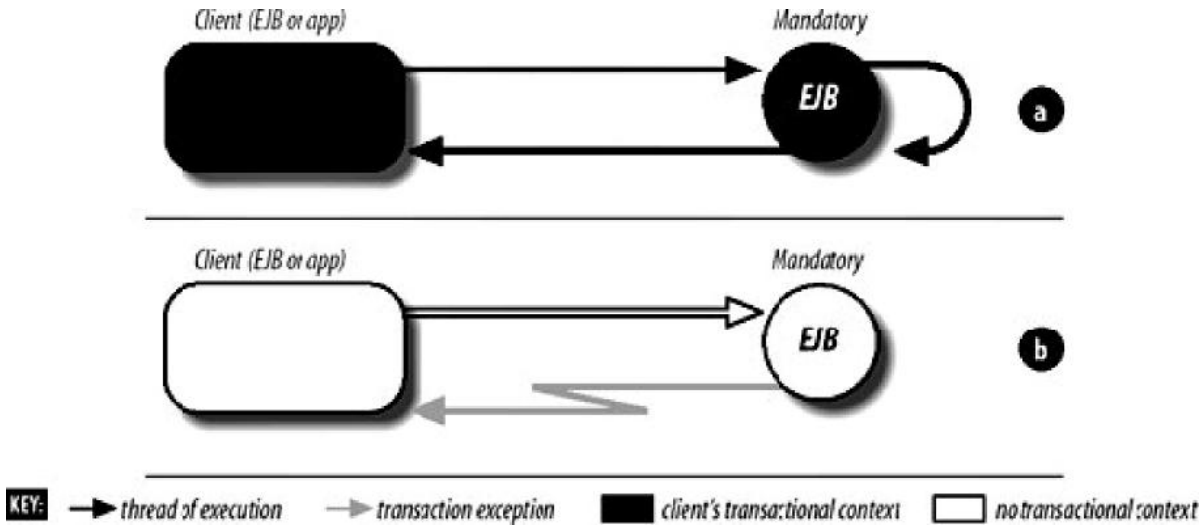
Burke and Monson-Haefel (2006), Fig. 16-3

The RequiresNew attribute:



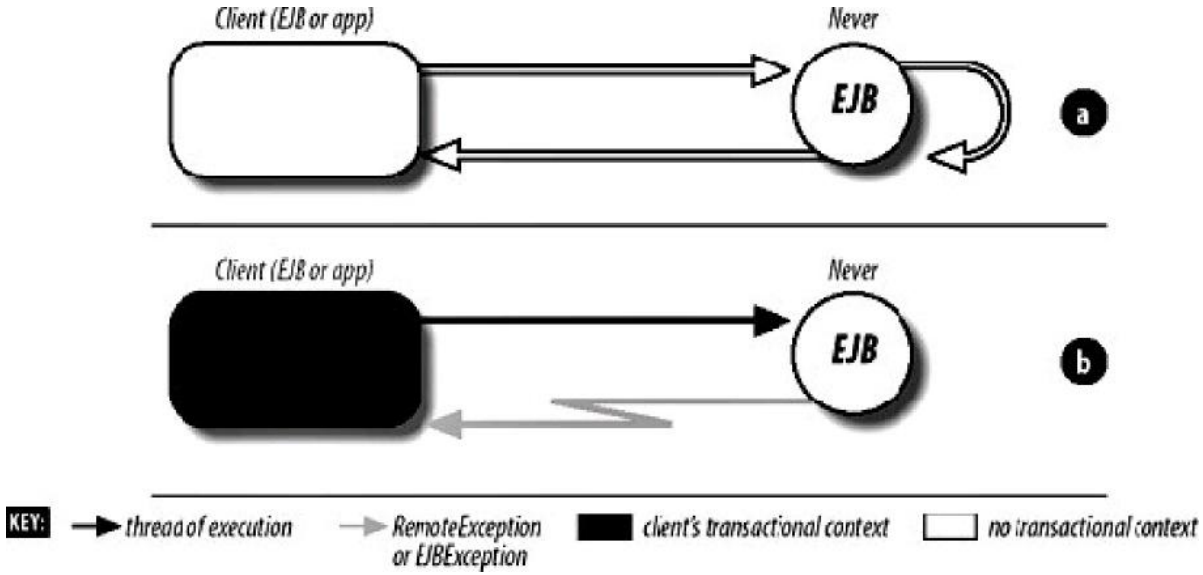
Burke and Monson-Haefel (2006), Fig. 16-4

The Mandatory attribute:



Burke and Monson-Haefel (2006), Fig. 16-5

The Never attribute:



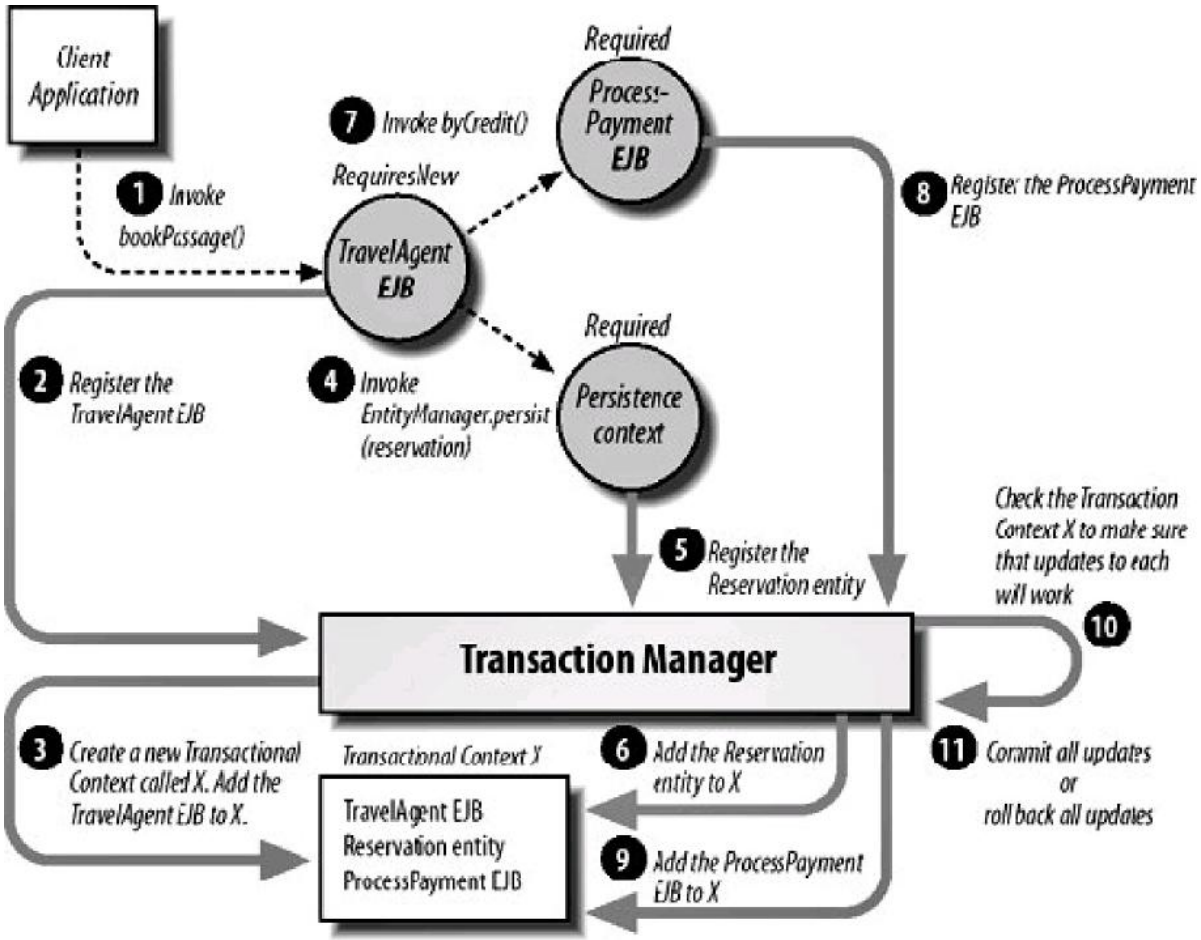
Burke and Monson-Haefel (2006), Fig. 16-6

When in a persistence context, the EJB designers strongly recommend Required, RequiresNew, or Mandatory transaction handling.

Transactions

Declarative transaction management

With declarative transaction management, the container monitors the transaction context and commits or rolls back at the end:



Burke and Monson-Haefel (2006), Fig. 16-7

If the transaction involves EJBs outside the local container, the server will use a distributed two-phase commit protocol to coordinate with the other servers, transparent to the application developer.

The standard levels of isolation are supported.

Optimistic locking is supported.

Although it is not recommended, explicit management of transactions is also supported. The declarative management system is designed to satisfy most applications.

Outline

Session beans

Introduction

Session beans describe taskflow (interactions between other beans) and implement specific tasks.

Session beans aren't mapped to a database but they can manipulate entities and local data as part of a business process.

When to use entities vs. session beans?

- Entity beans are used to provide a safe and consistent interface to a set of shared data defining a concept. They might be updated frequently. They model persistent data.
- Session beans model business logic that spans multiple concepts. They are useful as controllers and Remote Facades.

Session beans

Introduction

Session beans can be stateless or stateful.

- A stateless session bean is a collection of related services, each represented by a method, without any state maintained from one invocation to the next.
- A stateful session bean is an extension of the client application. It performs tasks for a specific client and maintains the client's state. The state is called the conversational state because it represents a continuing conversation between the stateful session bean and the client. Methods invoked on a stateful session bean read and write the conversational state.

Stateful session beans might time out, depending on the vendor. Stateless session beans can live forever, since they can be reused. A stateless session bean can also timeout, and it can be detached by its client, but it doesn't die, it's just returned to the pool of waiting beans.

A stateless session bean is lightweight and fast. A stateless session bean is similar to a procedure call in a transactional system.

In fact, session dependencies can be injected (more on this later), but conceptually, any injected state is not associated with the bean until invocation.

Likewise, a stateless session bean can have instance variables and have internal state that is maintained for the duration of a single invocation.

But neither form of state will ever be visible to the client.

As a general rule, if a request can be processed in one method call, it should be a stateless session bean method.

In the cruise ship example, ProcessPayment is a good candidate for a stateless session bean.

The ProcessPayment bean encapsulates all the logic related to charging customers for goods and services. We want to store each payment in a special database table called `PAYMENT`. Various business processes requiring payment will insert data into the `PAYMENT` table, and the accounting department will do batch processing on these payments.

We decide that since processing a payment can be done in a single method call, and that the payment data is not shared or frequently updated, we can use a stateless session bean to handle payments. Note the mix of relational data with OO technology.

Session beans

Example scenario for a session bean

```
CREATE TABLE PAYMENT
(
  customer_id      INTEGER,
  amount           DECIMAL(8,2),
  type             CHAR(10),
  check_bar_code   CHAR(50),
  check_number     INTEGER,
  credit_number    CHAR(20),
  credit_exp_date DATE
)
```

Session beans have one or more business interfaces that provide access to the business logic implemented by the bean.

In our case we have 3 main ways to pay, so we need 3 methods: `byCredit()`, `byCash()`, and `byCheck()`.

A business interface can be local or remote, but it cannot be both.

When a client invokes a remote interface, we should think of the parameters as being copied (the semantics is call by value).

Local interfaces are available to other objects within the same JVM as the session bean. Local interfaces use call by reference semantics.

If all of an interface's users are in the same deployment, they should be using call by reference for efficiency's sake (note that fine-grained interfaces don't apply so much for stateless session beans). So oftentimes, a stateless session bean will have a local and remote interface with the same API.

Here's the general interface for the ProcessPayment EJB:

```
package com.titan.processpayment;

import com.titan.domain.*;

public interface ProcessPayment {
   public boolean byCheck(Customer customer, CheckDO check, double amount)
      throws PaymentException;
   public boolean byCash(Customer customer, double amount)
      throws PaymentException;
   public boolean byCredit(Customer customer, CreditCardDO card,
      double amount) throws PaymentException;
}
```

The local and remote business interfaces will extend this base interface. Note that the methods are completely independent and all the information needed for processing is represented by the arguments.

Here are the local and remote interfaces:

```
package com.titan.processpayment;

import javax.ejb.Remote;

@Remote
public interface ProcessPaymentRemote extends ProcessPayment {
}
```

```
package com.titan.processpayment;

import javax.ejb.Local;

@Local
public interface ProcessPaymentLocal extends ProcessPayment {
}
```

Some notes on passing entities as parameters:

- Note that all of the methods take a `Customer` entity as a parameter.
- In EJB 3.0 entities are just POJOs, so in the case of the remote interface, the `Customer` object will have simply been serialized then reconstructed.
- In EJB 2.1 entites were first class components, so calls to a remote `Customer` would have been remote method invocations. Not so here; EJB 3.0 entities are much simpler.

The interfaces use two non-bean domain objects, `CreditCardDO` and `CheckDO`:

```
/* CreditCardDO.java */
package com.titan.processpayment;

import java.util.Date;

public class CreditCardDO implements java.io.Serializable {
   final static public String MASTER_CARD = "MASTER_CARD";
   final static public String VISA = "VISA";
   final static public String AMERICAN_EXPRESS = "AMERICAN_EXPRESS";
   final static public String DISCOVER = "DISCOVER";
   final static public String DINERS_CARD = "DINERS_CLUB_CARD";

   public String number;
   public Date expiration;
   public String type;
```

```
    public CreditCardDO(String nmbr, Date exp, String typ) {
        number = nmbr;
        expiration = exp;
        type = typ;
    }
}

/* CheckDO.java */
package com.titan.processpayment;

public class CheckDO implements java.io.Serializable {
    public String checkBarCode;
    public int checkNumber;
    public CheckDO(String barCode, int number) {
        checkBarCode = barCode;
        checkNumber = number;
    }
}
```

As we see, domain objects don't necessarily need to be entity beans:

- In the Titan application, credit card and personal check information is not kept persistent.

Some exceptions are generated by the container, some are generated by other Java subsystems, and some are generated by your application.

An application exception describes a problem in the business logic, in our case a problem making a payment.

Other exceptions, like `NamingException` and `SQLException`, have nothing to do with the business logic, and are generated by other subsystems.

Generally non-application exceptions should be caught and either dealt with or wrapped by an application exception.

There is an important distinction between checked and unchecked exceptions:

- Checked exceptions (non-runtime exceptions) are checked by the compiler (`throws` and/or `catch` are required) and by default don't cause rollback of persistence transactions.
- Runtime exceptions are unchecked by the compiler and do cause automatic rollbacks if thrown. Sometimes you will want to catch and handle runtime exceptions, and sometimes you will want to propagate them as application exceptions.

Here's an example of a `PaymentException` for the ProcessPayment EJB:

```
package com.titan.processpayment;

public class PaymentException extends java.lang.Exception {
   public PaymentException( ) {
      super( );
   }
   public PaymentException(String msg) {
      super(msg);
   }
}
```

Application exceptions should extend `java.lang.Exception`.
Non-application exceptions (extenders of `RuntimeException`) that you don't catch will be wrapped by an `EJBException`.

Any exception thrown by JPA is a `RuntimeException`. The relationship between exceptions and transactions is configurable.

Now let's take a look at the complete stateless ProcessPayment EJB:

```
package com.titan.processpayment;

@Stateless
public class ProcessPaymentBean
        implements ProcessPaymentRemote, ProcessPaymentLocal {

   final public static String CASH = "CASH";
   final public static String CREDIT = "CREDIT";
   final public static String CHECK = "CHECK";

   // Direct access to the JDBC connection pool
   @Resource(mappedName="java:/DefaultDS") DataSource dataSource;
   @Resource(name="min") int minCheckNumber = 100;
```

Session beans

Example stateless session bean

```
public boolean byCash(Customer customer, double amount)
   throws PaymentException {
   return process(customer.getId( ), amount, CASH, null, -1, null, null);
}

public boolean byCheck(Customer customer, CheckDO check, double amount)
      throws PaymentException {
   if (check.checkNumber > minCheckNumber) {
      return process(customer.getId( ), amount, CHECK,
         check.checkBarCode, check.checkNumber, null, null);
   } else {
      throw new PaymentException(
         "Check number is too low. Must be at least "+minCheckNumber);
   }
}
```

Session beans

Example stateless session bean

```
public boolean byCredit(Customer customer, CreditCardDO card,
      double amount) throws PaymentException {
   if (card.expiration.before(new java.util.Date( ))) {
      throw new PaymentException("Expiration date has passed");
   } else {
      return process(customer.getId( ), amount, CREDIT, null,
            -1, card.number, new java.sql.Date(card.expiration.getTime( )));
   }
}
```

Session beans

Example stateless session bean

```
private boolean process(int customerID, double amount, String type,
      String checkBarCode, int checkNumber, String creditNumber,
      java.sql.Date creditExpDate) throws PaymentException {
   Connection con = null;
   PreparedStatement ps = null;
   try {
      con = dataSource.getConnection( );
      ps = con.prepareStatement(
            "INSERT INTO payment (customer_id, amount, type,"+
            "check_bar_code,check_number,credit_number,"+
            "credit_exp_date) VALUES (?,?,?,?,?,?,?)");
      ps.setInt(1,customerID);
      ps.setDouble(2,amount);
      ps.setString(3,type);
      ps.setString(4,checkBarCode);
      ps.setInt(5,checkNumber);
      ps.setString(6,creditNumber);
      ps.setDate(7,creditExpDate);
      int retVal = ps.executeUpdate( );
```

Session beans

Example stateless session bean

```
            if (retVal!=1) {
               throw new EJBException("Payment insert failed");
            }
            return true;
         } catch(SQLException sql) {
            throw new EJBException(sql);
         } finally {
            try {
               if (ps != null) ps.close( );
               if (con!= null) con.close( );
            } catch(SQLException se) {
               se.printStackTrace( );
            }
         }
      }
}
```

The `process()` method inserts any kind of payment into the relational database table `payment`.

The database connection uses the `datasource` field. This field (and `minCheckNumber`) are injected by the EJB environment.

Every EJB container has an internal registry called the Enterprise Naming Context (ENC) where it stores configuration values and references to external resources.

The `@javax.annotation.Resource` annotation tells EJB that when an instance of the ProcessPayment bean is created, it should look in the ENC for the necessary value.

Session beans

Example stateless session bean

To set a configuration value like `minCheckNumber`, we put it in the XML deployment descriptor for the EJB:

```
<ejb-jar
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                        http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
    version="3.0">
  <enterprise-beans>
    <session>
      <ejb-name>ProcessPaymentBean</ejb-name>
      <env-entry>
        <env-entry-name>min</env-entry-name>
        <env-entry-type>java.lang.Integer</env-entry-type>
        <env-entry-value>250</env-entry-value>
      </env-entry>
    </session>
  </enterprise-beans>
</ejb-jar>
```

The XML descriptor would override the default of 100 specified in the code. Any constants used in your application should normally be configurable rather than hardcoded.

Resource injection is one of the areas where different EJB servers will be different. E.g. the `"java:/DefaultDS"` data source name is specific to JBoss. Portable code should use XML descriptors for the vendor-specific information.

The SessionContext interface allows an EJB to get access to the container's environment. To get the session context, just ask for the SessionContext @Resource:

```
@Stateless
public class ProcessPaymentBean implements ProcessPaymentLocal {
   @Resource SessionContext ctx;
   ...
}
```

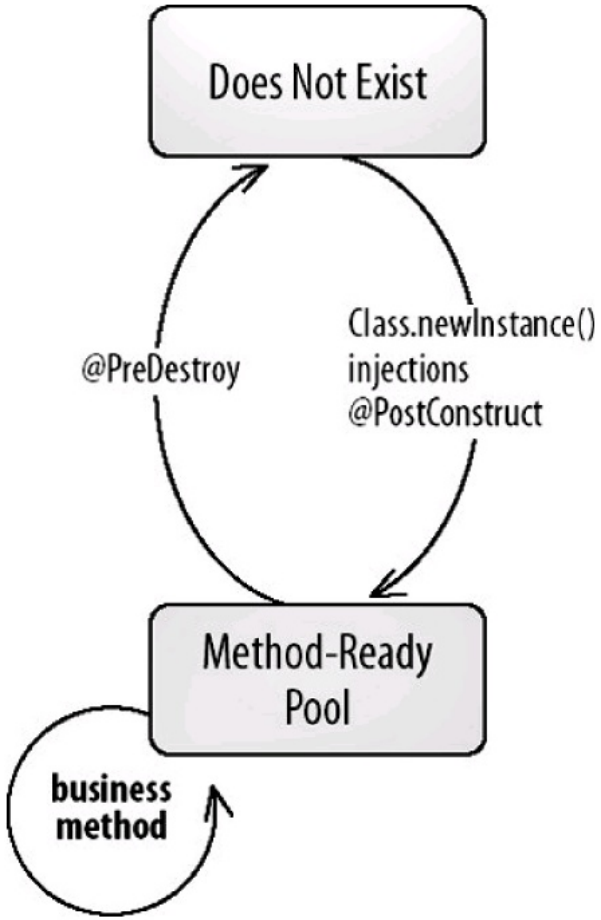
The SessionContext can give you information:

- `getBusinessObject()` which returns a reference to the calling EJB. So you can get a reference to yourself to pass to other objects, similar to `this` for a POJO.
- `lookup()` allows lookup in the container's ENC.
- `getTimerService()` to get a timer service object.
- `getCallerPrincipal()` to get the identify of the calling client invoking your bean, if you have security set up.
- `isCallerInRole()` to determine if the caller is a member of a specific role (access control group).
- Methods for getting and setting properties of the current transaction.

Session beans

Stateless session bean life cycle

Stateless session beans have a simple life cycle:



Burke and Monson-Haefel (2006), Fig. 11-1.

We pool stateless session beans due to their long startup time, caused mainly by injections.

A bean instance doesn't exist until it is created with the bean class' `newInstance()` method.

If the bean wants to do something special (connect to resources) after it's constructed, it can annotate a method with the `@PostConstruct` annotation to do so. These resources should be disconnected in the `@PreDestroy` method.

Death of a stateless session bean generally only occurs when the server decides it has more ready instances than necessary.

Stateful session beans behave very differently from stateless session beans. From the application programmer's view:

- A stateful session bean instance is dedicated to one client for its life.
- There is no swapping or instance pools for stateful session beans.

Stateful session beans allow maintenance of session state, but are much more costly to invoke than stateless session beans.

The two main uses of stateful session beans:

- As an extension of the presentation layer state, making the client thinner and easier to manage;
- To make taskflow modeling simple.

As an example, we'll consider the TravelAgent EJB for the Titan cruise operation. This bean needs to be stateful.

Before looking at the details of the bean class, we'll look at the entities, application exceptions, and DOs it uses.

Session beans

Entity bean for stateful session bean example

The `Reservation` entity uses a rich constructor to make creation code less cluttered:

```
@Entity
public class Reservation {
   ...

   public Reservation( ) {}

   public Reservation(Customer customer, Cruise cruise,
         Cabin cabin, double price, Date dateBooked) {
      setAmountPaid(price);
      setDate(dateBooked);
      setCruise(cruise);
      Set cabins = new HashSet( );
      cabins.add(cabin);
      this.setCabins(cabins);
      Set customers = new HashSet( );
      customers.add(customer);
      this.setCustomers(customers);
   }
}
```

Session beans

Remote interface for stateful session bean example

We only need a remote interface for this bean:

```
package com.titan.travelagent;

import com.titan.processpayment.CreditCardDO;
import javax.ejb.Remote;
import com.titan.domain.Customer;

@Remote
public interface TravelAgentRemote {
   public Customer findOrCreateCustomer(String first, String last);
   public void updateAddress(Address addr);
   public void setCruiseID(int cruise);
   public void setCabinID(int cabin);
   public TicketDO bookPassage(CreditCardDO card, double price)
     throws IncompleteConversationalState;
}
```

The client should set the customer, cruise, and cabin then ask the TravelAgent EJB to process the reservation by calling bookPassage().

Session beans

Application exception for stateful session bean example

Application exceptions are used to communicate business logic problems. We want to alert the client when there is a problem booking a customer on a cruise:

```
package com.titan.travelagent;

public class IncompleteConversationalState extends java.lang.Exception {
  public IncompleteConversationalState( ){super( );}
  public IncompleteConversationalState(String msg){super(msg);}
}
```

Application exceptions are those not extending `RuntimeException` or `RemoteException`.

By default, application exceptions do not cause persistence manager rollbacks, but this behavior can be altered by annotating the exception class with `@ApplicationException(rollback=true)`.

Session beans

Domain object for stateful session bean example

When a reservation is booked, we want to return some kind of ticket to the remote client.

The ticket needs to contain the relevant details of the reservation. Why not just serialize the reservation itself?

- It is a form of coupling
- Oftentimes, an entity bean will contain a lot of extraneous detail not needed by the client

In this case, similar to Fowler's DTOs, we choose to use a DO to encapsulate just the necessary information as a `Ticket`.

```
package com.titan.travelagent;

import com.titan.domain.Cruise;
import com.titan.domain.Cabin;
import com.titan.domain.Customer;

public class TicketDO implements java.io.Serializable {
```

Session beans

Domain object for stateful session bean example

```
    public int customerID;
    public int cruiseID;
    public int cabinID;
    public double price;
    public String description;
    public TicketDO(Customer customer, Cruise cruise, Cabin cabin,
          double price) {
       description = customer.getFirstName( ) + " " + customer.getLastName( ) +
                     " has been booked for the " + cruise.getName( ) +
                     " cruise on ship " + cruise.getShip( ).getName( ) + ".\n" +
                     " Your accommodations include " + cabin.getName( ) + " a " +
                     cabin.getBedCount( ) + " bed cabin on deck level " +
                     cabin.getDeckLevel( ) + ".\n Total charge = " + price;
       customerID = customer.getId( );
       cruiseID = cruise.getId( );
       cabinID = cabin.getId( );
       this.price = price;
    }
    public String toString( ) {
       return description;
    }
}
```

Session beans

Client code for stateful session bean example

In the client, we imagine there will be a Java application with GUI fields. It will look up our TravelAgent EJB in JNDI:

```
Context jndi = getInitialContext( );
Object ref = jndi.lookup("TravelAgentBean/remote");
TravelAgentRemote agent = (TravelAgentRemote)
      PortableRemoteObject.narrow(ref, TravelAgentRemote.class);
Customer cust = agent.findOrCreateCustomer(textField_firstName.getText( ),
                  textField_lastName.getText( ));
```

Note: everytime any client looks up a stateful session bean in JNDI, a new EJB object and session is created.

If the customer has changed address, the real-life travel agent will want to update it:

```
Address updatedAddress = new Address(textField_street.getText( ), ...);
agent.updateAddress(updatedAddress);
```

Then we'll get the cruise ID and cabin choice from the GUI and set them in the TravelAgent bean:

```
Integer cruise_id = new Integer(textField_cruiseNumber.getText( ));
Integer cabin_id = new Integer( textField_cabinNumber.getText( ));
agent.setCruiseID(cruise_id);
agent.setCabinID(cabin_id);
```

Finally, we'll get the customer's credit card info, create the credit card DO, and book passage:

```
String cardNumber = textField_cardNumber.getText( );
Date date = dateFormatter.parse(textField_cardExpiration.getText( ));
String cardBrand = textField_cardBrand.getText( );
CreditCardDO card = new CreditCardDO(cardNumber,date,cardBrand);
double price = double.valueOf(textField_cruisePrice.getText( )).doubleValue( );
TicketDO ticket = agent.bookPassage(card,price);
PrintingService.print(ticket);
```

Session beans

Bean class for stateful session bean example

Now that we know our interface and the client's expected behavior we can design the bean class:

```
package com.titan.travelagent;import com.titan.processpayment.*;

import com.titan.domain.*;
import javax.ejb.*;
import javax.persistence.*;
import javax.annotation.EJB;
import java.util.Date;

@Stateful
public class TravelAgentBean implements TravelAgentRemote {
   @PersistenceContext(unitName="titan") private EntityManager entityManager;
   @EJB private ProcessPaymentLocal processPayment;
   private Customer customer;
   private Cruise cruise;
   private Cabin cabin;
```

Session beans

Bean class for stateful session bean example

```
public Customer findOrCreateCustomer(String first, String last) {
   try {
      Query q = entityManager.createQuery(
         "from Customer c
          where c.firstName = :first and c.lastName = :last");
      q.setParameter("first", first);
      q.setParameter("last", last);
      this.customer = (Customer)q.getSingleResult( );
   } catch (NoResultException notFound) {
      this.customer = new Customer( );
      this.customer.setFirstName(first);
      this.customer.setLastName(last);
      entityManager.persist(this.customer);
   }
   return this.customer;
}
```

Session beans

Bean class for stateful session bean example

```
public void updateAddress(Address addr) {
   this.customer.setAddress(addr);
   this.customer = entityManager.merge(customer);
}

public void setCabinID(int cabinID) {
   this.cabin = entityManager.find(Cabin.class, cabinID);
   if (cabin == null) throw new NoResultException("Cabin not found");
}

public void setCruiseID(int cruiseID) {
   this.cruise = entityManager.find(Cruise.class, cruiseID);
   if (cruise == null) throw new NoResultException("Cruise not found");
}
```

Session beans

Bean class for stateful session bean example

```
    @Remove
    public TicketDO bookPassage(CreditCardDO card, double price)
          throws IncompleteConversationalState {
       if (customer == null || cruise == null || cabin == null) {
          throw new IncompleteConversationalState( );
       }
       try {
          Reservation reservation = new Reservation(
                customer, cruise, cabin, price, new Date( ));
          entityManager.persist(reservation);
          processPayment.byCredit(customer, card, price);
          TicketDO ticket = new TicketDO(customer, cruise, cabin, price);
          return ticket;
       } catch(Exception e) {
          throw new EJBException(e);
       }
    }
}
```

Session beans

Bean class for stateful session bean example

Comments on TravelAgent bean class:

- First, we ask the container to inject references to the persistence manager and the ProcessPayment EJB.
- `findOrCreateCustomer()` tries to retrieve the Customer entity for a given name. If the query fails, the `NoResult` will be thrown, in which case we create a new customer and persist it.
- `updateAddress()` updates the customer's address and asks the entity manager to synchronize with the database (remember the actual synchronization won't happen until the persistence manager flushes its state).
- The setters for the cruise and cabin query for the relevant cruise and cabin entities and set them. Using integer IDs in the GUI and bean instances in the EJB is a simple and common approach.
- Once `bookPassage()` completes, since it is annotated with `javax.ejb.@Remove`, the container will destroy the session bean instance and invalidate the session.

Session beans

Bean class for stateful session bean example

The example shows how encapsulating state in a session bean:

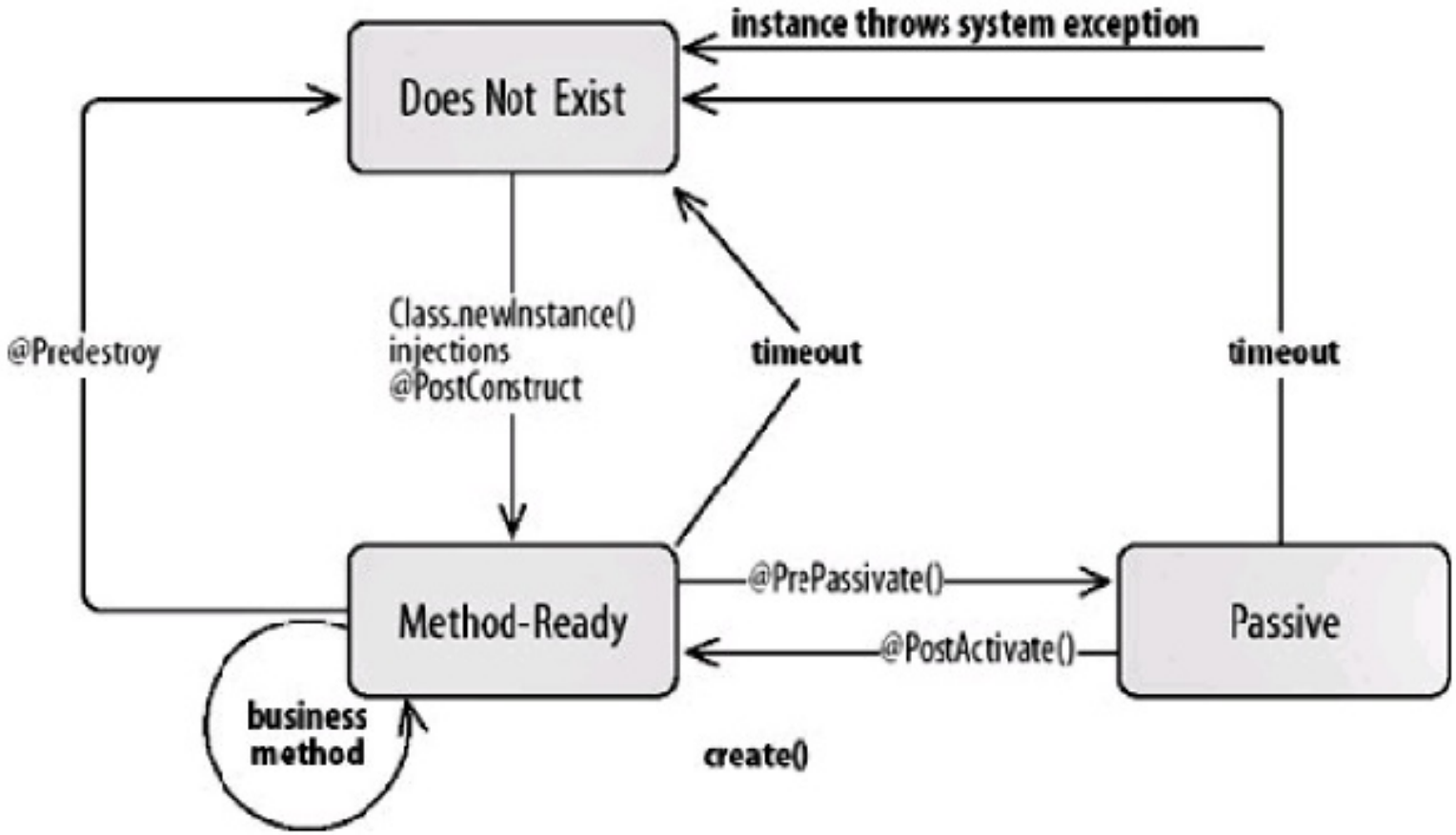
- Makes taskflow simple to implement server side.
- Makes life easier for the client.

The stateful bean also makes the client "thinner" so we could more easily replace the GUI presentation with a Web interface without changing the business logic layer.

Session beans

Stateful session bean life cycle

A state machine for stateful session bean instances:



Burke and Monson-Haefel (2006), Fig. 11-2.

The Does Not Exist state:

- Indicates the the bean instance is unborn or dead.
- Transitions to Method-Ready when the client invokes the first method on its session bean reference.

The Method-Ready state:

- Indicates the bean is ready to serve requests from its clients.
- The container invokes `newInstance()` on the bean class, to create the instance.
- Then the container injects any dependencies (`@Resources`, `@PersistenceContexts`, etc.).
- Then the container invokes any `@PostConstruct`-annotated method.
- While in the Method-Ready state, the bean instance receives method invocations from the client, maintains conversational state, and opens resources.
- The bean instance leaves Method-Ready when it is passivated by the container or removed through timeout or execution of a `@Remove` method.
- Any `@Predestroy` methods are invoked before final destruction.

In the Passive state:

- The bean instance is evicted to secondary storage and its resources are freed.
- Any serializable state information and any EJB-injected resources are automatically saved.
- The bean must manually tear down any non-serializable state in a `@PrePassivate` method and restore them in a `@PostActivate` method.
- Passivated bean instances can also be destroyed (moved to Does Not Exist) if they time out. `@PreDestroy` methods are not called in this case.

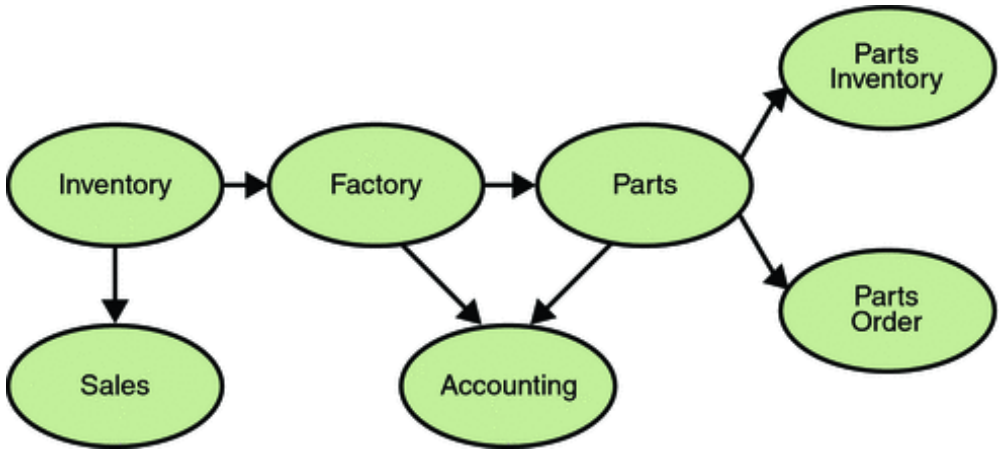
When a system exception (an exception including EJBException not marked as an `@ApplicationException`) occurs, the EJB object becomes invalid and the bean instance is destroyed. `@PreDestroy` is not invoked.

Outline

All Java EE containers are required to provide the Java Message Service (JMS) API.

JMS supports most enterprise application integration patterns.

It also provides asynchronous communication between components in the same application, e.g.:



Jendrock et al. (2006), Fig. 31-1

This material is from *The Java EE 5 Tutorial* from Sun.

JMS on Java EE supports these features:

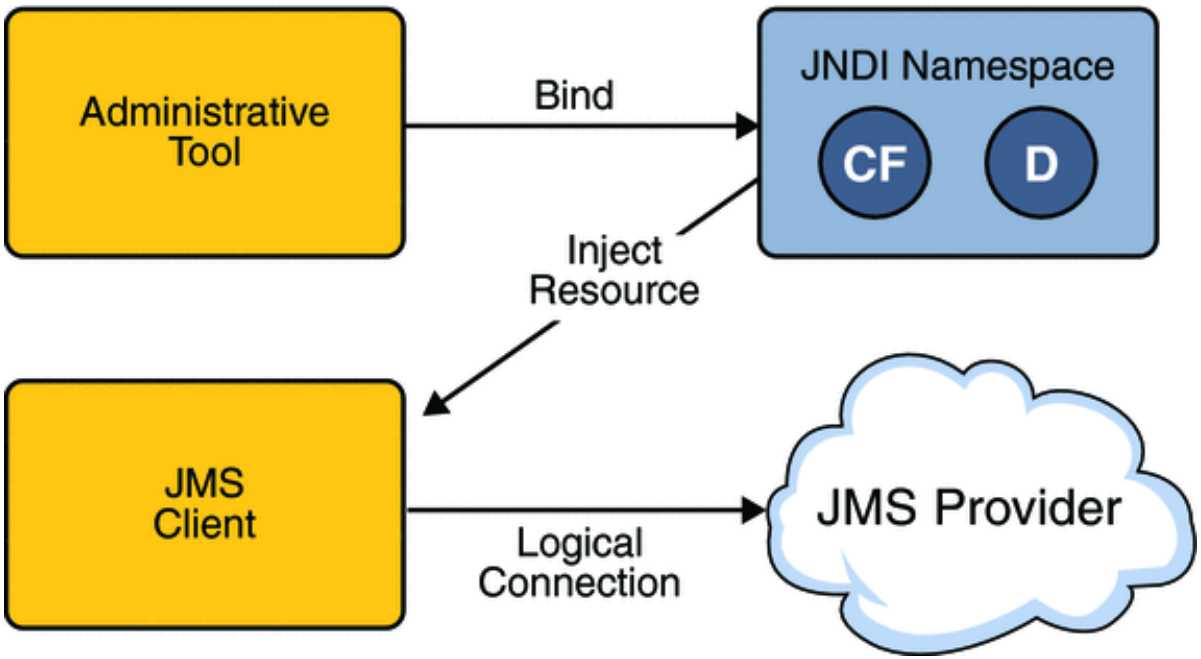
- Application clients, EJBs, and Web components can send or synchronously receive JMS messages. Application clients can also receive messages asynchronously.
- Message-Driven Beans enable asynchronous consumption of messages within the container.
- Message sends and receives can participate in distributed transactions, allowing JMS operations and database accesses to take place in the same atomic transaction.

The JMS API architecture is composed of several parts:

- A JMS provider is a Messaging system implementing the JMS interfaces and administrative control features. Every Java EE implementation includes a JMS provider.
- JMS clients are the Messaging Endpoints that produce and consume messages.
- Messages are objects that communicate information between JMS clients.
- Administered objects are objects created by the administrator for client use. They include `Destination` objects (Message Channels) and `ConnectionFactorys`.

Java Message Service

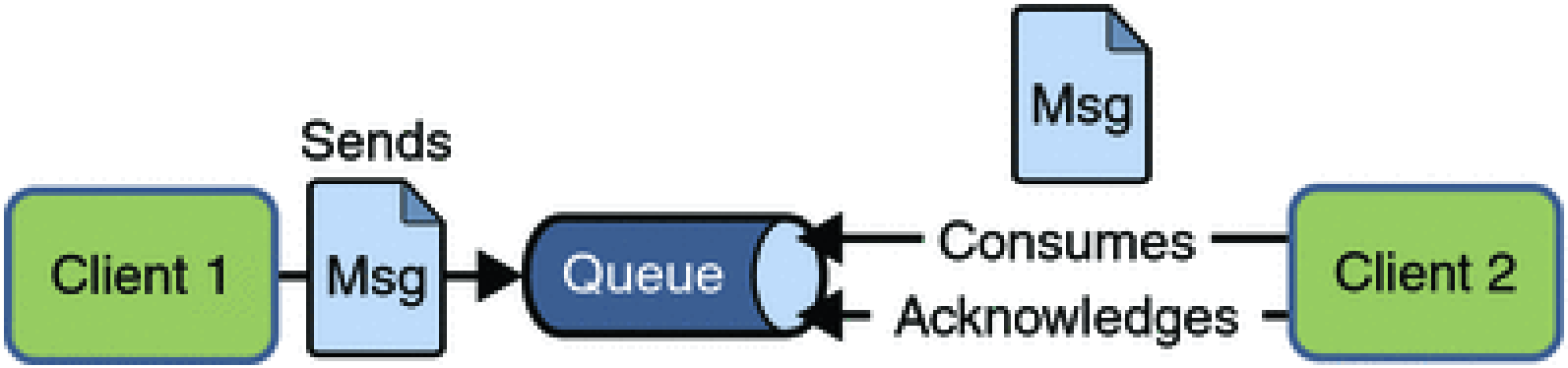
JMS Architecture



Jendrock et al. (2006), Fig. 31-2

JMS supports both Point-to-Point Channels and Publish-Subscribe Channels.

In JMS terminology, the channel types are separated into two messaging domains, the point-to-point domain and the publish/subscribe domain.



Jendrock et al. (2006), Fig. 31-3

As in the abstract Point-to-Point Channel, point-to-point (PTP) messages have ony one consumer.

Similar to Reliable Messaging, the sender and receiver do not depend on each other temporally. Receivers can receive messages sent when they were not running.

As in the Publish-Subscribe Channel, publish/subscribe (pub/sub) messaging allows multiple consumers.

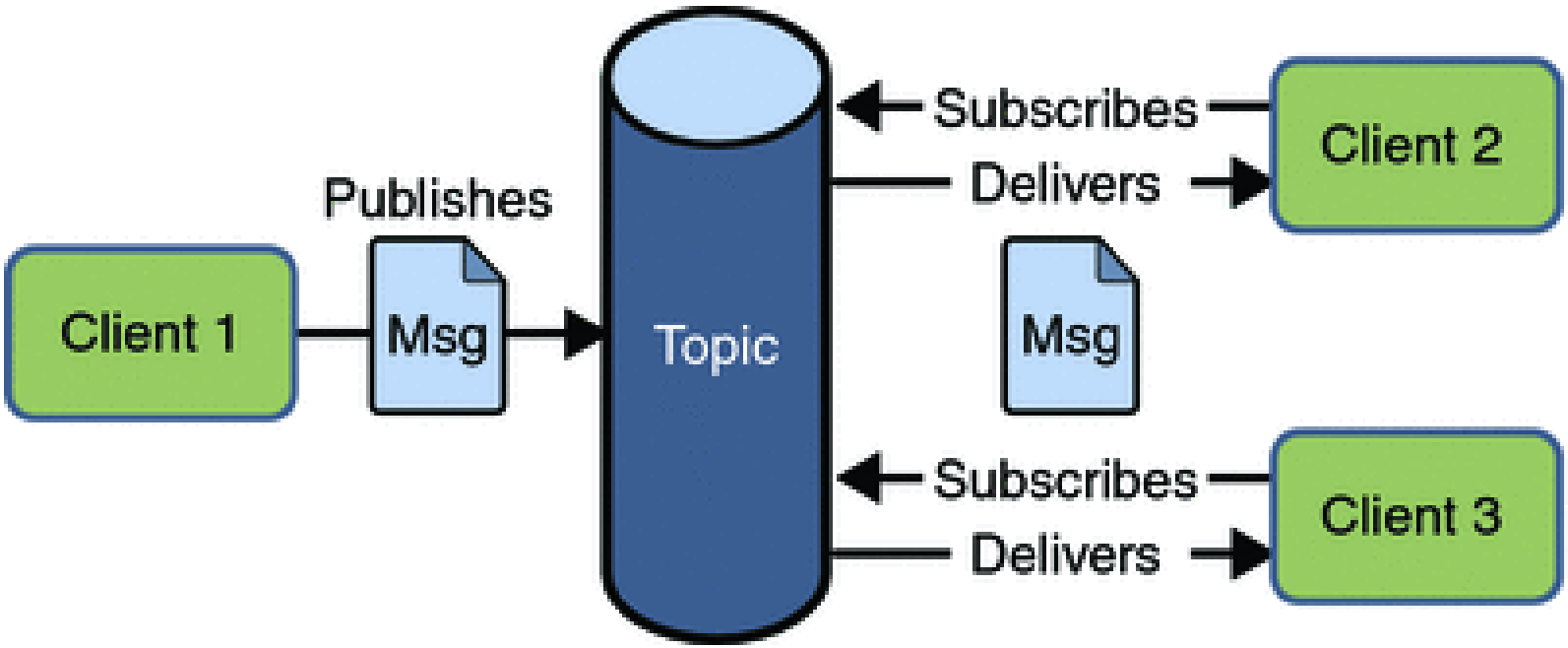
Pub/sub messaging differs from PTP messaging:

- A client subscribing to a `Topic` can only consume messages published after the subscription.
- Subscribers must remain active to continue consuming messages.

Durable subscriptions relax these restrictions, allowing receipt of messages sent while the subscriber is not active.

Java Message Service

Publish/subscribe domain



Jendrock et al. (2006), Fig. 31-4

The abstract `Destination` class provides a uniform interface to `Queues` and `Topics`.

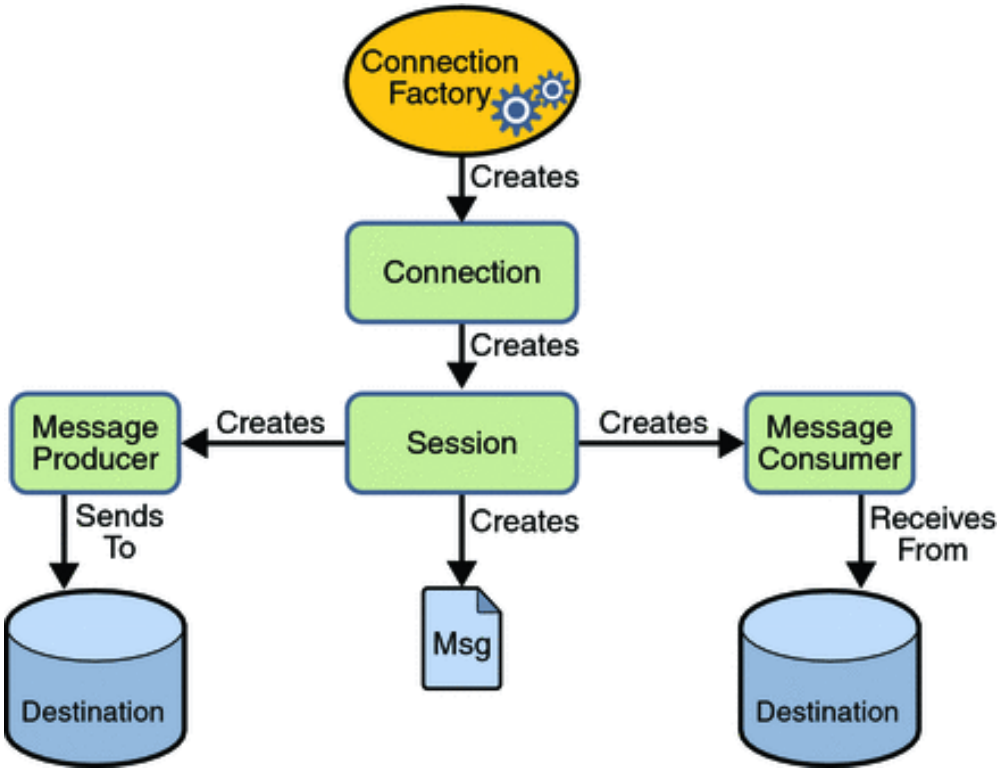
JMS supports both of the main mechanisms for receiving messages:

- For synchronous receipt, any component can use the `receive()` method, which allows blocking with or without a timeout.
- For asynchronous receipt, application clients can register a message listener implementing an `onMessage()` method with a consumer. Listener registration can be handled simply using message-driven EJBs.

Here are the building blocks of a JMS application:

- Administered objects (connection factories and destinations).
- Connections
- Sessions
- Message producers
- Message consumers
- Messages

Here is how the building blocks fit together:



Jendrock et al. (2006), Fig. 31-5

We'll look at each of these building blocks in turn.

JMS requires system administrators to create and set up JNDI names for `ConnectionFactory` and `Destination` objects:

- A connection factory allows clients to create connections to the JMS provider.

```
@Resource(mappedname="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
```

- A destination is the target of messages a client produces and the source of messages it consumes.

```
@Resource(mappedname="jms/Queue")
private static Queue queue;

@Resource(mappedname="jms/Topic")
private static Topic topic;
```

In Glassfish, you use `asadmin` or the admin Web console to manage administered objects.

A connection encapsulates a virtual connection to a JMS provider. Think of it as a TCP/IP socket between the application and the JMS provider.

Connections are used to create sessions.

```
Connection connection = connectionFactory.createConnection();
```

Connections should be explicitly closed by the client.

Message consumption does not begin until you call the connection instance's `start()` method.

You can temporarily suspend delivery using the connection instance's `stop()` method.

A session is a single-threaded context for producing and consuming messages.

Sessions can create the following:

- Message producers
- Message consumers
- Messages
- Queue browsers
- Temporary queues and topics

Session creation example (transactions off, receipt automatically acknowledged):

```
Session session = connection.createSession( false,
Session.AUTO_ACKNOWLEDGE );
```

Message producers are used to send messages.

```
MessageProducer producer = session.createProducer( dest );
MessageProducer producer = session.createProducer( queue );
MessageProducer producer = session.createProducer( topic );
```

Once you create a message (with the help of a `Session` object), you use the producer to send:

```
producer.send( message );
```

Java Message Service

JMS message consumers

A message consumer receives and processes messages.

```
MessageConsumer consumer = session.createConsumer( dest );
MessageConsumer consumer = session.createConsumer( queue );
MessageConsumer consumer = session.createConsumer( topic );
```

For durable subscribers in the pub/sub domain, use `Session.durableSubscriber()`.

Use `consumer.close()` to make a consumer inactive.

Message deliver does not begin until you call `connection.start()`.

Synchronous receives are straightforward:

```
connection.start();
Message m = consumer.receive();
connection.start();
Message m = consumer.receive( 1000 ); // time out after one second
```

Message listeners handle asynchronous message receipt.

Listeners implement the `MessageListener` interface by implementing the `onMessage(Message)` method.

```
Listener myListener = new Listener();
consumer.setMessageListener( myListener );
```

Once you set up the listener and call `Connection.start()`, message delivery begins.

The `onMessage()` method should handle all exceptions.

In an EJB module, we use message-driven beans as message listeners.

A message consumer can implement Selective Consumer using JMS message selectors to filter the messages it receives.

We set the `messageSelector` property to string based on SQL syntax such as `NewsType = 'Sports' OR NewsType = 'Opinion'`.

The messaging system will then only deliver messages matching the specified criteria.

Messages in JMS have a header, a set of properties, and a body.

Message header field values can be queried and set:

- `JMSDestination` is set through `send` or `publish`
- `JMSDeliveryMode` is set through `send` or `publish`
- `JMSExpiration` is set through `send` or `publish`
- `JMSPriority` is set through `send` or `publish`
- `JMSMessageID` is set through `send` or `publish`
- `JMSTimestamp` is set through `send` or `publish`
- `JMSCorrelationID` is set by the client
- `JMSReplyTo` is set by the client
- `JMSType` is set by the client
- `JMSRedelivered` is set by the JMS provider

Message properties include message selectors and arbitrary user-defined properties.
JMS message bodies come in five types:

- `TextMessage`: a literal string or XML.
- `MapMessage`: key-value pairs similar to map. E.g. `getInt("numberOfItems")`.
- `BytesMessage`: a stream of raw bytes.
- `ObjectMessage`: a single serializable object.
- `StreamMessage`: a stream of Java primitives like `int`.
- `Message`: an empty body.

Creating and reading messages depends on the body type.

```
TextMessage message = session.createTextMessage();
message.setText(msg_text);     // msg_text is a String
producer.send(message);
```

```
Message m = consumer.receive();
if (m instanceof TextMessage) {
    TextMessage message = (TextMessage) m;
    System.out.println("Reading message: " + message.getText());
} else {
    // Handle error
}
```

It is possible to inspect messages in a `Queue` (but not a `Topic`) without consuming them using a queue browser.

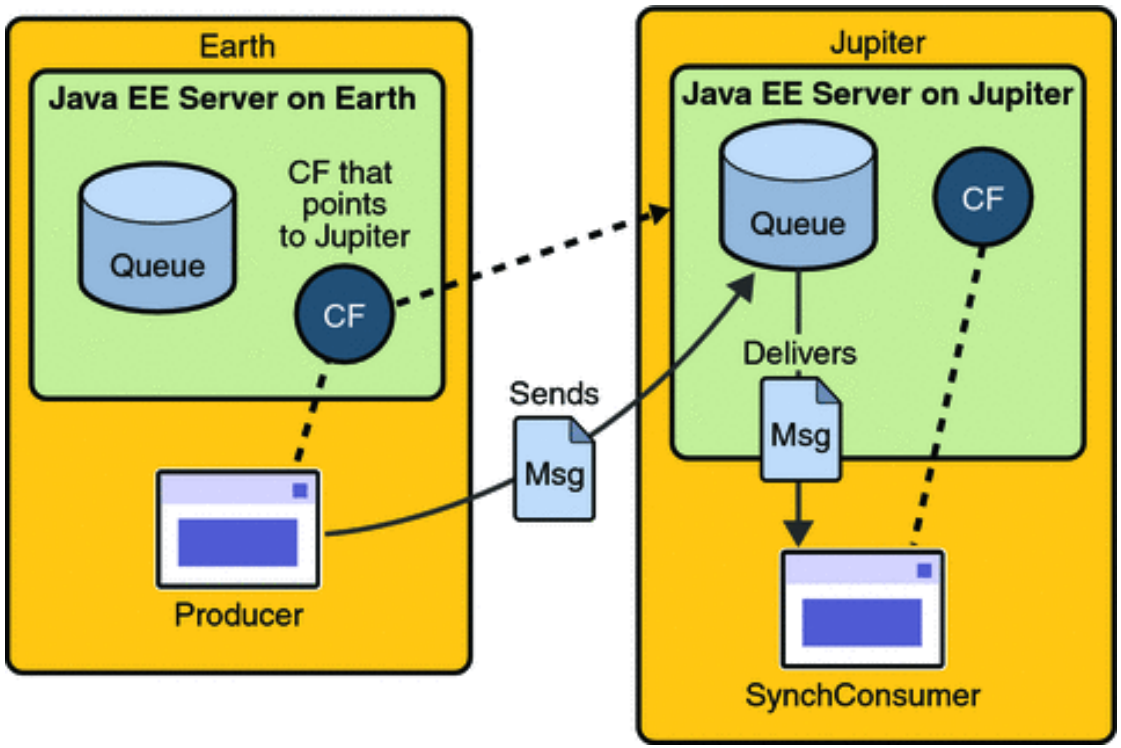
See *The Java EE 5 Tutorial* for examples.

JMS defines several exceptions for various error conditions, all subclasses of `JMSException`.

Java Message Service

Bridges between systems

If two systems are both running the Glassfish application server, they can communicate using remote connection factories:



Jendrock et al. (2006), Fig. 31-6

```
asadmin
create-jms-resource
--host <host> --restype
javax.jms.ConnectionFactory
<factoryname>
```

Note that this feature is vendor-specific.

JMS client applications can create local transactions which group a sequence of message sends and receives into an atomic unit of work that can be committed or rolled back.

Rollback for a local transaction means

- All produced messages are destroyed
- All consumed messages are recovered and redelivered

Rollback thus resets the message channels to the state they were in before the transaction began.

Do not attempt to receive a message such as a reply to a request that depends on a previous send in the same transaction!

EJB components that are JMS clients use distributed transactions, not local transactions.

Now, returning to the EJB container, session beans can synchronously receive messages, but they shouldn't.

- Session bean business methods need to be invoked by some EJB client.
- If a session bean blocks, its client also block or times out.

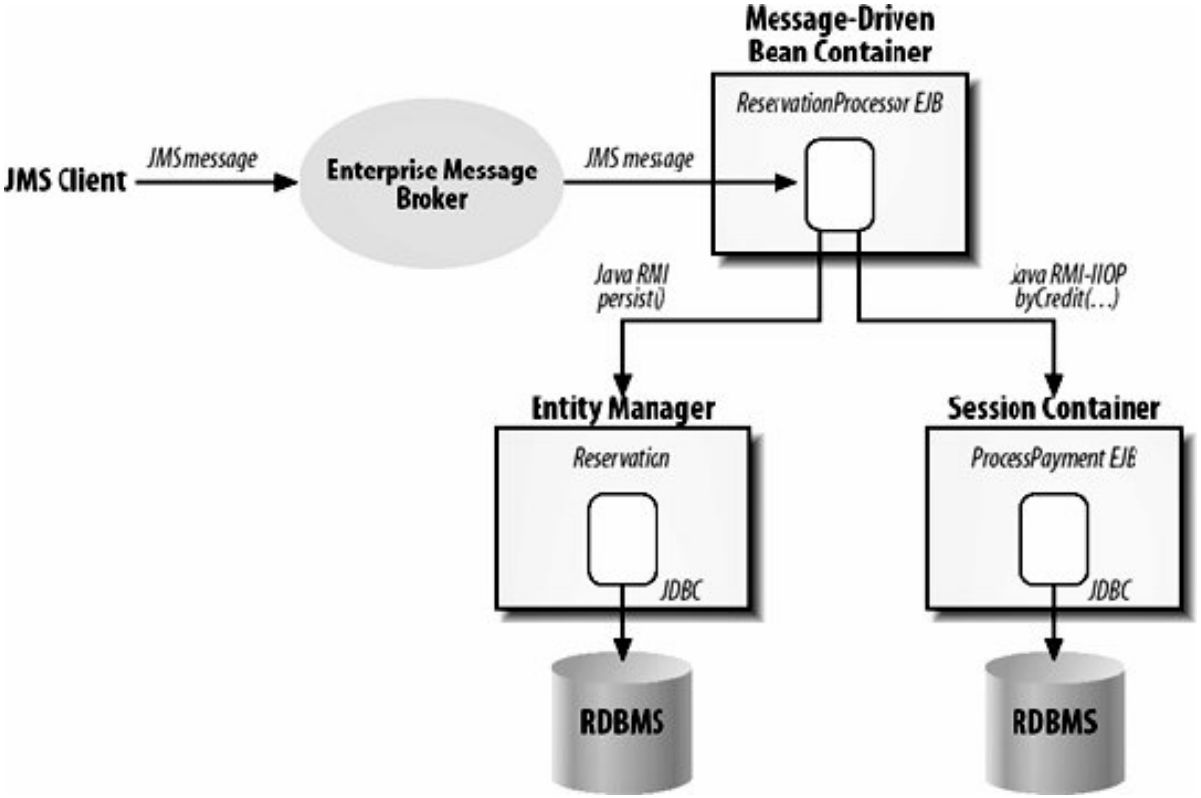
Message-driven beans are stateless, transaction aware components for asynchronous processing of JMS messages.

The container manages concurrency, allowing hundreds of JMS messages to be processed concurrently.

MDBs do not have business interfaces, just an `onMessage()` message handler.

MDBs can receive messages from systems besides JMS (e.g. an email server) through JCA.

As an example, let's look at a JMS version of the Titan cruise reservation system:



Burke and Monson-Haefel (2006), Fig. 12-3.

Here are the annotations for the bean class. We see examples of Selective Consumer and Format Indicator:

```
package com.titan.reservationprocessor;

@MessageDriven(activationConfig={
          @ActivationConfigProperty(
              propertyName="destinationType",
              propertyValue="javax.jms.Queue"),
          @ActivationConfigProperty(
              propertyName="messageSelector",
              propertyValue="MessageFormat = 'Version 3.4'"),
          @ActivationConfigProperty(
              propertyName="acknowledgeMode",
                propertyValue="Auto-acknowledge")})
```

Now the bean class declaration and its injected resources:

```
public class ReservationProcessorBean implements javax.jms.MessageListener {
  @PersistenceContext(unitName="titanDB")
  private EntityManager em;
  @EJB
  private ProcessPaymentLocal process;
  @Resource(mappedName="ConnectionFactory")
  private ConnectionFactory connectionFactory;
  public void onMessage(Message message) {
    ...
  }
  public void deliverTicket(MapMessage reservationMsg, TicketDO ticket) {
    ...
  }
}
```

Now the details of the message handler onMessage() (extracting details of the message and fetching needed data from the database):

```
public void onMessage(Message message) {
  try {
    MapMessage reservationMsg = (MapMessage)message;
    int customerPk = reservationMsg.getInt("CustomerID");
    int cruisePk = reservationMsg.getInt("CruiseID");
    int cabinPk = reservationMsg.getInt("CabinID");
    double price = reservationMsg.getDouble("Price");
    // get the credit card
    Date expirationDate =
       new Date(reservationMsg.getLong("CreditCardExpDate"));
    String cardNumber = reservationMsg.getString("CreditCardNum");
    String cardType = reservationMsg.getString("CreditCardType");
    CreditCardDO card = new CreditCardDO(cardNumber,
       expirationDate, cardType);
    Customer customer = em.find(Customer.class, customerPk);
    Cruise cruise = em.find(Cruise.class, cruisePk);
    Cabin cabin = em.find(Cabin.class, cabinPk);
    ...
```

Finally, we create and persist the reservation then complete the transaction:

```
      ...
      Reservation reservation = new Reservation(
                customer, cruise, cabin, price, new Date( ));
      em.persist(reservation);
      process.byCredit(customer, card, price);
      TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
      deliverTicket(reservationMsg, ticket);
    } catch(Exception e) {
      throw new EJBException(e);
    }
}
```

EJB applications that use JMS combine message sends and receives with ongoing distributed transactions.

MDBs can declare transaction attributes `NotSupported` or `Required` (the default) but the other attributes involving transaction propagation are not available.

When an EJB creates a JMS session, the transaction and autoacknowledge parameters are ignored since the container is in charge of managing transactions and acknowledgment.

As with local transactions, you cannot send a message and receive a reply to that message within the same transaction.

If `onMessage()` throws a `RuntimeException`, as with other EJBs, the container rolls back the transaction including redelivering the unacknowledged message.

Outline

Conclusion

Now you have an idea of how the major pieces of EJB work.

Java EE supports most of the enterprise architectural patterns we've discussed this semester.

The philosophy is to build the common requirements of large enterprise applications into the middleware layer.

The developer just needs to understand the framework and plug in her code.

The system is large and complex but promises reliability and maintainability when we're building large systems.

For small and medium sized projects, consider using lighter weight platforms to increase your agility. Frameworks based on Ruby and Python are excellent alternatives.