Advanced Java Programming

Legacy Systems

Chris Wong
chw@it.uts.edu.au

Based on notes by Wayne Brooks and the Sun tutorials

Topics

- Legacy systems integration & Java
- Approaches
  - Java Native Interface (JNI)
  - Network protocols (TCP/IP, HTTP)
  - Middleware (RMI, CORBA)
  - Java Message Service (JMS)
  - J2EE Connector Architecture

Legacy Systems

- Most enterprises have existing, legacy systems.
  - “Enterprise Information Systems” (EIS)
  - Mainframe applications
  - Proprietary applications
- These run the “back office” of most medium-large enterprises.
- Current trends:
  - Web enablement – intranet, extranet, internet
  - Application integration – internal integration
  - Need to access legacy systems in enterprise applications

Legacy Systems and Java

- In the J2EE world, anything not written in Java is legacy by definition!
- Works both ways:
  - J2EE app accessed by non-Java, non-web client
    - E.g. Native Windows app accessing EJBs and displaying data
  - J2EE application accessing non-Java servers
    - E.g. EJB connects to mainframe app to retrieve some data
    - Main focus of this lesson

Approaches

- Java Native Interface
  - Make an EJB “half-Java, half-native”
- Network Protocols (TCP/IP, HTTP)
  - Make an EJB establish a network connection
- RPC Middleware (RMI, CORBA, DCOM)
  - Make an EJB call methods on Java/non-Java objects
- Messaging
  - Make an EJB exchange messages with other applications
- J2EE Connector architecture

Topics

- Legacy systems integration & Java
- Approaches
  - Java Native Interface (JNI)
  - Network protocols (TCP/IP, HTTP)
  - Middleware (RMI, CORBA)
  - Java Message Service (JMS)
  - J2EE Connector Architecture
Java Native Interface (JNI)

- Define a Java object containing methods that are implemented with native OS code
  - the Java code acts as "wrapper" for the native code, making the native code facilities available in the Java environment

```java
public synchronized native String getWindowsRegistryKey(String keyname)
```

- the implementation of this method would be written in C/C++ code for the Windows environment
- it would be contained in a Dynamic Link Library (DLL) file that would be loaded at runtime (.so file on Unix)

JNI advantages

- Allows access to non-Java code libraries
  - some vendors may only provide C/C++ libraries to access their proprietary products
- You can 'wrap' Application Programming Interfaces (API) such as SAP
  - however, better to use J2EE Connector Architecture

JNI disadvantages

- A Java object with native methods obviously loses its "write once run anywhere" ability
  - it becomes platform-specific
- Non-distributed solution
  - Java object must run on server with native libraries
- Memory allocation, thread management, transaction management, security management are not shared between Java and native "parts"
  - bad idea to have EJBs with native code – delays/crashes in the native code could adversely affect app server

Topics

- Legacy systems Integration & Java
- Approaches
  - Java Native Interface (JNI)
  - Network protocols (TCP/IP, HTTP)
  - Middleware (RMI, CORBA)
  - Java Message Service (JMS)
  - J2EE Connector Architecture

Network Protocols

- An EJB can create connections to other servers using any TCP/IP protocol
  - therefore can connect to arbitrary TCP/IP servers
  - using the java.net.* libraries
  - often use socket programming

- Advantage:
  - almost unlimited potential for connecting to other servers

- Disadvantage:
  - very low level approach
  - requires understanding of the client/server protocol used

Protocols – HTTP URL

- Accessing HTTP URLs is a special case of TCP/IP
  - but EJB container has special support
- In ejb-jar.xml:
  ```
  <resource-ref>
    <res-ref-name>url/MyHttpEAIProvider</res-ref-name>
    <res-type>java.net.URL</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
  ```
- In MySessionBean.java:
  ```
  InitialContext ctx = new InitialContext();
  URL myURL = (URL) ctx.lookup("java:comp/env/url/MyHttpEAIProvider");
  ```
Protocols – HTTP URL

- Your application treats this connection as a standard stream.
- → Read/Write application specific data.

Uniform Resource Locator (URL)

- Uniform Resource Locator (URL) – allows the identification of resources on the Internet
- See RFC 2396: Uniform Resource Identifiers (URI)
- A URL is made up of:
  - Protocol/Scheme
  - Userinfo (optional)
  - Host
  - port (optional)
  - Path (optional)
  - Query (optional)
- http://cw@learn.it.uts.edu.au:80/dsp/index.htm?print=1

URL & URLConnection classes

- URL class has 2 main methods
  - openConnection() returns a URLConnector object
  - openStream() method returns InputStream
- URLConnection class has many interesting methods eg:
  - getInputStream() – returns InputStream
  - getHeaderField(header) – returns headers
  - See class documentation for more....

URL example

URL u = new URL("http://"+host+"/"+file);
//create a BufferedReader to read the
//data from the URL
in = new BufferedReader(
    new InputStreamReader(
        u.openStream()()));
// & read it in.
while ((line = in.readLine()) != null) {
    System.out.println(line);
}
in.close();

Topics

- Legacy systems Integration & Java
- Approaches
  - Java Native Interface (JNI)
  - Network protocols (TCP/IP, HTTP)
  - Middleware (RMI, CORBA)
  - Java Message Service (JMS)
  - J2EE Connector Architecture

Middleware – RMI

- Accessing RMI servers from EJBs is easy
  - just like accessing RMI server from a standalone client
- Advantage:
  - remote access to Java objects w/o app server overheads
  - can combine with JNI to provide remote access to native services
- Disadvantages:
  - only Java clients can access the RMI server
  - no app server = no transactions, security infrastructure
Middleware – CORBA

• = Common Object Request Broker Architecture

• CORBA is a specification developed by the Object Management Group (http://www.omg.org)

• CORBA is most popular vendor-independent, language-independent, platform-independent middleware environment
  – can have a Windows or Unix native client app accessing EJB in a “standard” way using CORBA calls
  – can have an EJB accessing a CORBA server implemented in almost any language on almost any platform

Middleware – CORBA

• Advantages:
  – allows programmer to link programs written in different languages/platforms into one client.
  – Object-Oriented
  – Re-use existing objects

• Disadvantages:
  – Complex to program
  – Requires a broker to be installed
  – Some integration issues between vendors/languages

Middleware – CORBA

• An Object Request Broker (ORB) is a software layer which provides interoperability between objects implemented on different platforms

• IIOP is the on-the-wire protocol used when CORBA clients make method calls on remote CORBA servers

Middleware – CORBA

• RMI and CORBA have many similarities
  – definition of remote interfaces, generation of stubs and skeletons
  – RMI over IIOP is now standard

• Some differences:
  – CORBA objects may be implemented in many programming languages eg: C, C++, Perl, Java
  – CORBA has its own Naming Service, Transaction Service, etc.

Middleware – CORBA

• CORBA is a popular choice for legacy integration, when Remote Procedure Call (RPC) style interaction is required

• Create a CORBA "wrapper" around legacy system, with defined methods, and allow remote clients (J2EE or other) to call methods
CORBA development

- What do you need to know to create/consume CORBA objects?
- Development process:
  1. Create/Use IDL
  2. Create remote object class (i.e., implement the interface)
  3. Create server code
  4. Create client code

CORBA development - IDL

- CORBA has a Interface Definition Language (IDL)
  - Defines the operations but not the implementation
  - Many bindings for various languages
  - Need to use language/platform/OS specific IDL compiler to generate stubs from the IDL file.
  - IDL supports modularisation, inheritance, operations+attributes of objects, exceptions, datatypes
  - This is based on C++, with some changes for portability between languages & systems

Example of an IDL file

```idl
module StockApp {
  struct Quote {
    string symbol;
    double price;
  };
  exception Unknown{};
  interface Stock {
    Quote get_quote() raises(Unknown);
    void set_quote(in Quote symbol);
    readonly attribute string desc;
  };
  interface StockFactory {
    Stock create_stock(in string symbol, in string desc);
  };
}
```

Equivalent to:

```java
package java package;  
public class Stock {
  String symbol;
  double price;
  
  public Stock() {}  
  public Stock(String symbol, double price) {
    this.symbol = symbol;
    this.price = price;
  }
  
  public String getSymbol() {
    return symbol;
  }
  
  public double getPrice() {
    return price;
  }
  
  public void setSymbol(String symbol) {
    this.symbol = symbol;
  }
  
  public void setPrice(double price) {
    this.price = price;
  }
  
  public String getDescription() {
    return desc;
  }
  
  public void setDescription(String desc) {
    this.desc = desc;
  }
}
```

Standard data type mappings

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>char/wchar</td>
<td>char</td>
</tr>
<tr>
<td>octet</td>
<td>byte</td>
</tr>
<tr>
<td>short/unsigned short</td>
<td>short</td>
</tr>
<tr>
<td>long/unsigned long</td>
<td>long</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>string/wstring</td>
<td>String</td>
</tr>
</tbody>
</table>

Next step: You write server code

1. Develop the implementation
   - Extend the use the classes in the *_ImplBase.java file generated by idlj program
2. Define a main method
3. Initialize the ORB
4. Create at least one object
5. Connect each object to the orb
   (you also need to record the IOR often stored as a string in a text file eg: stock.ref)
6. Wait for requests
   See the following for details:
   http://java.sun.com/developer/onlineTraining/corba/corba.html
CORBA development

Next step: You write client code:
1. Initialise the ORB code, use:
   
   ```java
   org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(argv, null);
   ```
2. Create the CORBA object. Note that you need to have the IOR. Often this is stored as a text file.
   ```java
   BufferedReader in = new BufferedReader(new FileReader("stock.ref"));
   ref = in.readLine();
   org.omg.CORBA.Object obj = orb.string_to_object(ref);
   ```
   contents of stock.ref could look like:
   IOR:010000000d00000049444c3a4563 ....

3. You now use the helper class to convert the CORBA.Object to your class:
   ```java
   Stock s = StockHelper.narrow(obj);
   ```
4. You can now call the remote method:
   ```java
   try {
   Quote s.get_quote();
   ...
   } catch (Unknown e) {
   ...
   }
   ```

To run application, you need to
1. Start Orb
2. Start Server
3. Run client
   - We only lightly cover CORBA here.
   - See more detailed tutorials and information at:
     - OMG website http://www.omg.org
     - Sun Java website
     - Today's tutorial

Topics

- Legacy systems Integration & Java
- Approaches
  - Java Native Interface (JNI)
  - Network protocols (TCP/IP, HTTP)
  - Middleware (RMI, CORBA)
  - Java Message Service (JMS)
  - J2EE Connector Architecture

Messaging

- Messaging is simply communication between components
  - NOT e-mail, instant messaging, SMS etc
- Some applications are more suited for asynchronous programming paradigm
- De-couples applications – sender/receiver are independent!
- Many legacy applications use Message Oriented Middleware (MOM).
  - one application sends a message to another application
  - destination application will process it when it is ready.
  - destination application may not be connected all the time
  - message may contain arbitrary data
    - (e.g. an XML document representing a purchase order)

Messaging Examples

- B2B interaction
  - when stocks are low, my company’s application sends an automated “purchase order” message to a supplier’s messaging system
- Load sharing
  - hundreds of service requests arrive each second – they are placed in a queue and a set of worker EJBs each takes the next item off the queue and processes it
- Legacy Integration
  - a mainframe app holds the number of remaining seats for a sporting event. This data is duplicated in an Entity EJB for a web app. But if someone books a seat through the mainframe, any EJB must be notified immediately that one less seat is available
Messaging Patterns

• Point-to-point messaging

• Publish-subscribe messaging

• Request/Reply messaging pattern

Messaging Patterns

• Point-to-Point messaging (Queue)
  – based on a "Queue"
  – Producers send messages to the queue
  – Consumers remove messages from the queue
  – Each message will be delivered to exactly one consumer
  – Consumer can filter messages based on header

Messaging Patterns

• Publishing-Subscribe (Topic) – called "Pub/Sub"
  – Based on a "topic" – this is a special case of a Queue
  – One-to-Many conversation from a publisher to 1 or more subscriber(s)
  – Publishers can publish messages under a topic
  – Consumers can read messages in a given topic
  – Consumers can filter messages based on header

Messaging Patterns

• "Request/Reply" pattern
  – Producer establishes Queue for replies
  – Producer issues request to Queue or Topic
  – Consumer receives request from Queue to Topic
  – Consumer issues reply to Reply Queue
  – Original Producer receives reply from Reply Queue

Messaging – push vs. pull

• 2 main paradigms on how a consumer gets a message

  1. Push model – message is "pushed" to consumer
     – whenever a new message is available, a method on the consumer object is called to process the message

  1. Pull model – consumer has to ask for message
     – a consumer can periodically poll the message queue to check if any new messages have arrived since it last checked

Messaging – reliability

• A message queue/topic may be persistent, meaning that if the server crashes the messages are not lost
  – implemented using backend file system or DBMS

• Messages may be acknowledged
  – different models, including:
    • no acknowledgement expected
    • automatic acknowledgements sent when message received
    • application sends acknowledgement manually
Messaging and Java

- Main vendors of non-Java MOM products:
  - IBM MQSeries
  - Microsoft MSMQ

- Java Message Server (JMS) = standard Java API for accessing these MOM’s.
  - NOT the same as JavaMail API
  - Provides a standard interface to vendor specific implementation of the MOM
  - Some vendors provide built in pure Java MOMs
    - e.g.
      - WebLogic
      - SonicMQ

- Java Message Server (JMS) = standard Java API for accessing these MOM’s.
  - NOT the same as JavaMail API
  - Provides a standard interface to vendor specific implementation of the MOM
  - Some vendors provide built in pure Java MOMs
    - e.g.
      - WebLogic
      - SonicMQ

- JMS supports all these combinations of messaging
- Application servers typically support an implementation of JMS
  - i.e. you can set up message queues and topics within the app server, and applications can send and receive messages to these queues/topics

- Application servers typically support integration with external messaging systems
  - messages to your EJB may come from an external, legacy application
  - your EJB may generate messages destined for some external messaging service

- What is missing in the spec?
  - Security
  - Load Balancing/Fault Tolerance
  - Error Notifications
  - Administration
  - Message Repository
  - Wire Protocol
    - This means you cannot mix JMS implementations eg: WebLogic and MQSeries
    - Connection provided by application server eg: WebLogic message bridge
  - Interoperability with non-Java clients
    - although if you use XML/Text messages, should be ok

- J2EE 1.2 only required the API to be present
- J2EE 1.3 makes implementation a CORE requirement.
- Other J2EE components interact with JMS
  - JDBC
    - May use JDBC in same transaction as JMS operation
  - JNDI
    - Location for configured JMS objects
  - JTA/JTS
    - Messages and other operations can be combined in one transaction
  - EJB
    - Asynchronous bean operations with JMS messages

JMS and J2EE

- JMS Classes
  - ConnectionFactory
    - Provided by Application server, Creates Connections
  - Connection
    - Active connection to JMS provider
  - Destination
    - Identity of a message destination
  - Session
    - single-threaded context for sending/receiving messages
  - MessageProducer
    - object that sends messages to a destination. Created by a session
  - MessageConsumer
    - object that receives messages from a destination. Created by a session

JMS class relationships
PTP and Pub/Sub Interfaces

<table>
<thead>
<tr>
<th>JMS Parent</th>
<th>Point-to-Point</th>
<th>Pub-Sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionFactory</td>
<td>QueueConnectionFactory</td>
<td>TopicConnectionFactory</td>
</tr>
<tr>
<td>Connection</td>
<td>QueueConnection</td>
<td>TopicConnection</td>
</tr>
<tr>
<td>Destination</td>
<td>Queue</td>
<td>Topic</td>
</tr>
<tr>
<td>Session</td>
<td>QueueSession</td>
<td>TopicSession</td>
</tr>
<tr>
<td>MessageProducer</td>
<td>QueueSender</td>
<td>TopicMessageConsumer</td>
</tr>
<tr>
<td>MessageConsumer</td>
<td>QueueReceiver</td>
<td>TopicSubscriber</td>
</tr>
</tbody>
</table>

JMS setup

- The J2EE application server must be configured for JMS
- It must provide a JNDI registry of
  - The ConnectionFactory (specific to implementation)
  - The Destination (Queue/Topic)
- This is usually specific to your application server e.g. WebLogic Console
- Some application servers provide bridges to alternative JMS engines e.g. WebLogic Foreign message bridge

JMS setup

- First step is to get the ConnectionFactory
  ```java
  import javax.jms.*; // import JMS classes
  import javax.naming.*; // ditto for JNDI
  ...
  InitialContext ctx = new InitialContext();
  QueueConnectionFactory qconFactory = (QueueConnectionFactory)ctx.lookup("weblogic.jms.ConnectionFactory");
  ...
  ```
- Here we use the weblogic specific JMS provider. You can define your own e.g. `myConnectionFactory`

Sender Example-1

- Next, get the Connection, Session, Destination "queue" & MessageProducer "sender"
  ```java
  QueueConnection qcon = qconFactory.createQueueConnection();
  QueueSession qsession = qcon.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
  Queue queue = (Queue) ctx.lookup("jms.MyMessageQueue");
  QueueSender qsender = qsession.createSender(queue);
  ```

Sender Example-2

- NOTE: regarding acknowledgements
- When we create a session, we have 2 parameters `CreateQueueSession(transaction?, ack_flag)`
  - `Transaction?` indicates if this is transaction oriented (we chose false)
  - `Ack_flag` indicates type of acknowledgement of messages
    - `Session.AUTO_ACKNOWLEDGE`
    - system automatically acknowledges message receipt.
    - `Session.CLIENT_ACKNOWLEDGE`
    - client manually acknowledges
    - `Session.DUPS_OK_ACKNOWLEDGE`
    - session acknowledges later, and duplicate messages are ok
Sender Example-4

- Next, create the message. This can be various types ie:
  - StreamMessage - self-describing sequence of Java objects/primitives
  - MapMessage - key-value pairs
  - TextMessage - single string or message body
  - ObjectMessage - single serialized object as message body
  - BytesMessage - stream of bytes
- Then, start the connection, and send the message
  ```java
  TextMessage msg = qsession.createTextMessage();
  qcon.start();
  msg.setText("Hello world");
  qsender.send(msg);
  ```

  Note: use specific methods for each message types

Message formats

- Messages have 3 main sections
  - Header
    - Used by clients and providers to identify and route messages
    - JMS standard headers e.g.
      - JMSDestination, JMSMessageID, JMSReplyTo, JMSRedelivered, JMSType, JMSTimeStamp, JMSCorrelationId, JMSDeliveryMode (PERSISTENT/NON_PERSISTENT), JMSExpiration, JMSPriority
    - Or user defined.
  - Properties
    - Application-specific, Standard, and Provider-specific
  - Body
    - Message contents eg: TextMessage

Message Properties

- Messages can have user defined properties
- Set property via `Message.setProperty()`
  - Where `Type` is one of boolean, byte, short, int, long, float, double, String
  - Eg: `msg.setProperty("isStaff",true)`
- Retrieve the property via `Message.getProperty()`
  - Eg: `msg.getBooleanProperty("isStaff")`

Message Selectors

- You can filter messages by headers or properties
  - This syntax is similar to SQL99.
  - Use as 2nd parameter on QueueSession.createReceiver() or TopicSession.createSubscriber()
  - Examples:
    - "isStaff = TRUE"
    - "mycount > 5"
    - "JMSPriority > 3"

Receiver example - 1

- Because application is asynchronous, we have to write thread safe, event driven code
- Our receiver has to implement MessageListener and have a method called onMessage(Message)
- JMS provider will call this method when it receives a message destined for this program.
- Note that messages NOT matching filter will be ignored.

Receiver example - 2

- We use JNDI to find the connection factory and destination "Queue"
  ```java
  QueueConnectionFactory qcf = (QueueConnectionFactory)ctx.lookup("weblogic.jms.ConnectionFactory");
  QueueConnection qcon = qcf.createQueueConnection();
  QueueSession qsess = qcon.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
  Queue queue = (Queue) ctx.lookup("jms.MyMessageQueue");
  ```
Receiver example - 3

- We now start a Queue receiver
  QueueReceiver qreceiver = qsession.createReceiver(queue);
- And set this class to be activated when we receive a message
  qreceiver.setMessageListener(this);
- And now start the thread which listens for incoming messages
  qcon.start();

Message-Driven Beans

- Under EJB 1.1 spec, EJB's could only produce messages, not receive them.
- You needed to write wrapper classes and RMI proxies to receive messages
- EJB 2.0 spec introduced a better method – Message Driven Beans (MDB)
- Not to be confused with Management Beans (MBeans)

Message-Driven Beans

- Message-Driven Beans have no methods that can be invoked by users directly
  - unlike Session and Entity Beans
- Instead, Message-Driven Beans are message receivers, and implement the onMessage() method
  - A Message-Driven Bean is attached to a message Queue
  - Whenever a new message arrives on the queue, the Bean will execute its onMessage() method
  - The only way to interact with Message-Driven Beans

Receiver example - 4

- Our application should now just wait for incoming connections. You might use the wait() method for this.
- You need to have a onMessage(Message) method, which is invoked when a message arrives
  public void onMessage(Message msg) {
    String msgText = ((TextMessage)msg).getText();
  }
- Note that you can have different message types. Your method needs to cope with any of them.

Message Driven Beans

- MDBs integrate EJBs with JMS
- Act as a standard JMS message consumer.
- Is a stateless component invoked by the EJB container as a result of receiving messages from a JMS Queue or Topic
- The MDB then performs business logic based on the message contents

Message Driven Beans

- Unlike session or entity beans, MDBs have no home or remote interface, and therefore cannot be directly accessed by internal or external clients
- Clients interact with MDBs only indirectly, by sending a JMS message to a JMS Queue or Topic
- The EJB container automatically creates and removes MDB instances as needed to process incoming messages
- The goal of the MDB model is to ensure that developing an EJB that is asynchronously invoked to handle the processing of incoming JMS messages is as easy as developing the same functionality in any other JMS MessageListener
Message Driven Beans

- Queue or Topic is defined in the deployment descriptor.
  - Message selector is also defined in DD - restrictive

- MDB must implement `javax.jms.MessageListener` and `javax.ejb.MessageDrivenBean`

- `onMessage()` must not throw JMS or application exceptions; instead they must be caught or handled. At least, they could be re-thrown as `EJBException`

- `MessageDrivenBean` specifies two methods:
  - `ejbRemove()`
    - Is invoked just before the bean is removed
  - `setMessageDrivenContext(MessageDrivenContext ctx)`
    - Sets the context for the bean

- `ejbCreate()` needs to be defined as well

MDB Life Cycle

- `newInstance()`
- `setMessageContext(mdc)`
- `onMessage()`
- `ejbCreate()`
- `ejbRemove()`

Example – Message-driven EJB2

```java
import javax.ejb.*;
import javax.jms.*;

public class MessageTraderBean implements MessageDrivenBean,
MessageListener {
    public void ejbCreate () throws CreateException { ... }
    public void ejbRemove() { ... }
    public void setMessageDrivenContext(MessageDrivenContext ctx) { ... }
    public void onMessage(Message msg) { try {
        // process message
        } catch (Exception ex) { ... }
    }
}
```

Example – Message-driven EJB3

```java
import javax.ejb.*;
@MessageDriven
public class MessageTraderBean implements
MessageListener {
    public void onMessage(Message msg) {
        try {
            // process message
        } catch (Exception ex) { ... }
    }
}
```

Example – Message-driven bean ejb-jar.xml

```xml
<ejb-jar>
<enterprise-beans>
<message-driven>
<ejb-name>exampleMessageDriven</ejb-name>
<ejb-class>examples.ejb20.message.MessageTraderBean</ejb-class>
<transaction-type>Container</transaction-type>
<message-driven-destination>
<jms-destination-type>javax.jms.Topic</jms-destination-type>
</message-driven-destination>
<security-identity>
<run-as-specified-identity>
<role-name>foo</role-name>
</run-as-specified-identity>
</security-identity>
</message-driven>
</enterprise-beans>
</ejb-jar>
```

Messaging summary

- Messaging is becoming an important topic in enterprise application development

- The messaging model is scalable, and well-suited to many forms of interactions between loosely-coupled enterprise systems

- More available than was presented in this lesson
  - E.g. publish/subscribe to Topics with many receivers
  - Messaging and transactions

- See http://edocs.bea.com/wls/docs100/jms/
Topics

- Legacy systems Integration & Java
- Approaches
  - Java Native Interface (JNI)
  - Network protocols (TCP/IP, HTTP)
  - Middleware (RMI, CORBA)
  - Java Message Service (JMS)
- J2EE Connector Architecture

J2EE Connector Architecture

- Intended as a standard way for third-party vendors to allow J2EE applications to access their proprietary systems
- Vendor provides a "Resource Adapter" for their system, which plugs into a J2EE container
  - JCA compliant
  - EJBs can then access the vendor system via the resource adapter, using a standard, generic API - CCI
  - Very similar to access relational databases via JDBC
  - Blackbox resource adapters provide a mock environment for testing purposes
- J2EE Connector Architecture
  - J2EE compliant server must support for JCA
  - JCA defines following services
    - System level contracts for J2EE server and resource adapter to co-ordinate with each other (security, transactions, etc.)
    - Common Client Interface (CCI) defines client API that J2EE components (EJBs, JSPs etc.) can use to connect and interact with EIS
  - Allows various EIS resource adapters to plug into J2EE applications

J2EE Connector process

- An application using a J2EE Connector would:
  - Locate a ConnectionFactory using JNDI lookup
  - Create a Connection (using ConnectionFactory)
  - Create an Interaction (using Connection)
  - Call Interaction.execute(...) method to access remote system

From "Programming weblogic J2EE Connectors" bea.com

J2EE Connector process

- Status:
  - support required by J2EE 1.4 app servers
  - Holds future promise for a uniform method for accessing legacy systems from Java
  - if/when vendors provide Resource Adapters for their particular legacy systems
Summary

- Many approaches to accessing legacy systems
- RMI/IIOP allows you to do RPC to existing Java RMI applications
- Network protocols allow you to connect to network based applications – eg. By using Sockets or HTTP URLs.
- CORBA is good for connecting to existing CORBA based legacy systems.
- JMS is good for message-oriented interactions which seem well suited to many legacy integration tasks
- J2EE Connector Architecture provides a neutral mechanism to connect to vendor specific API's