Taller de Sistemas de Información 2

Clase 4

Lógica de negocio con EJBs

Lógica de negocio con Session Beans

- Desde este punto de vista, los EJB de sesión son los componentes más importantes de la tecnología EJB
- Su propósito es modelar la lógica de negocio de la aplicación
- Esto es, los procesos de negocio de la solución

Entendiendo un Session Bean

- La teoría detrás de un session bean, se centra en la idea de que un cliente emite requests para completar un proceso de negocio
- Cada request, puede ser completado en una **session**
- **Una sesión, es una conexión entre cliente y servidor, que dura por un periodo finito de tiempo**

Entendiendo un Session Bean

- Son los únicos EJBs que pueden ser invocados directamente desde el cliente
- Un cliente puede ser
  - Un componente web (servlet, JSP, JSF)
  - Una aplicación de consola
  - Una aplicación de escritorio (Swing)
  - Un cliente .NET (vía web service)

¿Por qué utilizar Session Beans?

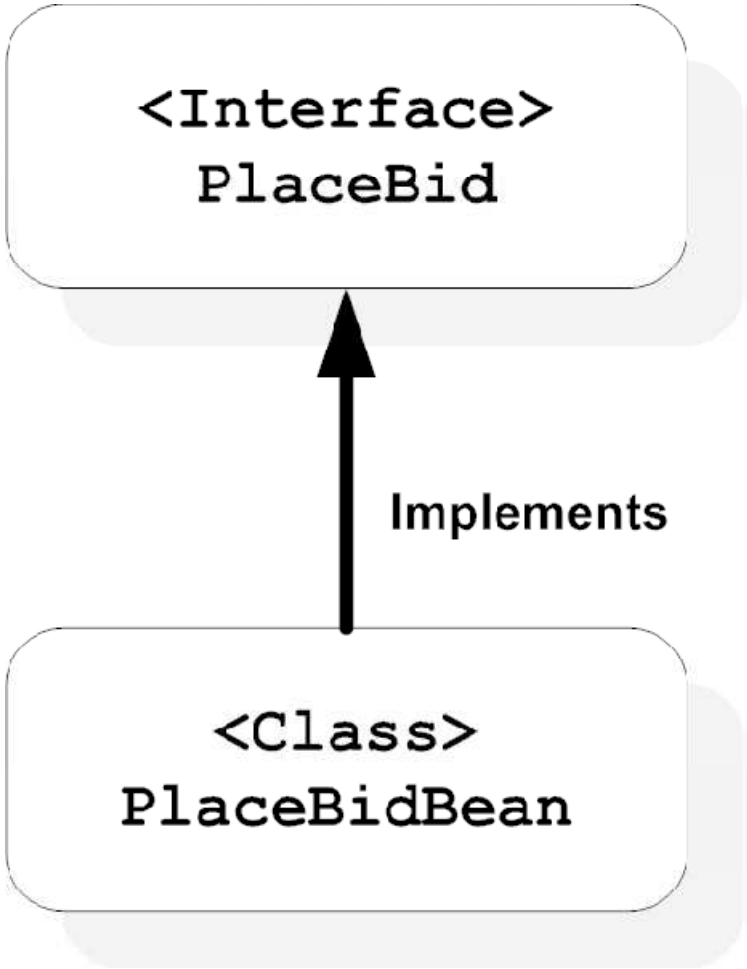
- Concurrency & thread safety
- Remoting y web services
- Transaccionalidad
- Seguridad
- Intercepción y AOP

Anatomía de un Session Bean

- Todo Session Bean tiene dos partes diferentes en su implementación
  - Una o más interfaces
  - Una clase de implementación de dichas interfaces

Anatomía de un Session Bean

- La implementación del bean PlaceBid consiste en la interfaz PlaceBid y la clase PlaceBidBean



Interfaz de negocio

- Una interfaz a través de la cual el cliente habla con el session bean, se denomina interfaz de negocio
- Esta define los métodos accesibles desde el cliente, a través de algún mecanismo de acceso

```
@Local
public interface PlaceBid {
   Bid addBid(Bid bid);
}
```

Clase de implementación

- Toda interfaz de negocio que el bean pretenda soportar, debe ser implementada por la clase de implementación, a través de la palabra clave implements

```
@Stateless
public class PlaceBidBean implements PlaceBid {
public PlaceBidBean() {}
   public Bid addBid(Bid bid) {
      ...
      return save(bid);
   }
}
```

Reglas de programación

- Debe existir una interfaz de negocio
- La clase de implementación debe ser concreta
- Debe existir un constructor sin parámetros
- La clase de implementación, puede extender otra clase
- Los métodos de la clase de implementación no deben comenzar con el prefijo “ejb”

Reglas de programación

- Si estamos exponiendo un método a través de una interfaz remota, entonces además debemos asegurarnos que:
  - Los argumentos del método son tipos primitivos o implementan `java.io.Serializable`
  - El tipo de retorno del método es un tipo primitivo o implementa `java.io.Serializable`

Lifecycle events

- El ciclo de vida de un session bean puede categorizarse en múltiples etapas o eventos
  - **Creación**
  - **Destrucción**
  - **Pasivado**
  - **Activado**

Lifecycle events

- El bean puede necesitar saber cuándo una transición entre sus estados del ciclo de vida ocurre
  - Por ejemplo, cuando se crea un bean o antes de que se destruya
  - Puede servir para inicializar información o liberar recursos
- En este punto es en donde entran los callbacks

Lifecycle callbacks

- Los lifecycle callbacks son métodos que el container llama, para notificar al bean que ocurrió una transición en el ciclo de vida
- Los lifecycle callbacks no son expuestos por la interfaz de negocio
- Cuando el evento ocurre, el container llama al método de callback correspondiente

Lifecycle callbacks

- Los métodos de callback se encuentran marcados por anotaciones como ser @PostContruct y @PreDestroy
  - PostConstruct es invocada luego de que la instancia del bean es creada, y sus dependencias inyectadas
  - PreDestroy es invocada justo antes de que la instancia sea destruida. Es útil para la limpieza de recursos

Lifecycle callbacks annotations

- Callback Annotation
  - `javax.annotation.PostConstruct`
- Tipos de EJB
  - stateless, stateful, MDB
- Es invocada luego de que la instancia del bean es creada y las dependencias inyectadas.
- Permite crear obtener recursos utilizados por el bean.

Lifecycle callbacks annotations

- Callback Annotation
  - `javax.annotation.PreDestroy`
- Tipos de EJB
  - stateless, stateful, MDB
- Es invocada antes de que la instancia del bean sea destruida.
- Se usa para reciclar recursos referenciados por el bean.

Lifecycle callbacks annotations

- Callback Annotation
  - `javax.ejb.PrePassivate`
- Tipo de EJB
  - stateful
- Este método es invocado justo antes de que el bean sea pasivado.
- Sirve para preparar el mismo para esta acción
  - por ejemplo, cerrando una conexión que el bean puede tener abierta

Lifecycle callback annotations

- Callback Annotation
  - `javax.ejb.PostActivate`
- Tipo de EJB
  - stateful
- Este método es invocado justo después de que una instancia del bean ha sido restaurada (activada).
- Sirve para restaurar estados no persistibles del bean.

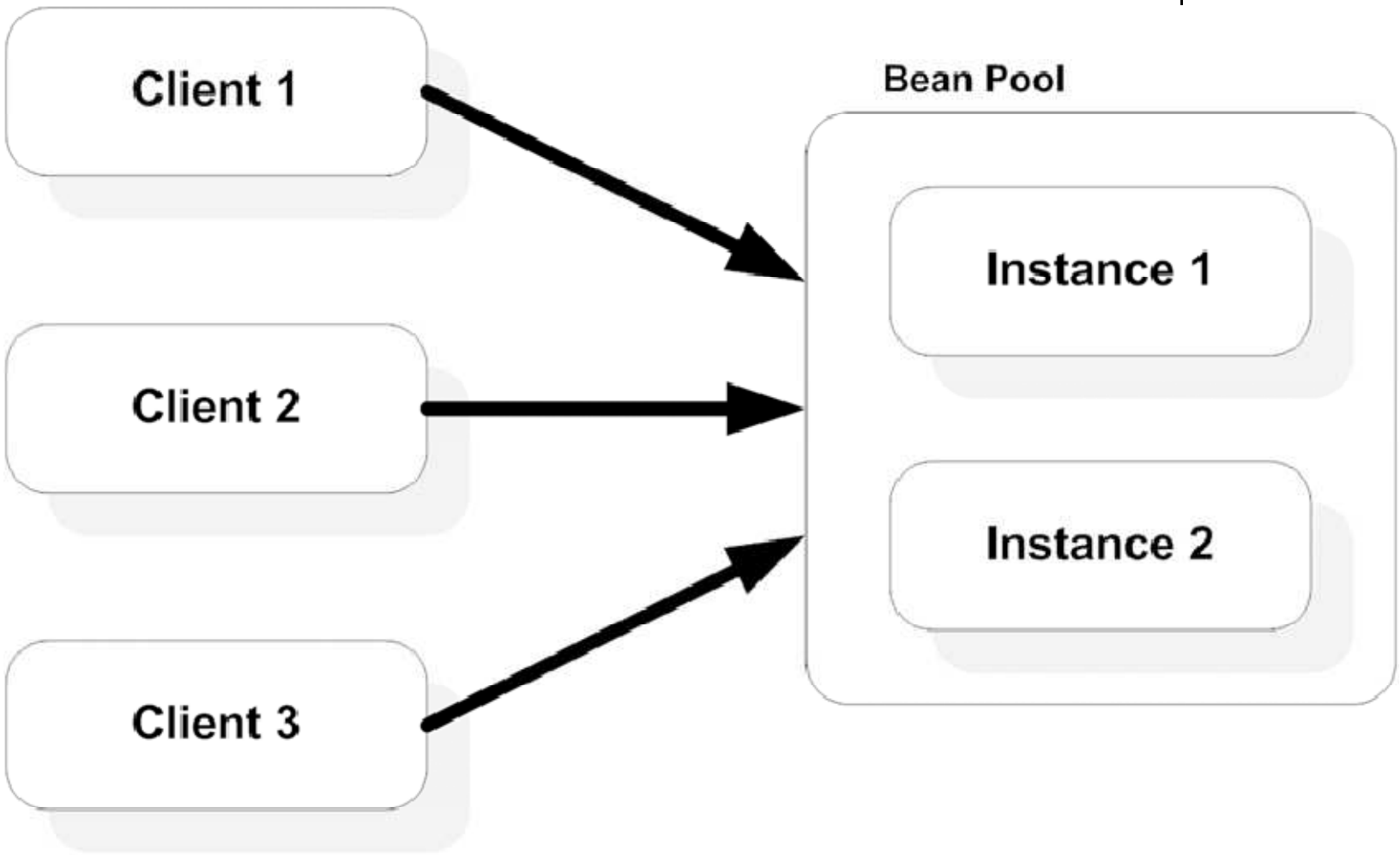
Stateless session beans

- Este tipo de bean modela tareas que no requieren un estado conversacional entre llamadas
- Esto significa que la operativa de negocio puede ser resuelta en una sola llamada
  - Por ejemplo, el método `addBid`
- Esto no significa que el bean stateless solo deba contener un método

Stateless session beans

- Por lejos, son el tipo de componente más versátil y popular de la plataforma Java EE
- También son los mas performantes y eficientes
  - Es posible hacer Pooling de instancias de stateless session bean

Stateless session beans



Stateless session beans

- Este pool de instancias del bean, permite que las instancias sean compartidas por diferentes clientes
  - Siempre de a uno por vez
- Cuando el cliente invoca un método en un bean, el container crea uno nuevo, o retira un bean libre del pool
- Luego del uso, el container retorna la instancia a dicho pool

BidManagerBean

```
@Stateless(name="BidManager")
public class BidManagerBean implements BidManager {

  @Resource(name="jdbc/ActionBazaarDS")
  private DataSource dataSource;
  private Connection connection;
  ...

  public BidManagerBean() {...}
  private Long getBidId() {...}
  public void cancelBid(Bid bid) {...}
  public List<Bid> getBids(Item item) {...}
```

BidManagerBean

```
@PostConstruct
public void initialize() {
    try {
        connection = dataSource.getConnection();
    } catch (SQLException e) { ... }
}

@PreDestroy
public void cleanup() {
    try {
        connection.close();
    } catch (SQLException e) { ... }
}
```

BidManagerBean

```
public void addBid(Bid bid){
 try {
   Long bidId = getBidId();
   Statement stmt = connection.createStatement();
   stmt.execute(
     "INSERT INTO BIDS (BID_ID, BID_AMOUNT, "
     + "BID_BIDDER_ID, BID_ITEM_ID) "
     + "VALUES (bidId + "," + bid.getAmount()
     + "," + bid.getBidder().getUserId()
     + "," + bid.getItem().getItemId()+ ")");
 } catch (Exception e) {
    e.printStackTrace();
 }
}
```

BidManagerBean

```
@Remote
public interface BidManager {
    void addBid(Bid bid);
    void cancelBid(Bid bid);
    List<Bid> getBids(Item item);
}
```

Interfaces de negocio para el bean

- Las aplicaciones cliente pueden invocar al bean stateless en tres formas diferentes
  - invocación local en la misma JVM
  - invocación remota a través de RMI
  - invocación vía web service a través de SOAP
- Cada una de estas corresponde a vías diferentes de acceso al bean

Local interface

- Una interfaz local esta diseñada para clientes de un Stateless Session Bean, localizados en la misma instancia de la JVM
- Se designa este tipo de interfaces colocando la anotación @Local en la interfaz

Remote interface

- Los clientes que se encuentran fuera del container, para ser más exacto, fuera de la instancia de la JVM, deben acceder remotamente
- Si el cliente es Java, puede utilizar RMI (lo más común)
- Para designar una interfaz de este tipo, usamos la anotación @Remote

Remote interface

- Este tipo de interfaces tienen un requerimiento especial
  - Todos los parámetros y tipos de retorno de los métodos en la interfaz, deben ser Serializables (implementar `java.io.Serializable`)
- Esto es debido a que los objetos serializables son los únicos que pueden salir de la JVM

Web service endpoint interface

- También conocida como SEI
  - Service Endpoint Interface

```
@WebService
public interface BidManagerWS {
   void addBid(Bid bid);
   List<Bid> getBids(Item item);
}
```

Múltiple interfaces de negocio

- No podemos marcar la misma interfaz, con mas de un tipo de anotación de acceso
- Por ejemplo, BidManager no puede ser marcada con @Local y @Remote
- En este caso, podemos:
  - Hacer 2 interfaces, con lo cual tendríamos código repetido
  - O utilizar herencia de interfaces

Múltiple interfaces de negocio

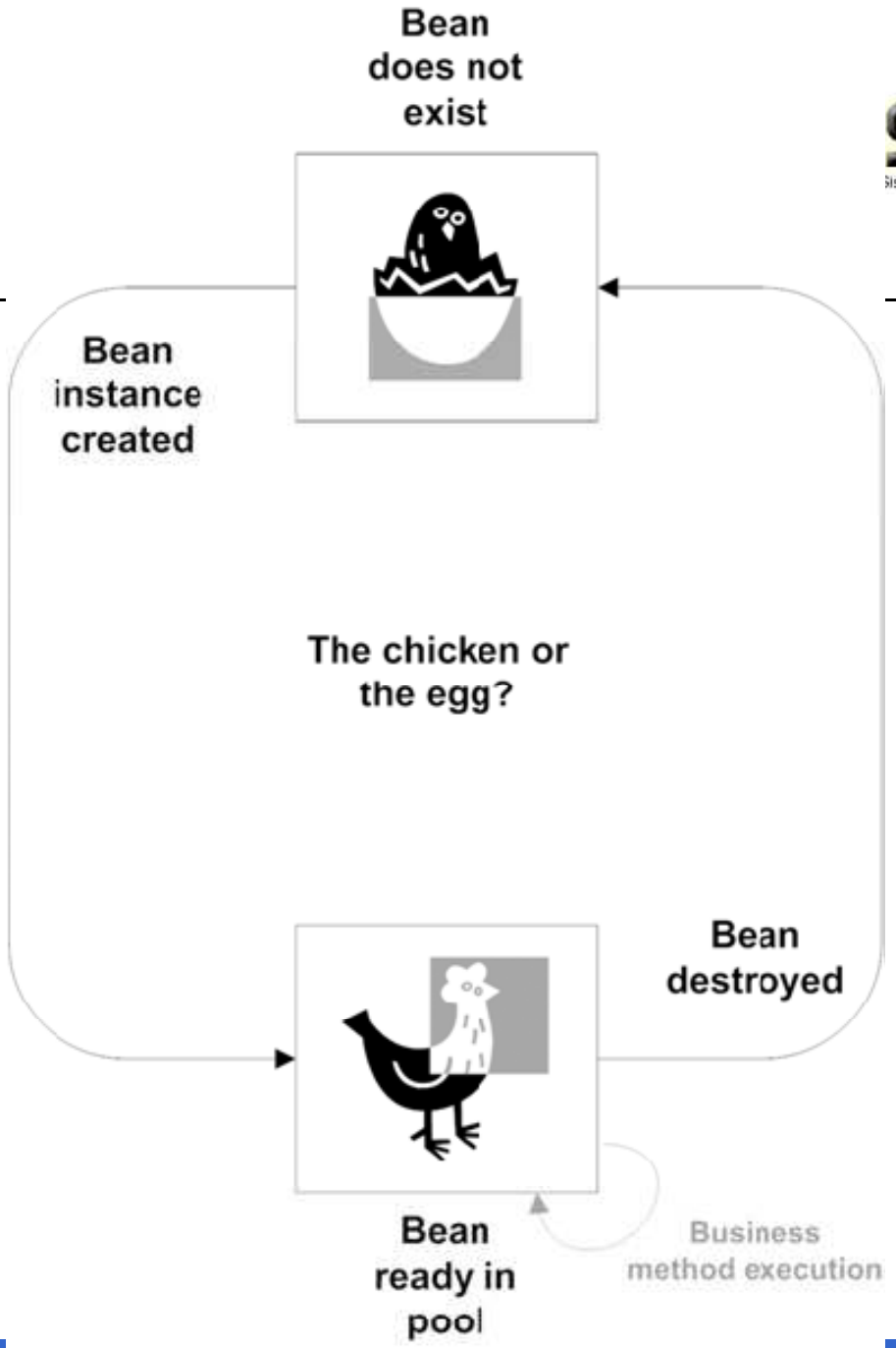

```
public interface BidManager{
   void addBid(Bid bid);
   List<Bid> getBids(Item item);
}
@Local
public interface BidManagerLocal extends BidManager {
   void cancelBid(Bid bid);
}
@Remote
public interface BidManagerRemote extends BidManagerLocal
{}
@WebService
public interface BidManagerWS extends BidManager {}
```

Múltiple interfaces de negocio

- También podemos aplicar @Local, @Remote, o @WebService directamente en la clase, sin necesidad de implementar la interfaz
- Esto es NO recomendable

```
@Remote(BidManager.class)
@Stateless
public class BidManagerBean {
    ...
}
```

Usando los bean lifecycle callbacks



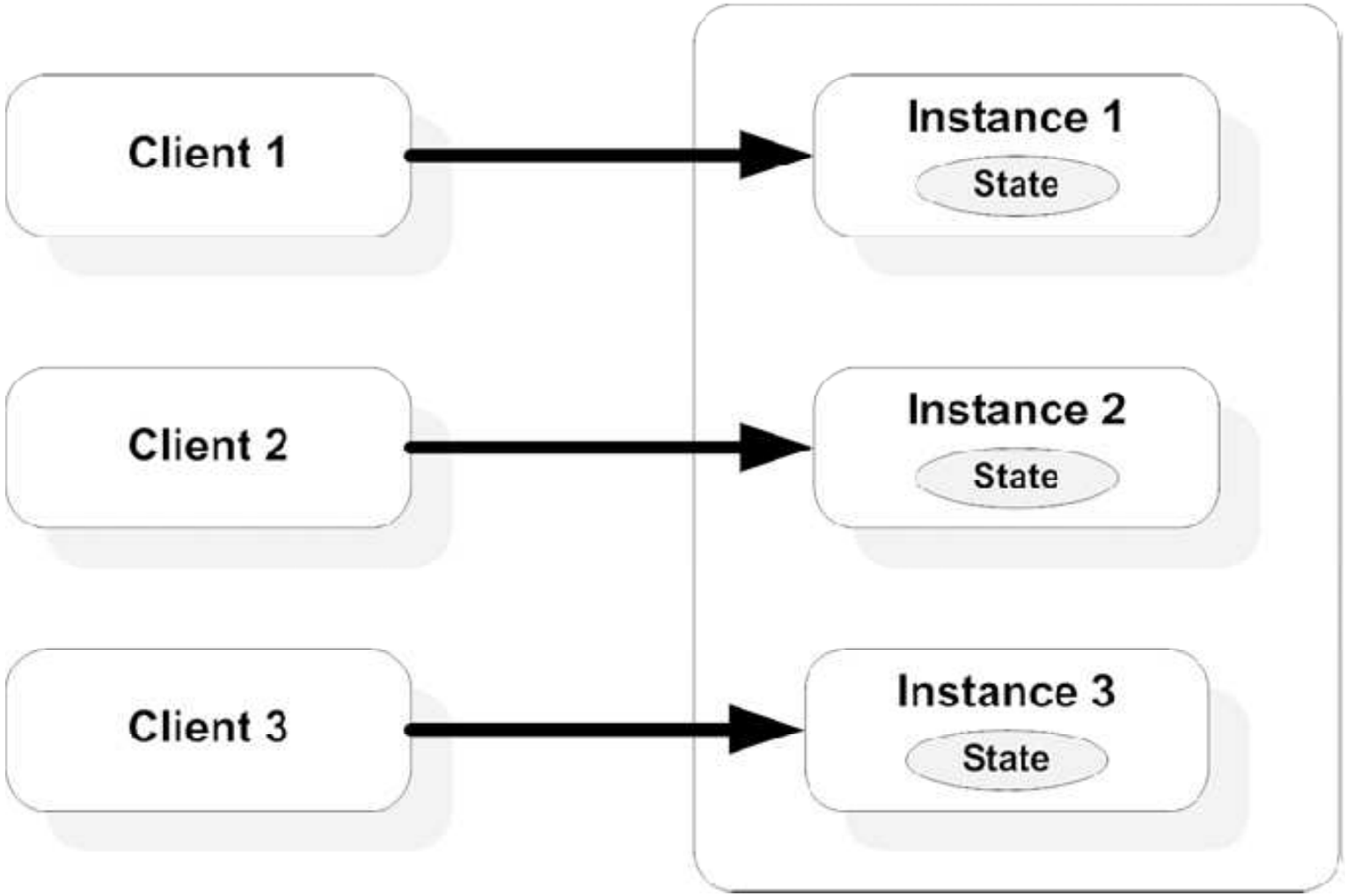
Usando los bean lifecycle callbacks

1. Se crea la instancia del bean, usando el constructor por defecto
2. Se inyectan los recursos, (ej. conexiones a la base de datos)
3. Se coloca la instancia en el managed pool de instancias
4. Cuando ocurre una invocación, se saca una instancia del pool. Si no hay, el container puede incrementar el tamaño del pool, creando nuevas instancias
5. Ejecuta los métodos de negocio solicitados, a través de la interfaz de negocio
6. Cuando el método de negocio termina, el bean se coloca en el pool, en el estado “method-ready”
7. Si es necesario, se retiran (destruyen) instancias del bean del pool

Stateful session beans

- Este tipo de beans garantizan mantener el estado conversacional con el cliente
  - procesos de varios pasos
  - workflows
- En realidad, la única diferencia entre ambos tipos de beans, es como el container maneja el ciclo de vida
- Existe un mapping uno a uno entre una instancia del bean, y el cliente que lo utiliza
  - Esto es mantenido por el container

Stateful session beans



Stateful session beans

- Esta relación uno a uno, tiene un precio
- Las instancias del bean no pueden ser devueltas al pool para ser reutilizadas por otro cliente, si la sesión del primero esta activa
- Esta instancia del bean debe esperar en memoria, a que el cliente finalice su interacción de varios pasos

Stateful session beans

- Por este motivo, el consumo de memoria de una aplicación que utiliza múltiples clientes concurrentes con este tipo de beans, es mucho más alto que con stateless session beans

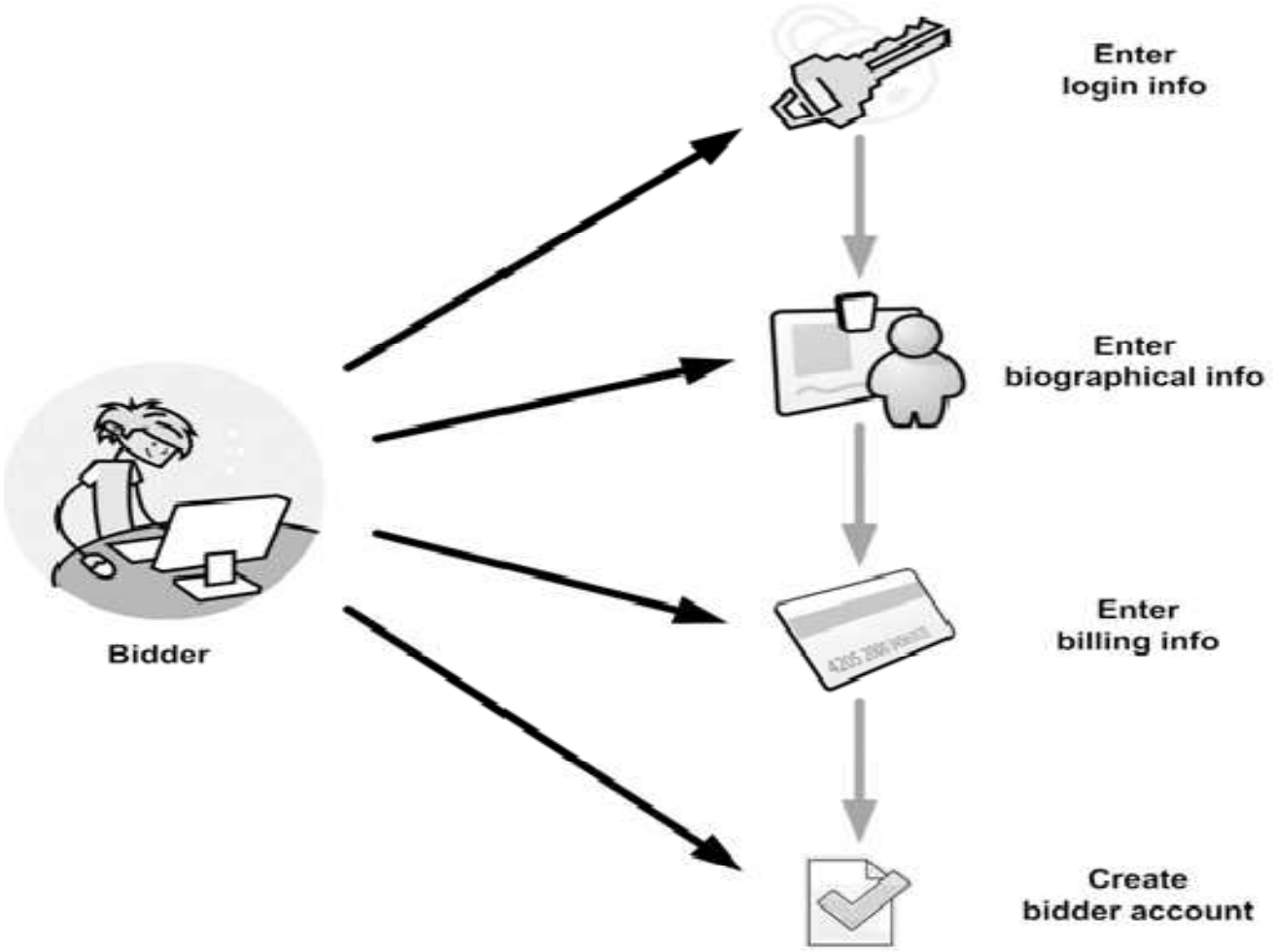
Más reglas de programación

- Las variables de instancia de un stateful session bean, deben ser
  - valores primitivos Java
  - objetos que implementan `java.io.Serializable`
- Debemos indicar al menos un método del bean, con la anotación @Remove
  - Le avisa qué método(s) termina(n) el ciclo de vida del bean
  - Debido a que no pueden ser reutilizados, existe un peligro si acumulamos muchos de estos

Más reglas de programación

- Como habíamos mencionado, aparte de los @PostConstruct y @PreDistroy ahora tenemos disponibles
  - @PrePassivate
  - @PostActivate

BidderAccountCreatorBean



BidderAccountCreatorBean


```
@Stateful(name="BidderAccountCreator")
public class BidderAccountCreatorBean
implements BidderAccountCreator {

    @Resource(name="jdbc/ActionBazaarDS")
    private DataSource dataSource;
    private Connection connection;

    private LoginInfo loginInfo;
    private BiographicalInfo biographicalInfo;
    private BillingInfo billingInfo;
```

BidderAccountCreatorBean


```
public BidderAccountCreatorBean () {}

@PostConstruct
@PostActivate
public void openConnection() {
  try {
    connection = dataSource.getConnection();
  } catch (SQLException sqle) {
    sqle.printStackTrace();
  }
}
```

BidderAccountCreatorBean

```
public void addLoginInfo(LoginInfo loginInfo) {
   this.loginInfo = loginInfo;
}

public void addBiographicalInfo(
         BiographicalInfo biographicalInfo) {
   this.biographicalInfo = biographicalInfo;
}

public void addBillingInfo(
         BillingInfo billingInfo) {
   this.billingInfo = billingInfo;
}
```

BidderAccountCreatorBean

```
@PrePassivate
@PreDestroy
public void cleanup() {
   try {
      connection.close();
      connection = null;
   } catch (SQLException sqle) {
      sqle.printStackTrace();
   }
}
```

BidderAccountCreatorBean

```
@Remove
public void createAccount() {
   try {
      Statement stmt = connection.createStatement();
      statement.execute("INSERT INTO BIDDERS(“ ...);
      statement.close();
   } catch (SQLException e) { ... }
}
      @Remove
      public void cancelAccountCreation() {
         loginInfo = null;
         biographicalInfo = null;
         billingInfo = null;
      }
```

BidderAccountCreator


```
@Remote
public interface BidderAccountCreator
                    implements Remote {
   void addLoginInfo(LoginInfo loginInfo);
   void addBiographicalInfo(
        BiographicalInfo biographicalInfo);
   void addBillingInfo(
        BillingInfo billingInfo);
   void cancelAccountCreation();
   void createAccount();
}
```

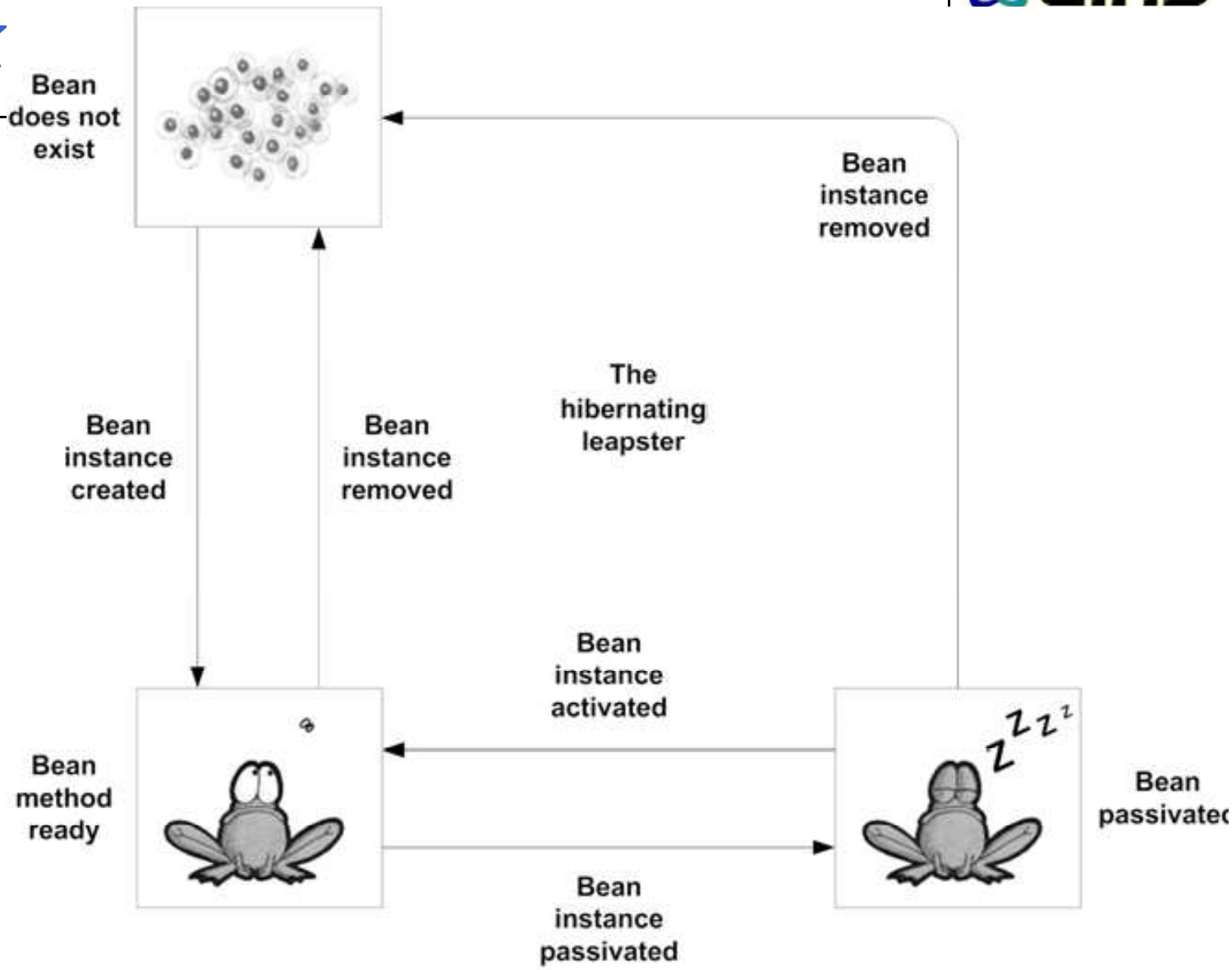
Algunos detalles importantes

- Son muy similares a los stateless
- El datasource es inyectado utilizando un @Resource como en el ejemplo de los stateless
  - Un Datasource es una especie de pool de conexiones mantenido por el container
  - La conexión se abre en @PostConstruct y @PostActivate
  - Se cierra en @PreDestroy y @PrePassivate
- El workflow se finaliza, cuando el cliente invoca alguno de los métodos @Remove

Business interfaces para stateful beans

- Son muy similares a las de un stateless, soportan invocación con @Local y @Remote
- Sin embargo, no soportan @WebService
  - Esto es debido a que los servicios SOAP son inherentemente stateless
- Además, siempre debemos incluir al menos uno de los métodos anotados con @Remove en la interfaz

Stateful bean lifecycle callback



Stateful bean lifecycle callbacks

1. Se crea la instancia del bean, usando el constructor por defecto
2. Se inyectan los recursos, como ser conexiones a la base de datos
3. Ejecuta los métodos de negocio solicitados, a través de la interfaz de negocio
4. Espera por y ejecuta sucesivas llamadas a los métodos del negocio

Stateful bean lifecycle callbacks

5. Si el cliente se queda ocioso por una cierta cantidad de tiempo, el container realiza el pasivado del bean. Esto significa que el bean es sacado de memoria, serializado y colocado en almacenamiento secundario
6. Si el cliente invoca un bean pasivado, este es activado
7. Si el cliente no invoca un bean pasivado por una cierta cantidad de tiempo, este es destruido
8. Si el cliente solicita una remoción de un bean pasivado, este es activado y luego es removido

Passivation y activation

- “Passivation” es el proceso que salva el estado del bean en memoria, en un almacenamiento secundario
  - Este proceso implica serialización del contenido del bean
- “Activation” es el proceso inverso, invocado cuando el bean es necesario nuevamente
  - Este proceso implica de serialización del contenido almacenado en disco

@PrePassivate

- La idea de esta anotación, es la de prepararnos para el pasivado
- Esta tarea implica por lo general
  - liberar recursos pesados que no son serializados
  - limpiar datos innecesarios para ocupar menos espacio
- Una conexión a la base de datos es un buen ejemplo de esto

@PostActivate

- La idea de esta anotación, es la de recuperarnos de un pasivado previo, de forma de estar prontos para ejecutar nuevamente
- Esta tarea implica por lo general recuperar recursos pesados que no debieron o no pudieron ser serializados
- Una conexión a la base de datos es un buen ejemplo de esto

Destruyendo un stateful session bean

- En el ejemplo anterior, tenemos dos métodos, cancelAccountCreation y createAccount que están marcados con @Remove
- Tienen un rol muy importante manteniendo la performance del servidor de aplicaciones
- Invocar un método con @Remove, significa que el cliente quiere terminar la sesión

Destruyendo un stateful session bean

- La invocación de estos métodos, desencadena la destrucción del bean
- No realizarlo, generaría una acumulación imposible de mantener de beans en la memoria del application server
  - "Clásico problema de fuga de memoria"
  - En general se terminarían eliminando por timeout.

Diferencias entre los tipos de beans

Features	Stateless	Stateful
Conversational state	No	Yes
Pooling	Yes	No
Performance problems	Unlikely	Possible
Lifecycle events	`PostConstruct`, `PreDestroy`	`PostConstruct`, `PreDestroy`, `PrePassivate`, `PostActivate`
Timer	Yes	No
`SessionSynchronization` for transactions	No	Yes
Web services	Yes	No
Extended `PersistenceContext`	No	Yes

Session bean clients

- Casi cualquier componente puede ser un cliente de un session bean
  - POJOs, Servlets, JSPs, EJBs, etc.
- Desde el punto de vista del cliente, el acceso al EJB (stateless o stateful) no tiene mucha diferencia

Session bean clients

1. El cliente obtiene una referencia al bean, directa vía JNDI o indirecta vía @EJB
2. Toda invocación es realizada a través de una interfaz de negocio, marcada con la forma de acceso apropiada
3. El cliente invoca lo que quiere en el bean, para cumplir con el negocio
4. En el caso de un stateful, la ultima invocación debería ser un método marcado con @Remove

Session bean clients


```
@Stateless
public class GoldBidderManagerBean
implements GoldBidderManager {
    @EJB
    private BidManager bidManager;

    public void addMassBids(List<Bid> bids) {
        for (Bid bid : bids) {
            bidManager.addBid(bid);
        }
    }
}
```

@EJB

- Esta anotación esta diseñada para inyectar referencias a un EJB dentro de código cliente
- Hay que recordar que la inyección es solo posible en ambientes manejados
  - El cliente puede ser cualquier cosa administrada por un servidor de aplicaciones
  - El proveedor puede ofrecer mecanismos para soportar inyectar en POJOs (extensión)

@EJB especificación

```
@Target(...)
@Retention(RUNTIME)
public @interface EJB {
    String name() default "";
    Class beanInterface() default Object.class;
    String beanName() default "";
}
```

@EJB

- Si queremos inyectar especificando el nombre del bean (JNDI), podemos hacerlo así:

```
@EJB(name="BidManagerRemote")
private BidManager bidManager;
```

Injection y stateful session beans

- Podemos inyectar un stateful bean dentro de otro stateful bean si lo necesitamos

```
@Stateful
public class UserAccountRegistrationBean
implements UserAccountRegistration {
   @EJB
   private BidderAccountCreator
                bidderAccountCreator;
...
}
```

Injection y stateful session beans

- Se crea una instancia del bean asociada al cliente de la instancia UserAccountReg…
  - Si el cliente remueve la instancia del UserAccount también se removerá la del BidderAccount..
- Nunca debemos (aunque podemos) inyectar un stateful session bean en un stateless bean, ya que este será compartido por múltiples clientes
  - Lo mismo con cualquier otro tipo de componente stateless

Consideraciones de performance

- Debemos usar session beans stateful, solo si es necesario, ya que necesariamente generan un mayor impacto en performance
- Debemos recordar usarlos apropiadamente, esto es
  - Limitar el tamaño de los objetos dentro del bean
  - Invocar @Remove cuando se finalice el uso del mismo

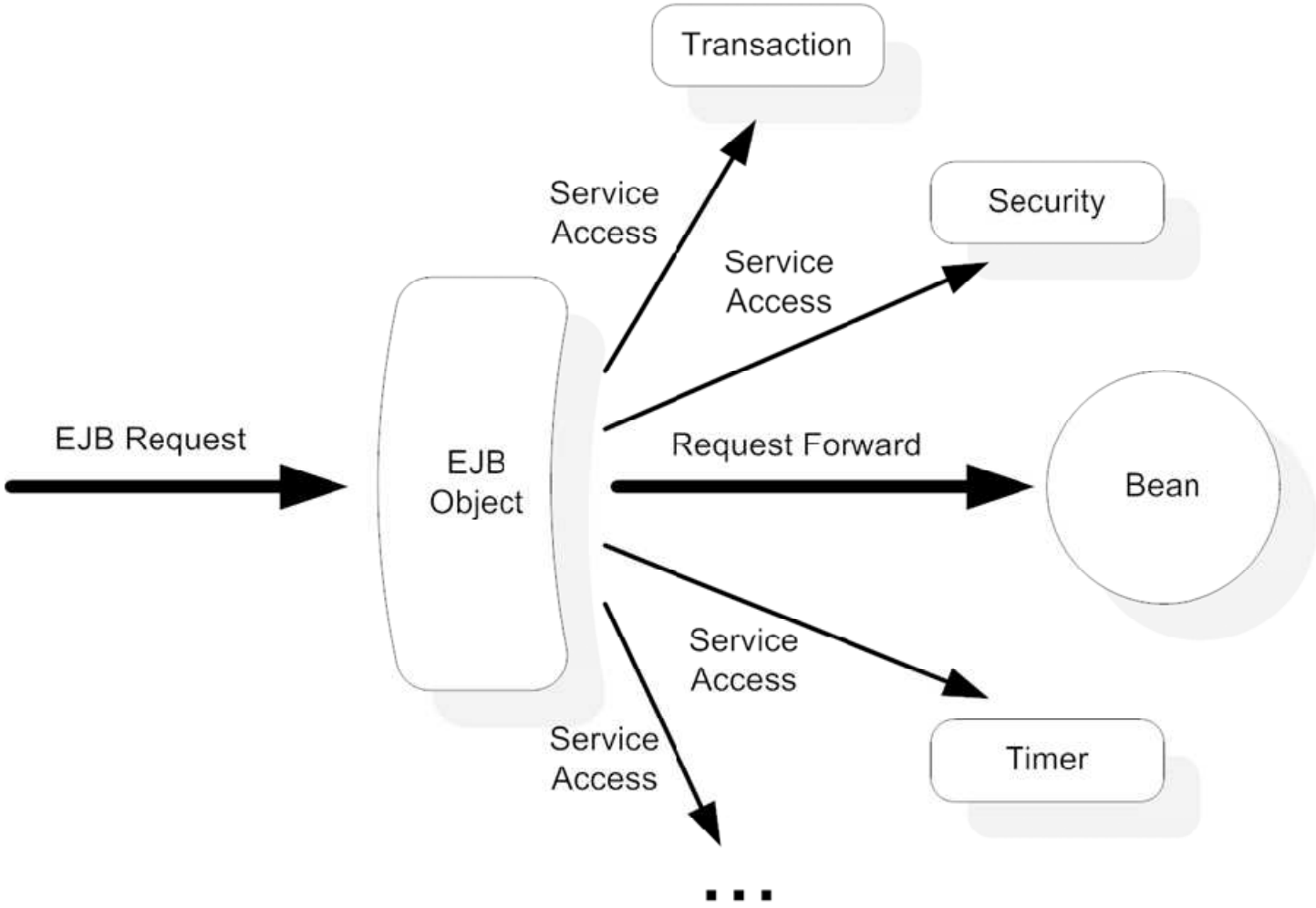
El detrás de escena de los EJB

- EJB se centra en la idea de los “managed objects”
- Un EJB 3 es simplemente un POJO anotado
- Como vimos ya:
  - cuando un cliente invoca un EJB a través de su interfaz, nunca trabaja directamente sobre la instancia

El detrás de escena de los EJB

- Para cada instancia del bean, el container genera un proxy denominado EJBObject
- Este tiene acceso a toda la funcionalidad del container, incluyendo
  - registro JNDI, seguridad, manejo de transacciones, thread pools, manejo de sesiones
- El EJBObject conoce la configuración del bean y los servicios que el POJO supuestamente provee

El detrás de escena de los EJB



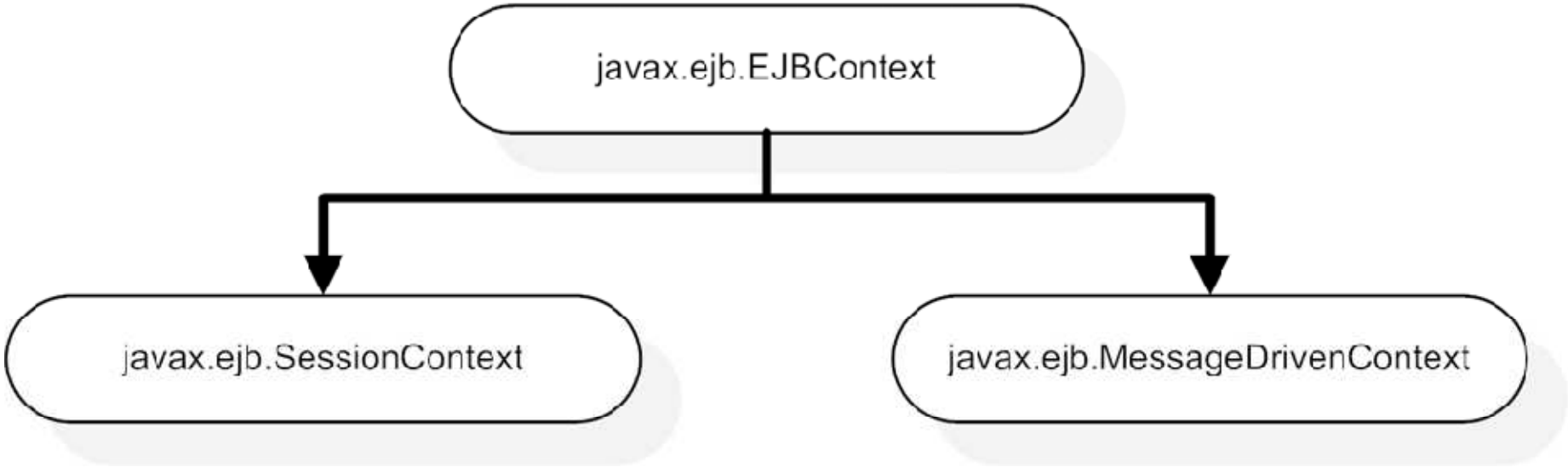
EJB Context

- Idealmente quisiéramos que los EJB no interactuaran directamente con el container
  - En la realidad resulta muy restrictivo
- El contexto del EJB permite acceder el runtime environment del componente
- El acceso a este ambiente, se hace a través de la interfaz `javax.ejb.EJBContext`
- Esta permite acceso programático a servicios como las transacciones y la seguridad

EJB Context

```
public interface EJBContext {
 public Principal getCallerPrincipal();
 public boolean isCallerInRole(String roleName);

 public EJBHome getEJBHome();
 public EJBLocalHome getEJBLocalHome();

 public boolean getRollbackOnly();
 public UserTransaction getUserTransaction();
 public void setRollbackOnly();

 public TimerService getTimerService();

 public Object lookup(String name);
}
```

EJB Context


javax.ejb.EJBContext
javax.ejb.SessionContext
javax.ejb.MessageDrivenContext

Utilizando el EJB Context

- Podemos ganar acceso al EJB Context utilizando inyección de dependencias

```
@Stateless
public class PlaceBidBean implements PlaceBid {
   @Resource
   SessionContext context;

   ...
}
```

Utilizando el EJB Context

- En el caso del MDB, es similar…

```
@MessageDriven
public class OrderBillingMDB {
   @Resource
   MessageDrivenContext context;
...
}
```

Acceso a recursos vía DI o JNDI

- La anotación @Resource es el mecanismo de inyección mas versátil de EJB 3
- En la mayoría de los casos, esta se utiliza para inyectar
  - datasources JDBC
  - conexiones JMS
  - contextos EJB

Acceso a recursos vía DI o JNDI

- Sin embargo, puede utilizarse para inyectar una variedad mucho mayor de recursos
  - parámetros de ambiente, e-mail, timer-service, etc
  - inclusive referencias a EJBs (@EJB debería ser utilizada)
- @Resource también se puede definir a nivel de atributo, método o clase
  - método se usa para usar setters/getters y no atributos
    - permiten setear dependencias desde afuera: Unit Testing
    - ventajas usuales de setters/getters
  - clases permite definir dependencias hacia recursos

Acceso a recursos vía DI o JNDI

- Por ejemplo, el caso del datasource es uno de los mas sencillos

```
@Stateless
public class PlaceBidBean implements PlaceBid {
 ...
 @Resource(name="jdbc/actionBazaarDB")
 private DataSource dataSource;
```

EJBContext.lookup()

```
@EJB(name="ejb/BidderAccountCreator",
     beanInterface = BidderAccountCreator.class)
@Stateless
public class GoldBidderManagerBean
             implements GoldBidderManager {

  @Resource SessionContext sessionContext;
  ...
  BidderAccountCreator accountCreator =
       (BidderAccountCreator)sessionContext.lookup(
                          "ejb/BidderAccountCreator");
  ...
  accountCreator.addLoginInfo(loginInfo);
  ...
  accountCreator.createAccount();
```

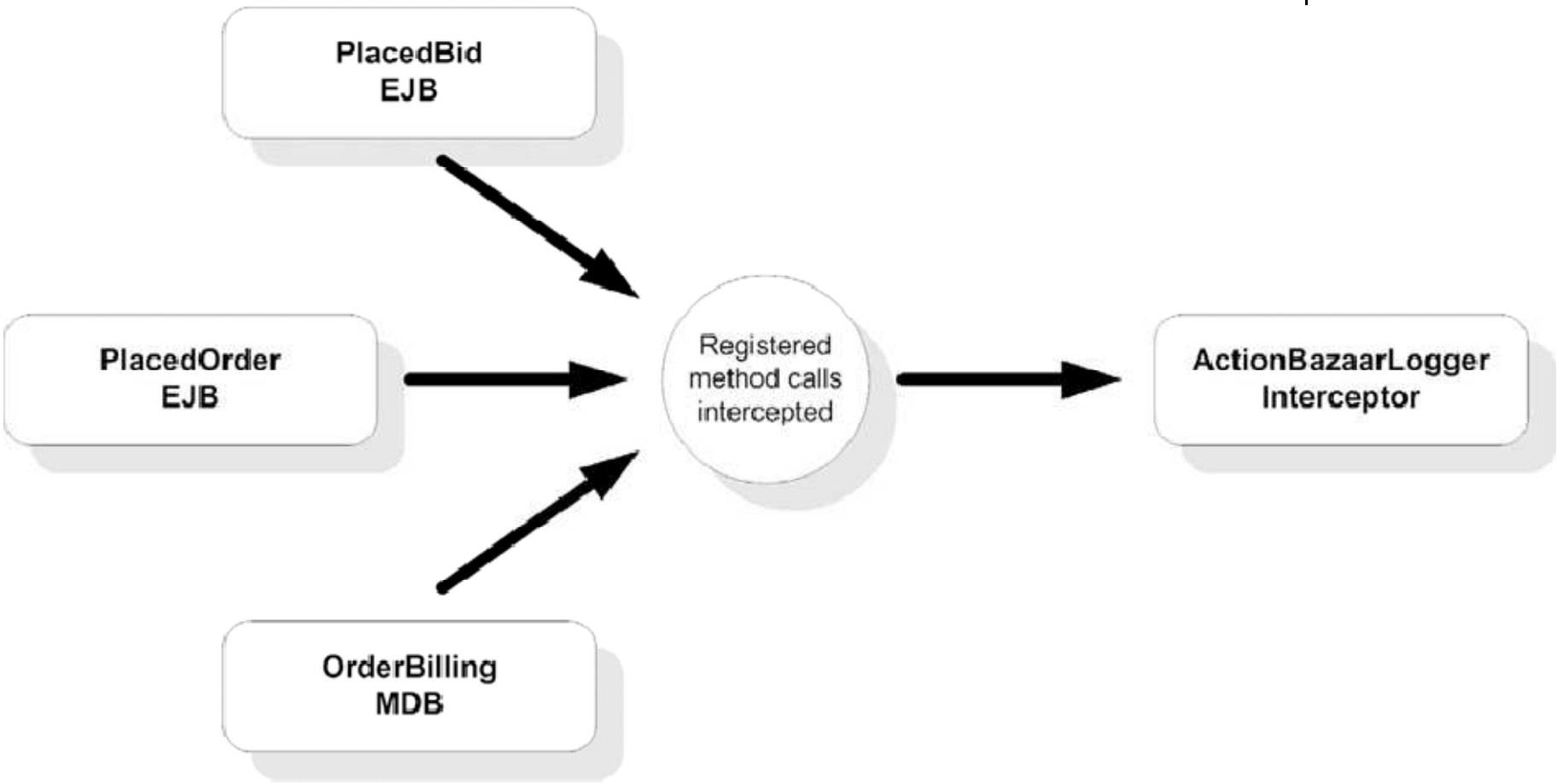
JNDI InitialContext.lookup()


```
Context context = new InitialContext();
String path =
      "java:comp/env/ejb/BidderAccountCreator“;
BidderAccountCreator accountCreator =
      (BidderAccountCreator) context.lookup(path);
...
accountCreator.addLoginInfo(loginInfo);
...
accountCreator.createAccount();
```

Interceptors

- Son objetos invocados automáticamente cuando un método de un EJB es invocado
  - objetivo, modelar crosscutting concerns
  - ofrecen mecanismo AOP
- Si bien cumplen con una funcionalidad muy útil, no pueden cubrir el conjunto de funcionalidades brindados por un buen framework de aspectos

Interceptors



Interceptors

- Los interceptors son “around invoke advices” en la jerga AOP
- Se pueden aplicar tanto a Session Beans como a Message Driven Beans

Interceptors

```
@Stateless
public class PlaceBidBean
                    implements PlaceBid {
   ...
 @Interceptors(ActionBazaarLogger.class)
 public void addBid(Bid bid) {
  ...
 }
}
```

Interceptors

```
public class ActionBazaarLogger {
  @AroundInvoke
  public Object logMethodEntry(
      InvocationContext invocationContext)
  throws Exception {
    System.out.println(
      “Entering method: ” +
      invocationContext.getMethod().getName());
    return invocationContext.proceed();
  }
}
```

@Interceptors

- Permite especificar uno o más interceptores para una clase o un método

```
@Interceptors(ActionBazaarLogger.class)
public void addBid (...
```

@Interceptors

- Podemos aplicarla también a toda una clase
- En este caso, el interceptor es disparado cuando cualquier de los métodos es invocado

```
@Interceptors(ActionBazaarLogger.class)
@Stateless
public class PlaceBidBean implements PlaceBid {
    public void addBid (...
    public void addTimeDelayedBid (...
}
```

Around invoke

```
@AroundInvoke
public Object logMethodEntry(
     InvocationContext invocationContext)
throws Exception {

   System.out.println(
      "Entering method: " +
      invocationContext.getMethod().getName());

   return invocationContext.proceed();
}
```

Around invoke

- Cualquier método designado de esta forma, debe seguir el patrón
  - `Object <METHOD>(InvocationContext) throws Exception`
- La clase `InvocationContext` provee una serie de métodos que permiten obtener información sobre el método invocado

Around invoke

- La llamada a `InvocationContext.proceed` es critica en el interceptor
- Es la que permite que la cadena de llamadas continúe
- No realizar esta invocación, implicará detener la cadena de llamadas, no invocando el método de negocio, ni el otros posibles interceptores

Around invoke

```
@AroundInvoke
public Object validateSecurity(
          InvocationContext invocationContext)
throws Exception {
   if (!validate(...)) {
      throw new SecurityException(
         "Security cannot be validated. " +
         "The method invocation " +
         "is being blocked.");
   }
   return invocationContext.proceed();
}
```

InvocationContext interface

```
public interface InvocationContext {
    public Object getTarget();
    public Method getMethod();
    public Object[] getParameters();
    public void setParameters(Object[]);
    public Map<String,Object> getContextData();
    public Object proceed() throws Exception;
}
```

Conceptos de mensajería

- Cuando hablamos de mensajería en el mundo Java EE, de lo que realmente hablamos es de comunicación entre procesos, débilmente acoplados y asíncronos

Conceptos de mensajería

- Por ejemplo, uno se comunica sincrónicamente cuando llamamos a alguien por teléfono
- Qué sucede si la persona no está?
  - En general mandamos un mensaje (SMS) o dejamos un mensaje en el contestador
  - Luego, nos desentendemos de la llamada

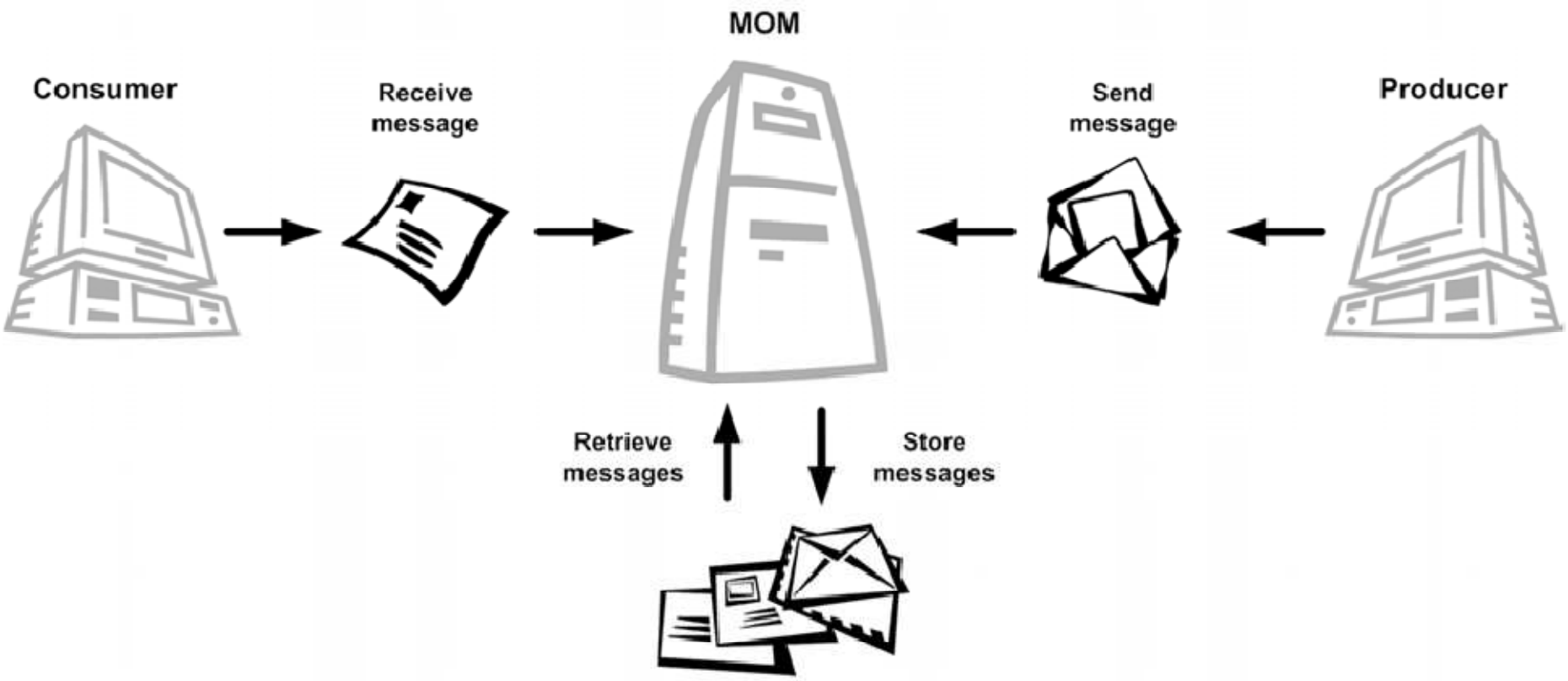
Conceptos de mensajería

- Message-oriented middleware (MOM) permite el uso de mensajería, actuando como el middleman entre el emisor de un mensaje y el receptor del mismo, de forma de que ambas partes no tengan que estar presentes a la vez

Message-oriented middleware

- El emisor del mensaje se denomina **productor**
- La location donde se almacena el mensaje, se denomina **destination**
- Los componentes que reciben los mensajes son denominados **consumers**

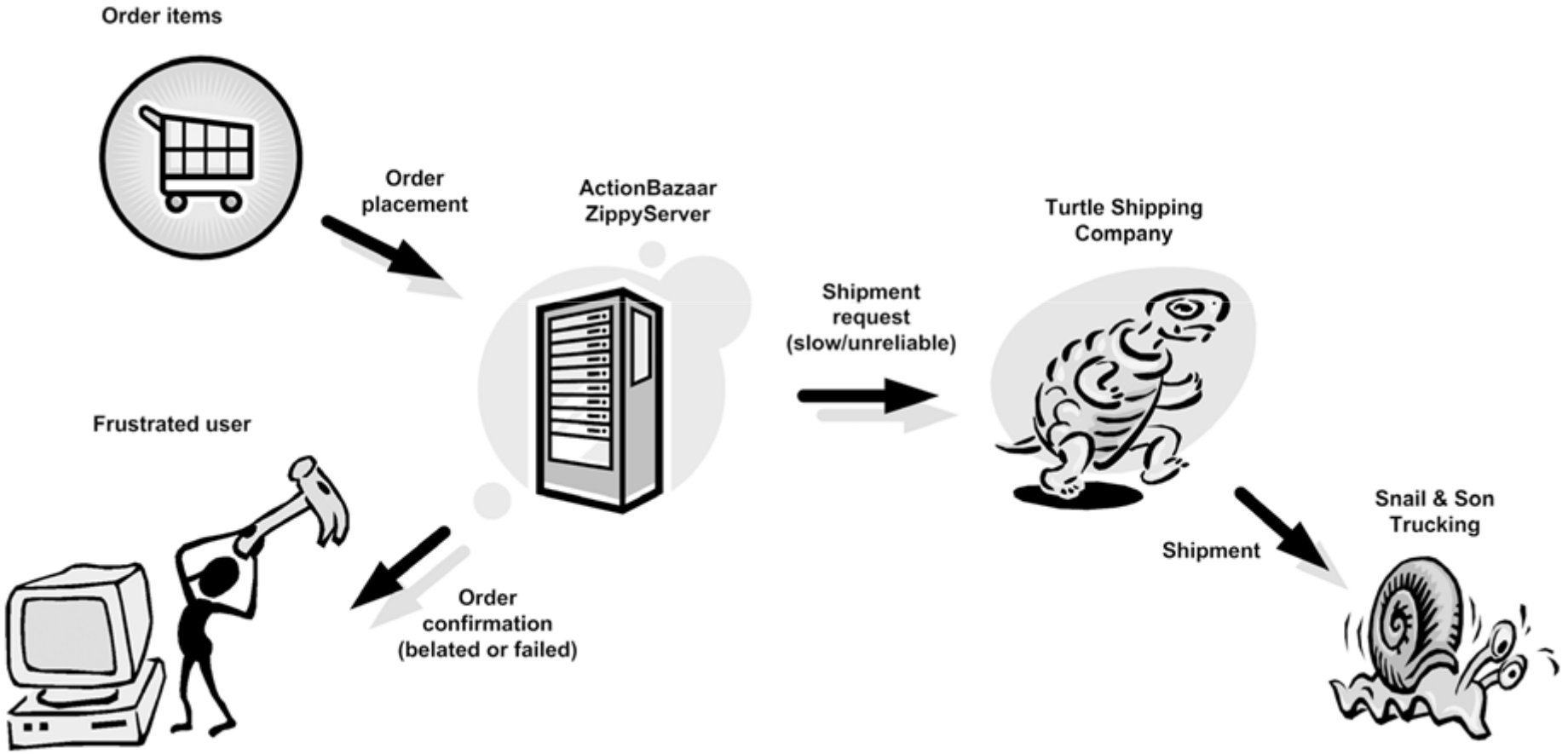
Message-oriented middleware



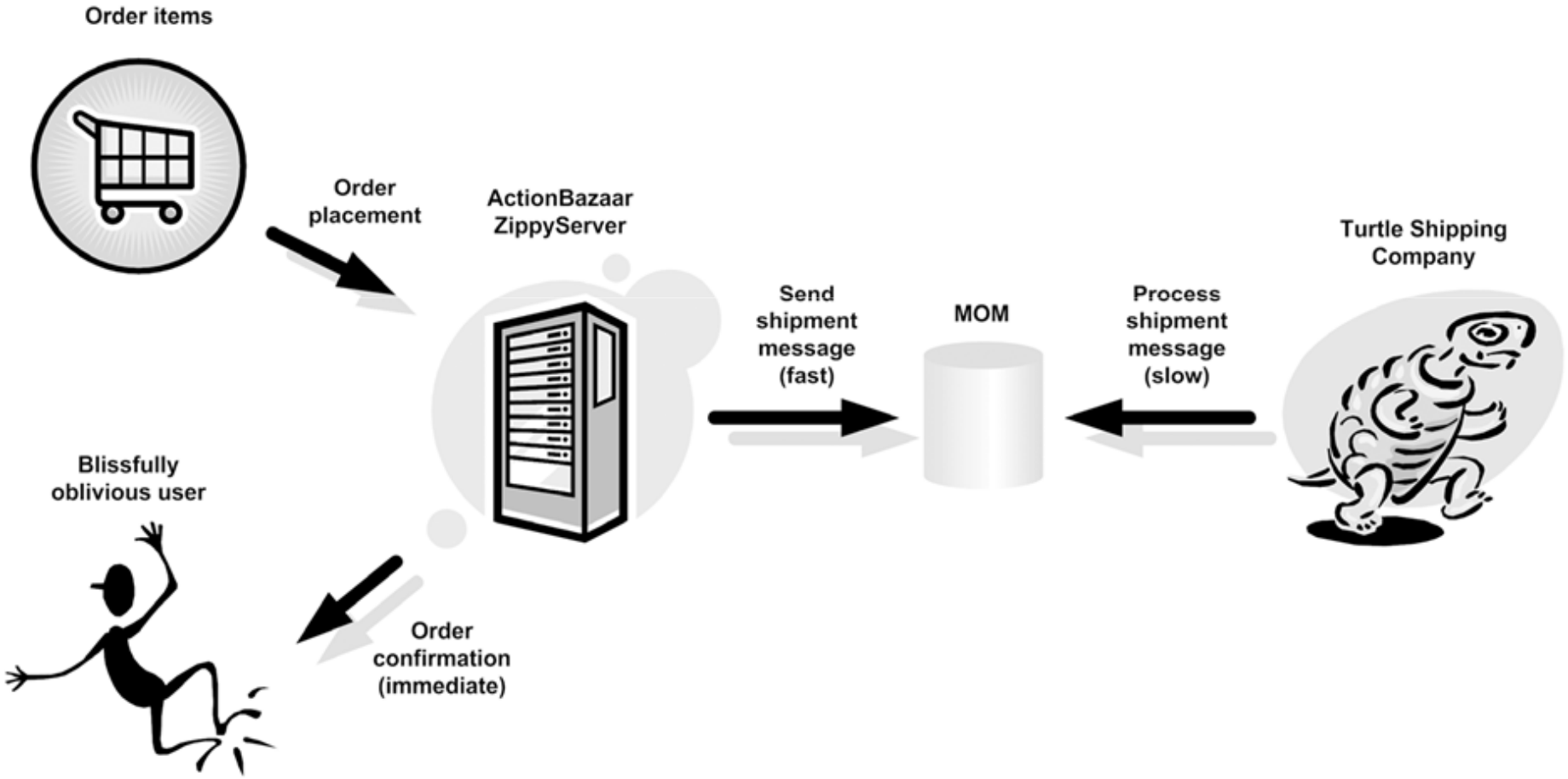
Message-oriented middleware

- Los MOM no son un concepto nuevo
- Productos de MOM existen hace mucho tiempo
- Ejemplos de estos:
  - IBM Web-Sphere MQ
  - TIBCO Rendezvous
  - SonicMQ
  - ActiveMQ
  - Oracle Advanced Queuing

Messaging en acción



Messaging en acción



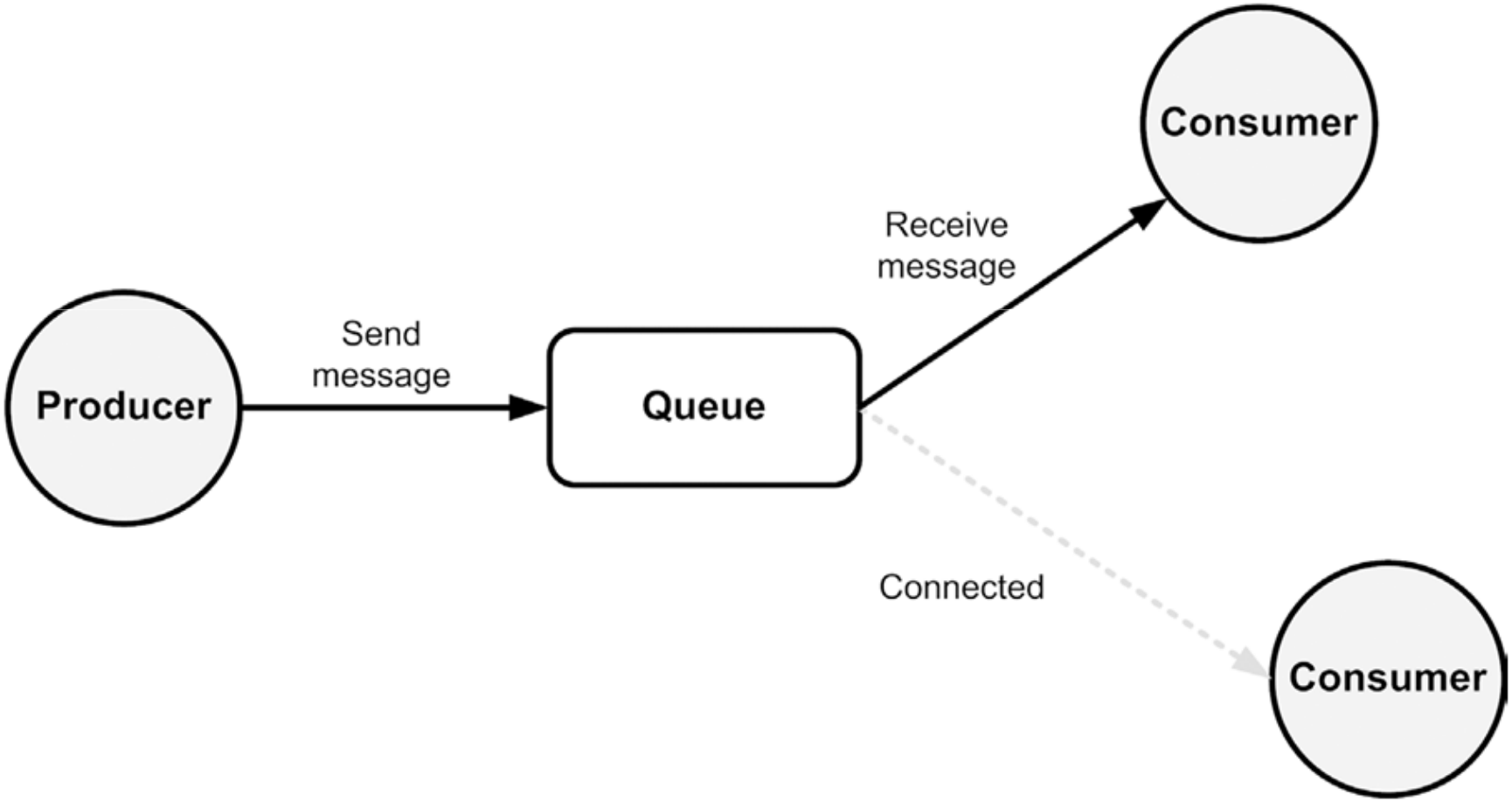
Messaging models

- Un modelo de mensajería es una forma de producir y consumir mensajes, cuando existe más un consumidor y más de un productor
- En Java EE tenemos dos modelos
  - Point to Point (PTP)
  - Publish – Subscribe (Pub-Sub)

Point to point

- Un solo mensaje viaja desde un único productor, a un único consumidor
  - Puede haber más de un consumidor
- Las destinations PTP se denominan **queues** o colas de mensajes

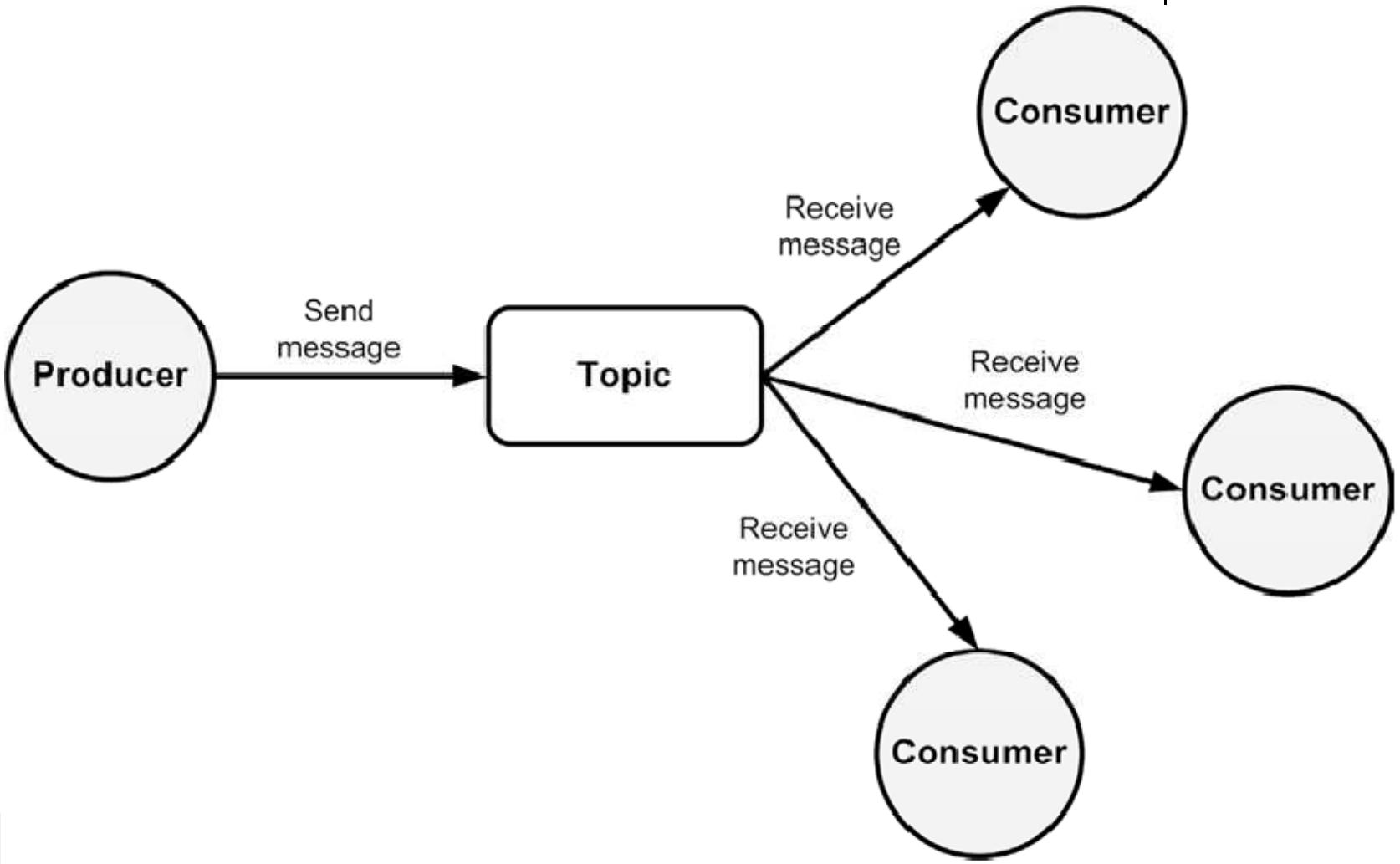
Point to point



Publish – Subscribe

- Este mecanismo es muy similar a realizar un post en un newsgroup / grupo de noticias
- El message destination en este modelo se denomina **topic** y el consumidor se denomina **subscriber**
- **Pub-Sub** funciona muy bien cuando queremos hacer broadcasting de información en un sistema

Publish – Subscribe



Java Messaging Service

- El API JMS es el equivalente en el mundo de los mensajes, del API JDBC en el mundo de las bases de datos
- JMS provee una forma unificada de acceder a un MOM desde Java, evitando el uso de APIs propietarias
- Salvo el caso MSMQ, los demás MOMs soportan JMS

JMS message producer

- Inyecta el connection factory y la destination queue

```
@Resource(name="jms/QueueConnectionFactory")
private ConnectionFactory connectionFactory;

@Resource(name="jms/ShippingRequestQueue")
private Destination destination;
```

JMS message producer

- Se conecta, crea una sesión de uso, y luego crea el producer de mensajes

```
Connection connection =
   connectionFactory.createConnection();
Session session =
   connection.createSession(true,
            Session.AUTO_ACKNOWLEDGE);

MessageProducer producer =
   session.createProducer(destination);
```

JMS message producer

- Creamos el payload…

```
ObjectMessage message =
          session.createObjectMessage();
ShippingRequest shippingRequest =
          new ShippingRequest();

shippingRequest.setItem(item);
shippingRequest.setShippingAddress(address);
shippingRequest.setShippingMethod(method);
shippingRequest.setInsuranceAmount(amount);
```

JMS message producer

- Colocamos el payload, y enviamos el mensaje

```
message.setObject(shippingRequest);

producer.send(message);

session.close();
connection.close();
```

JMS message producer

- Algunos comentarios
  - Como en el caso de JDBC, los objetos administrados que representan la conexión y la factory, se obtienen por inyección
  - El objeto Connection representa una conexión viva con el MOM
    - Objeto costoso de crear
    - Diseñado para ser compartido, thread-safe
  - La sesión provee un contexto single-thread para el envío y recepción de mensajes

JMS message producer

- Algunos comentarios
  - La sesión no es usada directamente para mandar mensajes
  - Se utiliza para esto, un MessageProducer
  - Para el envío del mensajes, se crea un objeto Request (a nivel de aplicación), el cual es colocado dentro de un mensaje
  - A través del producer, el mensaje es enviado

Message-driven beans

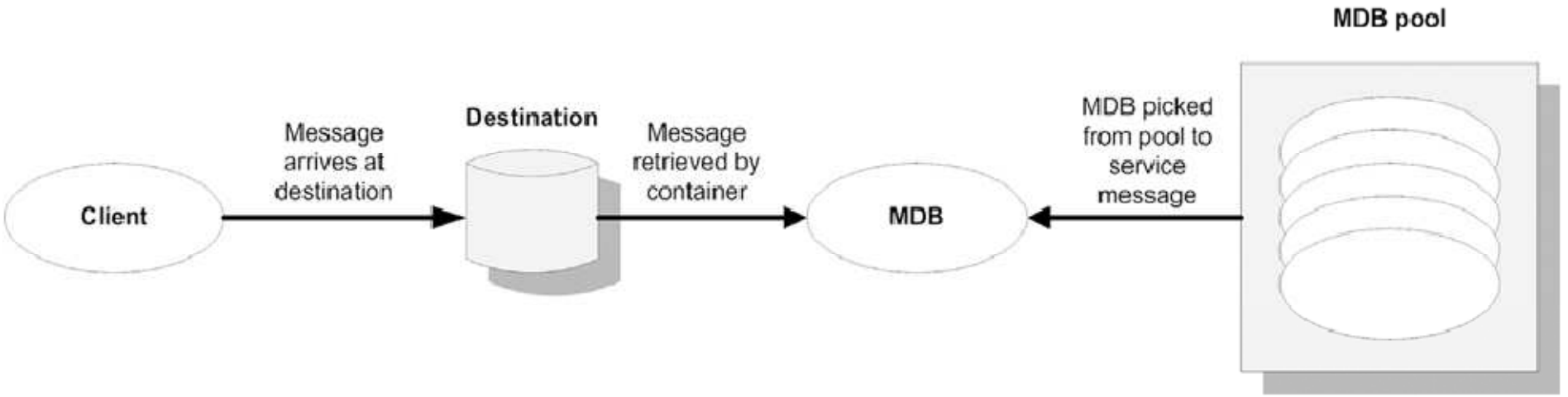
- Son un tipo de componente EJB, diseñado para el consumo de mensajes asincrónicos, como los expuestos anteriormente
- En nuestro caso, los MDBs actuarán como consumidores de mensajes JMS, producidos por otras (o la misma) aplicaciones en la organización

Multithreading

- Los múltiples mensajes que llegan, son procesados por MDBs provenientes de un pool, como en el caso de los stateless session beans
- Cuando un mensaje llega, una instancia de un MDB se saca del pool, se procesa el mensaje, y luego este es devuelto al pool

Multithreading

- Esto se conoce como MDB pooling



Mensajería simplificada

- Los MDBs alivian el costo de implementar el código necesario para manejar los mensajes
- Esto incluye tareas como:
  - Buscar la connection factory
  - Buscar destinations
  - Abrir conexiones
  - Crear sesiones
  - Crear consumidores
  - Adjuntar listeners

Reglas de programación

- La clase del MDB debe implementar la interfaz `javax.jms.MessageListener`
- La clase del MDB debe ser concreta
- La clase del MDB debe ser pública
- La clase del MDB debe proveer un constructor por defecto
- No podemos definir un método “finalize” en la clase del MDB

Reglas de programación

- No debemos propagar ninguna `javax.rmi.RemoteException` o cualquier otra runtime exception
  - La propagación de cualquier runtime exception provocará la terminación de la instancia del MDB

Message Consumer usando MDB


```
@MessageDriven(
  name="ShippingRequestProcessor",
  activationConfig = {
    @ActivationConfigProperty(
      propertyName="destinationType",
      propertyValue="javax.jms.Queue"),
    @ActivationConfigProperty(
      propertyName="destinationName",
      propertyValue="jms/ShippingRequestQueue")
  }
)
public class ShippingRequestProcessorMDB
      implements MessageListener {
```

Message Consumer usando MDB


```
private java.sql.Connection connection;
private DataSource dataSource;

@Resource
private MessageDrivenContext context;

@Resource(name="jdbc/TurtleDS")
public void setDataSource(
                    DataSource dataSource) {
   this.dataSource = dataSource;
}
```

Message Consumer usando MDB

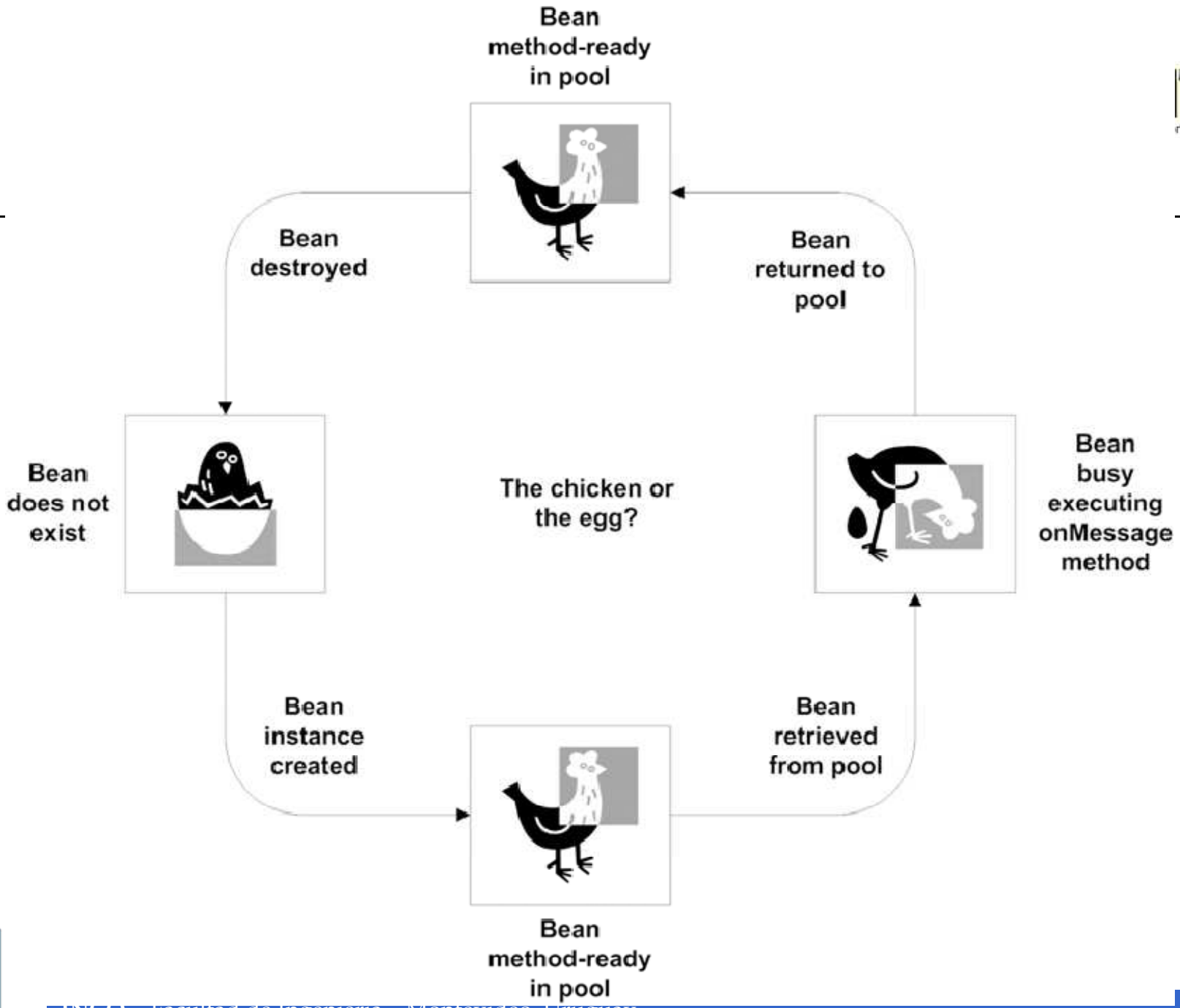

```
@PostConstruct
public void initialize() {
  try {
     connection = dataSource.getConnection();
  } catch (SQLException e) { ... }
}

@PreDestroy
public void cleanup() {
  try {
     connection.close();
     connection = null;
  } catch (SQLException e) { ... }
}
```

Message Consumer usando MDB


```
public void onMessage(Message message) {
  try {
    ObjectMessage objMsg = (ObjectMessage)message;
    ShippingRequest shippingRequest =
            (ShippingRequest)objMsg.getObject();
    processShippingRequest(shippingRequest);
  } catch (JMSException jmse) {
      jmse.printStackTrace();
      context.setRollBackOnly();();
  } catch (SQLException sqle) {
      sqle.printStackTrace();
      context.setRollBackOnly();
  }
}
```

Message Consumer usando MDB


```
private void processShippingRequest(
                              ShippingRequest request)
throws SQLException {
   Statement stmt = connection.createStatement();
   stmt.execute(
     "INSERT INTO “ + "SHIPPING_REQUEST ("
     + "ITEM, " + "SHIPPING_ADDRESS, "
     + "SHIPPING_METHOD, " + "INSURANCE_AMOUNT ) "
     + "VALUES ( " + request.getItem() + ", "
     + "\'" + request.getShippingAddress() + "\', "
     + "\' " + request.getShippingMethod() + "\', "
     + request.getInsuranceAmount() + " )");
}
```


Bean
method-ready
in pool
Bean
destroyed
Bean
returned to
pool
Bean
does not
exist
The chicken or
the egg?
Bean
busy
executing
onMessage
method
Bean
instance
created
Bean
retrieved
from pool
Bean
method-ready
in pool

MDB lifecycle callbacks

- El bean tiene 2 posibles callbacks
  - @PostConstruct
  - @PreDestroy
- La semántica de estas dos es similar a los stateless session beans

Referencias

- Java EE Platform

  http://java.sun.com/javaee/
- EJB 3 In Action. Debu Panda, Reza Rahman, Derek Lane. Manning Publications. 2007.
- Mastering Enterprise JavaBeans 3. Rima Patel Sriganesh, Gerald Brose, Micah Silverman. Wiley. 2006.

  http://www.theserverside.com/tt/books/wiley/masteringEJB3/index.tss