



## CHAPTER 1

# FUNDAMENTALS

## OBJECTIVES

*After completing "Fundamentals," you will be able to:*

- Describe the motivation for the Java Message Service, and its place in the broader Java EE architecture.
- Distinguish between point-to-point and publish-and-subscribe messaging styles.
- Describe the use of the Java Naming and Directory Interface in JMS applications.
- Send messages to a JMS queue and receive them by various means including synchronous receipt, asynchronous listening, and browsing.
- Publish messages to a JMS topic and receive them using subscribers.

## Asynchronous Messaging

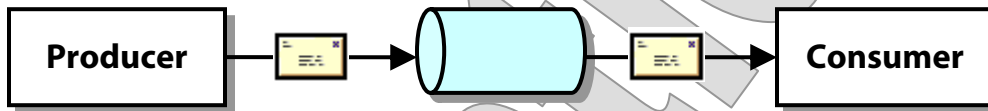
---

- **Distributed application components sometimes require services that are not immediately available.**
  - This could be due to **network or server failure**, or unacceptably **low availability** of services, meaning that the caller would have to wait too long for a response.
  - A **web component** might need to act very quickly so as to stay **responsive** to a high volume of incoming HTTP requests – even knowing that complete processing of a request will be transferred to another process or host, and/or deferred.
  - It could also be that the producer of some information needs to communicate it in **real time**, but the consumer is a **batch process** that only runs periodically.
- **Also, some communiqués do not require the reliability of a request-response pattern, but rather can live with a best-effort approach to delivery.**
- **For either of these needs, synchronous method invocation – HTTP request/response, Java RMI, etc. – is ill-suited.**
- **What is needed is a means of passing a message from one place to another asynchronously.**
  - Where the application demands it, the **delivery, receipt**, and even **acknowledgement** of the message must be at least as **reliable** as a synchronous method call.
  - Lesser reliability is sometimes acceptable (or even preferable for performance reasons), and this too should be part of the messaging infrastructure.

## The Java Message Service

---

- The Java 2 Enterprise platform includes the **Java Message Service**, or **JMS** – at version 1.1 as of this writing.
- JMS defines the message **destination**, which acts as an intermediary between the **producer** and **consumer** of a message.

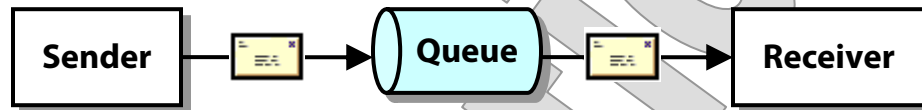


- This allows the producer and consumer to operate in blissful ignorance of each other.
  - There are no **semantic dependencies** between the two; they mutually depend on the semantics of the JMS object.
  - There are no **timing dependencies**, except perhaps for the familiar attributes of time itself: the consumer can't receive a message until it's produced!
  - Both parties are known as **clients** of the JMS service; this term is not being used in the context of client/server systems here.
- JMS defines both reliable and best-effort service levels, called **delivery modes**.
  - We will mostly take advantage of what JMS calls **persistent** messages, which survive the failure of the JMS provider.
  - We'll see that applications that might only require best-effort delivery can derive a performance benefit by using **non-persistent** messages instead.

## Point-to-Point Messaging

---

- JMS offers two distinct messaging styles: **point-to-point** and **publish-and-subscribe**.
- The point-to-point model is the more intuitive, and the more commonly used:

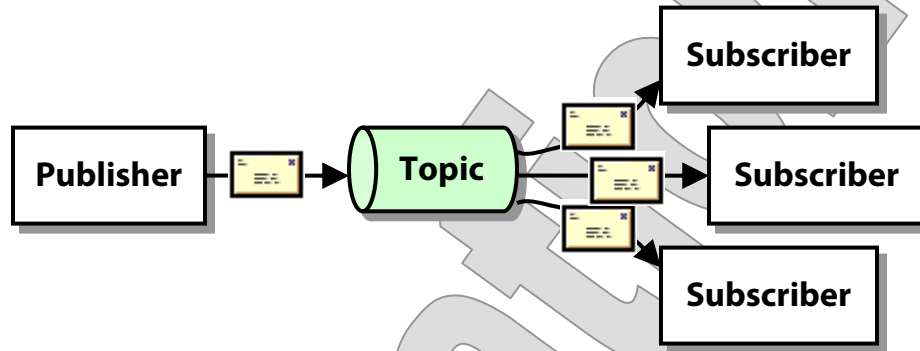


- The producer is called a **sender**, and this client sends messages to a **queue**.
- The consumer is called a **receiver**, and gets messages off the queue in the order in which they were sent (with a few exceptions such as message priority and failure recoveries).
- The receiver does truly **consume** the message; it is removed from the queue.
- We'll see a few other sorts of point-to-point clients in a moment.
- **Both parties enjoy benefits of working with the queue as a messaging intermediary:**
  - The sender doesn't need to block waiting for final processing.
  - The receiver can retrieve messages at its convenience.
  - Yet we have total reliability, which along with asynchronicity is one of the major value propositions of JMS.

## Publish-and-Subscribe Messaging

---

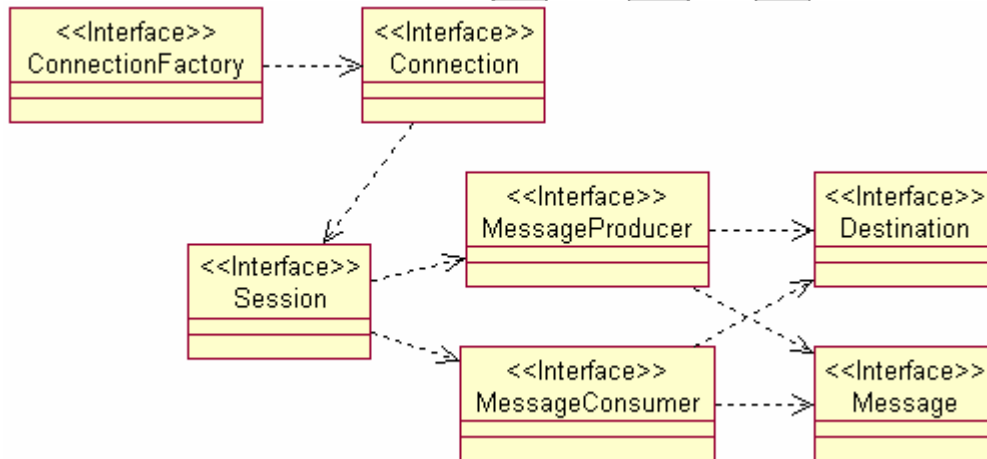
- In the publish-and-subscribe style, multiple parties can receive a message from the JMS destination.



- The producer is called a **publisher**.
  - The destination is called a **topic**.
  - Consumers are called **subscribers**, and (in the simplest usage anyway) they must be **active** and listening for messages at the time they are published, in order to receive them.
- **Publish-and-subscribe offers a different feature set from point-to-point:**
    - It supports **broadcasting**, whereas a queued message can only be consumed by one party.
    - It does not support **deferred consumption**; subscribers must be active at the time of publishing to receive the message.
    - For this reason, while features such as **message persistence** and reliability are technically supported, they are less interesting.
    - Later we'll discuss **durable subscribers**, which can be dormant when a message is published and JMS will activate them. Still, there is no deferred processing; receipt is immediate.

## Interface Model

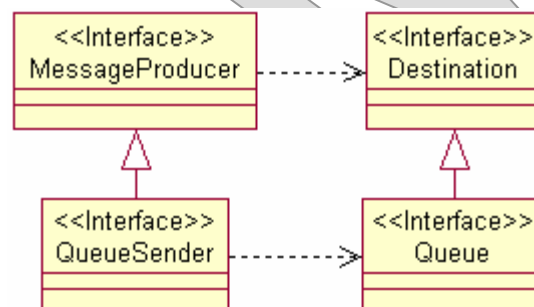
- The JMS API defines a unified interface model for both messaging styles, in the package **javax.jms**.
  - Many of the dependencies shown below are factory relationships: **ConnectionFactory** creates a **Connection** creates a **Session**, etc.



- Then at the end, we see producers and consumers not creating but simply using destinations.

- Each style defines a subtype of each interface shown above.

- For instance **Queue** and **Topic** are the **Destination** subtypes.
- Produce to a **Queue** with a **QueueSender**; publish to a **Topic** with a **TopicPublisher**.



## Tools

---

- For our hands-on exercises in JMS, we'll rely on the **WebLogic Server**, which is already installed on your systems and includes:
  - A **Java 6 DK**, including compiler, launcher, and API docs (in fact, two of them! and we'll use **JRockit**)
  - The **Java EE 5** libraries and API docs, including JMS.
  - A Java EE 5 **application server** – this is WebLogic proper – which implements EJB and web containers, JMS message queues, and other Java EE features
  - The Java EE **application client container**, which will host many of exercises deployed as standalone applications
  - The **Ant** make utility, bundled with WL-specific task definitions
- We also use the **MySQL** relational database management system for one of our case studies.
  - We'll come back to this when we need it, which won't be for several chapters.

## Environment

---

- WebLogic is installed on your system, typically in **c:\BEA\WebLogic**.
- Set your development environment as follows:
  - Set an environment variable **CC\_MODULE** – this is used by Ant build files to locate a few centralized tools.

```
set CC_MODULE=c:\Capstone\JMS_WebLogic
```

- Eliminate the environment variables **PATH** and **CLASSPATH**:

```
set PATH=
```

```
set CLASSPATH=
```

- Run the environment-setup script that comes with WebLogic:

```
c:\BEA\WebLogic\server\bin\setWLSEnv
CLASSPATH=c:\BEA\patch_wlw1030\profiles...
PATH=c:\BEA\patch_wlw1030\profiles...
Your environment has been set.
```

- This will set up your **PATH** and **CLASSPATH** variables.

## Environment

---

- Now, create the WebLogic domain and server that we'll use for all the course exercises.

- An Ant target is prepared for this purpose.
- Open a command console and set its working directory to **Examples/Queue/Step1**.
- Then type:

```
ant create-domain
```

```
Created dir: C:\Capstone\JMS_WebLogic\WLDomain
... <Server started in RUNNING mode>
... Killing WLS Server Instance WLServer
... <... listening on 127.0.0.1:8080 was shutdown.>
```

- This target creates a new domain called **WLDomain**, with a single server called **WLServer** that listens for HTTP at port 8080 – all housed in **c:\Capstone\JMS\_WebLogic\WLDomain**.
  - It starts the server, performs some additional configuration, and then shuts the server back down.
- Now, in a separate console (since this console will represent the WebLogic server process, and won't accept further commands), start the new server:

```
ant start-server
```

```
... <Server started in RUNNING mode>
```

- You can leave this server running for the rest of the course, but when you do want to shut it down, run the following from any other console – or simply **Ctrl-C** in the server's console:

```
ant stop-server
```

## Ant Tasks

---

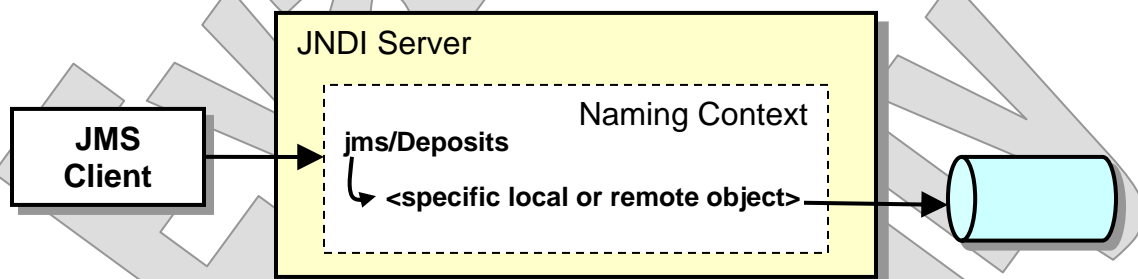
- To simplify repetitive tasks for our exercises, we take advantage of prepared Ant build files.
- We shouldn't dive into every detail of the structure of these Ant files, but there are a couple that deserve some attention.
- The target **build-appclient** creates a JAR file containing:
  - All the compiled **.class files**
  - A deployment descriptor **application-client.xml**, necessary to announce the JAR to the application-client container
  - A **manifest file** with an entry that identifies a specific application class – that is, the class whose main method will be run:

**Main-Class: cc.jms.Sender**

- A prepared script then launches the WebLogic ACC, and asks it to run a specific JAR.
- The upshot is that building with **ant** and then running one of the prepared scripts is roughly equivalent to a simple **javac** compile and **java** launch.
  - The big difference is that the application runs inside the ACC and thus gets the benefit of Java EE API calls and JNDI visibility.
- You don't really need to know these Ant and script details to carry out exercises in this course.
- But, if you're curious, you might want to review the files in **c:\Capstone\JMS\_WebLogic\Ant** in more detail, or discuss them with your instructor.

## JNDI

- In a complex system, each component should ideally be independent of the location of needed resources.
  - Resources (such as databases, message queues, and other Java EE components) might live on different networked machines, hosted in different running processes.
  - Administrators want to be able to **relocate** resources as available hardware evolves, and to balance loads.
  - Also, different deployments will require different absolute locations for the same resources.
- Yet components must find those resources at runtime, in order to interact with them.
- Java defines a standard API to **naming services**: the **Java Naming and Directory Interface, or JNDI**.



- A naming service allows an administrator to **publish** a resource under a unique name.
- A client can then **look up** the resource, knowing the name but not the actual location.
- JMS clients typically use JNDI to locate object factories, queues, and topics.

## JNDI Report

**EXAMPLE**

- In **Examples/JNDIReport**, there is a simple Java EE application client that can show a complete listing of names in the application server's local JNDI context.
  - This will help us get familiar with the use of JNDI for our upcoming exercises, and we can rely on this tool later, to confirm that JMS objects are actually available.
  - It will also give us a quick test of tools and environment.

- The application class **cc.util.JNDIReport** creates a default JNDI **context** and passes it to its own **list** function:

```
list (new InitialContext (), "");
```

- This function just reads out every available name, and calls itself to drill down through any **sub-contexts**.
- The **javax.naming.InitialContext** constructor is very flexible: it will create the appropriate context based on configuration that it discovers at runtime.
  - This may be a **local** or **remote** context.
  - JNDI is ultimately just an interface model; the actual service may be a **CORBA Naming** service, **native JNDI**, **LDAP**, or some other.
  - Client **credentials** required to connect to a secure context may be supplied by system properties or other external means.

## JNDI Report

**EXAMPLE**

- With the application server running, have Ant build the JNDIReport application, and try it out.
  - The prepared script **Report** uses Ant, too, asking it to run the Java class as an application in the **application-client container**.
  - This gives it full visibility to the JNDI service, without having to make a TCP connection from outside the server process.

```
ant
Report
weblogic
ejb
  mgmt
    MEJB
javax
mejbmejb_jarMejb_EO
```

## The Connection Interface

---

- The **Connection** interface represents a live connection to a JMS server:

```
public interface Connection
{
    public String getClientID ();
    public void setClientID (String ID);
    public createSession (boolean transacted,
        int acknowledgementMode);
    public void start ();
    public void stop ();
    public void close ();
}
```

- The **client ID** identifies the JMS client using this connection – uniquely, within some scope as designed by the application developer.
  - The connection is a heavyweight resource and must be **closed** when the application is done with it.
  - **start** and **stop** the connection to manage the flow of messages.
- **Connection** is then a factory for **Session**.
  - Applications rarely use the factory method shown above; since the factory will determine the runtime type of the session, it's far more common to use one of the subtype methods:

```
QueueConnection.createQueueSession ();
TopicConnection.createTopicSession ();
```

## The Session Interface

---

- **Session** represents a body of work performed by the JMS client.

```
public interface Session
{
    public static final int AUTO_ACKNOWLEDGE;
    public static final int CLIENT_ACKNOWLEDGE;
    public static final int DUPS_OK_ACKNOWLEDGE;
    public static final int SESSION_TRANSACTED;

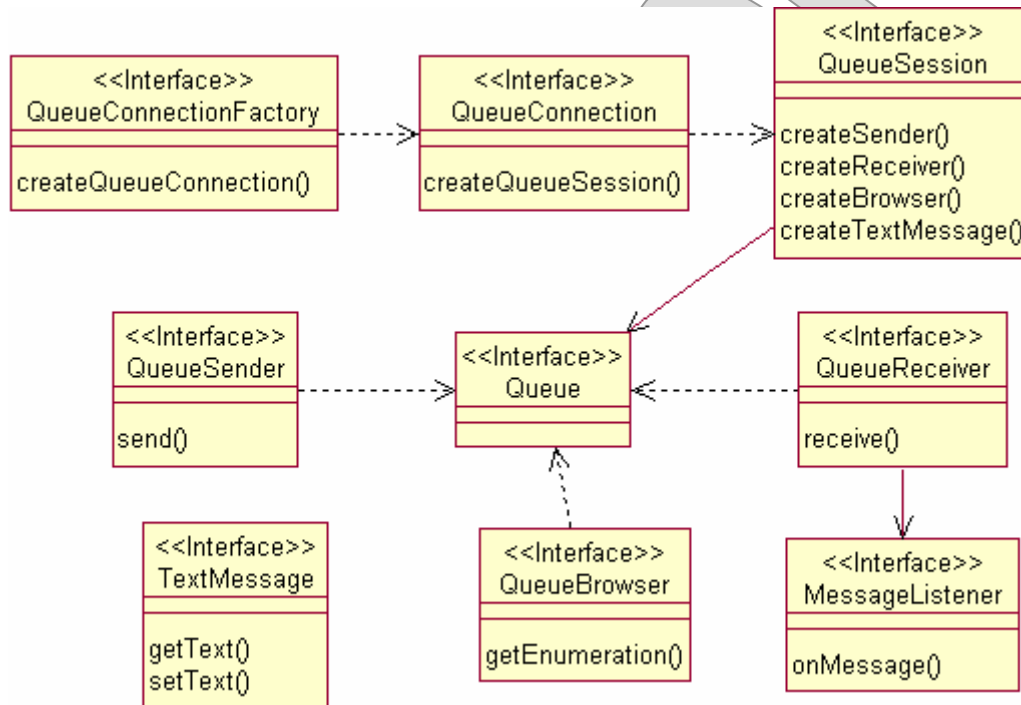
    public Message createMessage ();
    public MessageProducer createProducer
        (Destination dest);
    public MessageConsumer createConsumer
        (Destination dest, MessageSelector selector);

    public void recover ();
    public void commit ();
    public void rollback ();
}
```

- Much of its apparent function as an interface is to provide factories for other JMS objects.
- But then these objects are **bound** to the session, and therefore will participate in certain scenarios that are essential to JMS' guarantees of reliability, such as message acknowledgement, redelivery, and transactions.
- The **Session** interface also defines key constants, including:
  - The possible **acknowledgement modes**
  - The ability to make a session **transacted**
- We'll study these features in a later chapter.

## Working with Queues

- A complete model of interface subtypes is dedicated to point-to-point messaging; so that we can focus on concrete usage, let's start thinking in terms of these specific types:

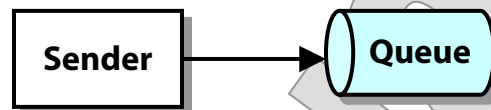


- We'll also focus on **TextMessages**, though there are various **Message** subtypes as well (which we'll consider later).
- There is only one way to send to a queue, but there are actually three distinct means of consuming messages:
  - A **QueueReceiver** can **receive** any messages on the queue – this is known as a **pull model** because the client takes the action.
  - It can be connected to a **MessageListener** for a **push** approach by which the service notifies the listener as messages arrive.
  - A **QueueBrowser** can see what's on the queue without actually consuming messages; browsed messages are not removed.

## Sending to a Queue

**EXAMPLE**

- Let's try a few simple exercises in sending to and receiving from JMS queues.
- In **Examples/Queue/Step1** are several standalone-application classes, each of which interacts with a JMS queue in some way.
- First we'll look at **cc.jms.Sender**, which sends a simple text message to a queue called "Queue."



- The **main** method accepts a command-line argument for the message text, or uses a default, and calls **write**:

```
public static void main (String[] args)
    throws Exception
{
    new Sender ().write (args.length == 0
        ? "Testing: 1, 2, ..."
        : args[0]);
}
```

## Sending to a Queue

**EXAMPLE**

- **write** starts by looking up two objects via JNDI: a **Queue**, and a **QueueConnectionFactory**.

```
public void write (String text)
    throws Exception
{
    QueueConnectionFactory factory = null;
    QueueConnection connection = null;
    QueueSession session = null;
    Queue queue = null;

    Context context = null;
    try
    {
        context = new InitialContext ();
        factory = (QueueConnectionFactory)
            context.lookup ("jms/QueueFactory");
        queue = (Queue) context.lookup ("jms/Queue");
    }
    finally
    {
        if (context != null)
            try { context.close (); }
            catch (Exception ex) {}
    }
    ...
}
```

## Sending to a Queue

**EXAMPLE**

- It uses the connection factory to get the ball rolling:
  - Create a **Connection**
  - Create a **Session**
  - Create a **QueueSender**
  - Create a **TextMessage**
  - Set the message text and **send** the message

```
try
{
    connection = factory.createQueueConnection ();
    session = connection.createQueueSession
        (false, Session.AUTO_ACKNOWLEDGE);
    QueueSender sender =
        session.createSender (queue);

    TextMessage message =
        session.createTextMessage ();
    message.setText (text);
    sender.send (message);

    System.out.println ("Sent message to queue " +
        queue.getQueueName () + ":");
    System.out.println (message);
}
finally
{
    if (connection != null)
        try { connection.close (); }
        catch (Exception ex) {}
}
}
```

## Sending to a Queue

**EXAMPLE**

- Try building and running this application, using the prepared script **Send**:

**ant****Send**

```
Exception in thread "Main Thread"  
java.lang.reflect.InvocationTargetException  
...  
Caused by: javax.naming.NameNotFoundException:  
While trying to lookup 'jms.QueueFactory' didn't  
find subcontext 'jms' ...
```

- We tried to send a message to a queue... but what queue? How do we know it exists?
- It doesn't! We have to create it. Use the prepared script **CreateQueue** to do this – it invokes Ant targets which in turn use this application server's administration tools to create a queue and a queue connection factory.

**CreateQueue Queue**

```
Connecting to t3://localhost:8080 with userid admin  
...  
Bean type JMSServer with name QueueServer has been  
created successfully.  
MBean type JMSSystemResource with name  
QueueSystemResource has been created successfully.
```

## Sending to a Queue

**EXAMPLE**

- If you open a separate console in **Examples/JNDIReport**, you can see the published names of these JMS objects:

### Report

```
...
jms
  Queue
  QueueFactory
  ...
```

- **Sending should now work:**

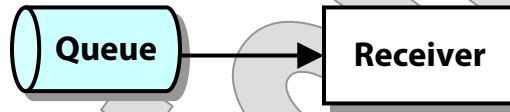
### Send

```
Sent message to queue QueueSystemResource!Queue:
TextMessage[ID:<261630.1234297711016.0>,
Testing: 1, 2, ...]
```

## Receiving from a Queue

**EXAMPLE**

- The class `cc.jms.Receiver` will check the queue for any outstanding messages, and read them off the queue:



- The source code is organized in a way very similar to **Sender**, with a **main** method that invokes a helper – this time it's called **read** – and the helper starting off with JNDI lookups.
- Then it moves to read any and all messages from the queue:

```
connection = factory.createQueueConnection ();
session = connection.createQueueSession
    (false, Session.AUTO_ACKNOWLEDGE);
connection.start ();
QueueReceiver receiver =
    session.createReceiver (queue);

System.out.println
    ("Receiving message(s) from queue " +
    queue.getQueueName () + ":");
Message message = null;
while ((message = receiver.receive (1000)) != null)
    System.out.println (message);
```

## Receiving from a Queue

**EXAMPLE**

- There are three methods that receive queued messages.
- All will return immediately if a message is available at the time of the call.

```
public Message receive ();
```

- This will block indefinitely waiting for a message to be available.

```
public Message receive (long timeoutMilliseconds);
```

- This will block and wait for a maximum time, then return empty-handed.

```
public Message receiveNowait ();
```

- This will check and return immediately.

- Each has different performance characteristics.
- We're using the timeout-based method, and this is the most common practice.
- In any case, we typically call the receiving method in a loop:
  - We read messages until no more are available.
  - Or, we read forever, until the thread is externally stopped. This "polling" approach is not favored, though – we'll see a better way in a moment.

## Receiving from a Queue

**EXAMPLE**

- Test using the script **Receive**:

### Receive

```
Receiving message(s) from queue
      QueueSystemResource!Queue:
TextMessage[ID:<261630.1234297711016.0>,
      Testing: 1, 2, ...]
```

- Try again, and of course the queue is exhausted:

### Receive

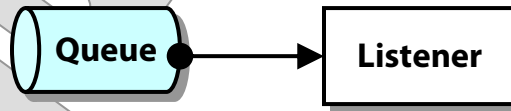
```
Receiving message(s) from queue
      QueueSystemResource!Queue:
```

- Try queuing a few messages in a row with **Send**, and you'll see that a single call to **Receive** brings them all down in one shot.

## Listening on a Queue

**EXAMPLE**

- The previous example illustrates “pull” consumption, in which the client actively asks for any available messages.
- For applications that want to receive messages whenever they’re available, the pull model is not so good.
  - It forces the application to loop indefinitely, which is inefficient, even if the programmer is careful to sleep the listening thread periodically.
- A “push” model is available: we implement the **MessageListener** interface and make that listener class available for callbacks by the JMS server:



- The **cc.jms.Listener** class does just this.

## Listening on a Queue

**EXAMPLE**

- The logic is a bit more complex.
  - First, in **listen**, we do the usual JNDI lookup; create our connection, session, and receiver; and **start** the connection.
  - But then, instead of calling **receive**, we call **setMessageListener** to make our listener method available to JMS:

```
receiver.setMessageListener (this);
```

- We then cause the main thread to enter a wait loop, so that our application won't just close up shop immediately:

```
while (running)
{
    synchronized (this) { wait (); }

    try { Thread.sleep (500); }
    catch (InterruptedException ex) {}
}
```

- When we are notified to shut down, we close the connection and let the thread die:

```
if (connection != null)
    try { connection.close (); }
    catch (Exception ex) {}
```

## Listening on a Queue

**EXAMPLE**

- So, whither our logic for processing messages?
- We've implemented **MessageListener.onMessage**:

```
public void onMessage (Message message)
{
    try
    {
        System.out.println (message);
        if (message instanceof TextMessage &&
            ((TextMessage) message).getText ()
                .equalsIgnoreCase ("SHUTDOWN"))
            synchronized (Listener.this)
            {
                running = false;
                notify ();
            }
    }
    catch (Exception ex)
    {
        ex.printStackTrace ();
        System.exit (-1);
    }
}
```

- So, until we get a message with the text “SHUTDOWN”, we keep processing; and when we get the cue, we shut down gracefully.

## Listening on a Queue

**EXAMPLE**

- To test the listener, start by **Sending** a couple messages:

**Send One**

**Send Two**

- Now open a separate console and run the listener:

**Listen**

```
Listening for messages from queue
                               QueueSystemResource!Queue:
  (Send "SHUTDOWN" message to shut down.)
TextMessage[ ID:<261630.1234297924875.0>, One]
TextMessage[ ID:<261630.1234297931422.0>, Two]
```

- So we see that a listener will consume any messages outstanding on the queue; no need to explicitly **receive** those first.

- Now **Send** another two messages:

**Send Three**

**Send SHUTDOWN**

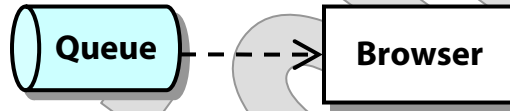
- ... and in the listening console you'll see them, and then the listener will shut down as directed:

```
TextMessage[ ID:<261630.1234298037641.0>, Three]
TextMessage[ ID:<261630.1234298045984.0>, SHUTDOWN]
```

## Browsing a Queue

---

- Finally, it is possible to read messages from a queue without consuming them.



- Among other things, this makes it possible for queued messages to be read by **multiple parties**.
  - This isn't the primary purpose of queuing, of course, and the idea of browsing seems to be more in line with **publish-and-subscribe** messaging.
  - But each messaging style has a primary usage and then offers some variations, many of which seem to overlap with the other.
  - Publish-and-subscribe offers **durable subscribers**, which sound a lot like queue listeners.
- You'll see some browsing in the upcoming labs.
  - So we have three ways to get messages:
    - **Pulling** them via a **QueueReceiver**, which consumes the messages, and reads only what's available at the time of the pull.
    - Having them **pushed** to us via a **MessageListener**, which consumes the messages and reads past messages and anything that comes up as long as the listener is active.
    - **Browsing** them, which is like the pull method but does not consume the messages.

## A Point-to-Point Utility

**LAB 1A**

### Suggested time: 30 minutes

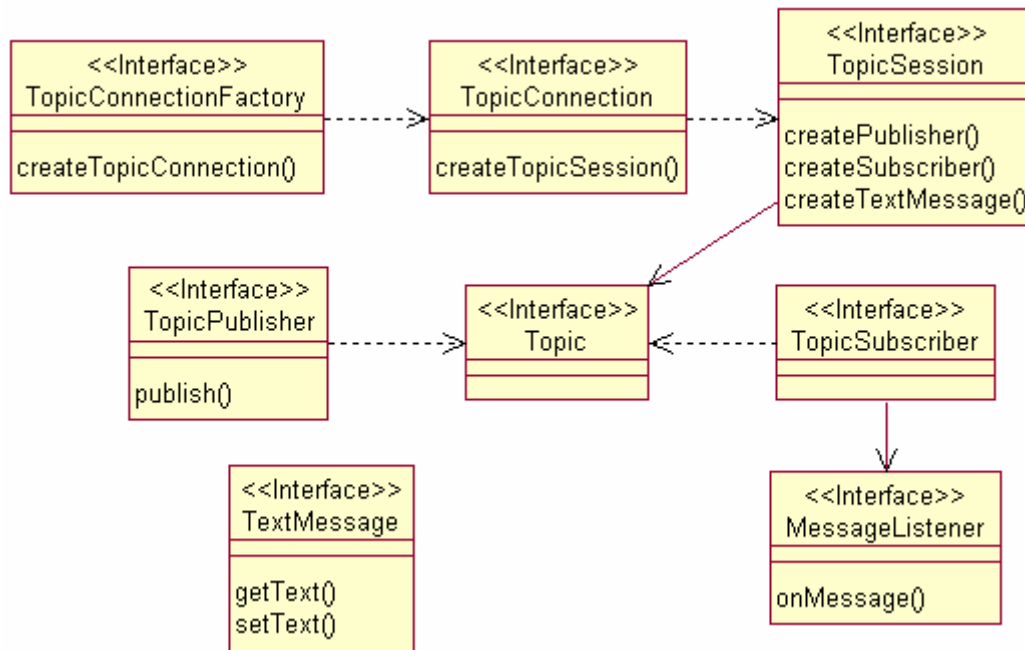
In this lab you will refactor common JMS client tasks into a reusable utility called the **PointToPointClient**. You may have noticed in the previous examples that a great deal of JMS code is repetitive – which means boring, and time-consuming, and especially error-prone. Let's factor things like JNDI lookup and creation of connections and sessions into a utility.

You will be instructed to carry out some of the first and most obvious refactoring tasks. This should be a useful exercise and help you get more familiar with JMS API details. However it's probably not worth the lab time to nail down every last bit of this utility class; at a certain point we break off from detailed instructions, and you're free to go as far along with the rest of the task as you like, or simply review the final utility class, which we will use in upcoming labs to simplify work there.

Detailed instructions are found at the end of the chapter.

## Working with Topics

- Much of the model for topics and the publish-and-subscribe style is the same as it is for queues – even if everything seems to have a new name:

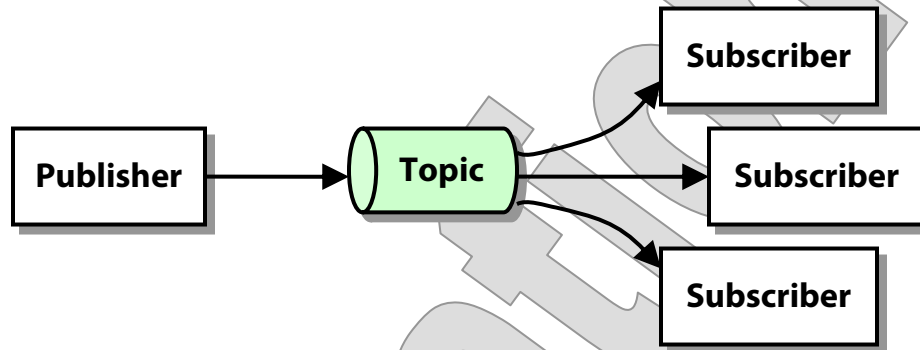


- If anything, the basic model is a little simpler, with fewer options.
- Though it's technically possible for a subscriber to actively **receive** messages, it's rarely done; a subscriber is more like a listener, participating in a push process, not a pull.
- The big differences with publish-and-subscribe are in the resulting behavior, not so much in the API.

## Publishing and Subscribing

**EXAMPLE**

- In **Examples/Topic** there are classes roughly analogous to those in the previous set of queue-based examples:



- **cc.jms.Publisher** publishes text messages to a topic called “Topic”. It is closely analogous to the queue **Sender**.
  - **cc.jms.Subscriber** subscribes to that topic – it is very like the queue **Listener**.
  - They both take advantage of a utility **PublishAndSubscribeClient**, which is the somewhat under-achieving cousin of the utility you created in the previous lab.
- **The code for the classes is so similar we don’t really need to review it.**
    - Maybe 95% of this code was created by performing global replacements of text tokens from the queue example; that’s how tight the naming conventions are in JMS.

## Publishing and Subscribing

**EXAMPLE**

- What is more interesting is how these applications behave.
- Try testing more or less as we did with **Sender** and **Listener**: start by publishing two messages.

```
CreateTopic Topic
ant
Publish One
Publish Two
```

- Now run the subscriber in a separate console:

```
Subscribe
Receiving messages from topic
TopicSystemResource!Topic:
(Send "SHUTDOWN" message to shut down.)
```

- Immediately we see a difference: published messages on a topic are not durable.
- Another way of saying this is a subscriber must be active at the time of publishing, or it won't receive the message – ever.

- Now publish a third message:

```
Publish Three
```

- ... and you'll see that it is received by the subscriber immediately:

```
TextMessage[ ID: <261630.1234298261969.0> , Three ]
```

## Publishing and Subscribing

**EXAMPLE**

- So we've seen one major distinction between queues and topics, which is a matter of message durability.
- The other big difference has to do with broadcasting.
- Open a third console and run a second subscriber:

**Subscribe**

- Publish another message, and see that both subscribers get it.

**Publish Four**

- Publish the shutdown message and see that both subscribers receive the message, and both do shut down.

**Publish SHUTDOWN**

- What would happen if you did this with senders, queues and listeners?
  - In theory the results are undefined; perhaps which listener got any given message would be a random matter of thread scheduling.
  - In practice, WebLogic handles this pretty gracefully: it simply distributes incoming messages to each listener in turn.
  - Still, this means that no one listener will get all the messages sent to a given queue; this is rarely the behavior one is seeking.

## The WebLogic Administration Console

---

- Like most Java EE servers, WebLogic provides a web-based console application that allows administrators to manage domains and servers interactively.
  - One server in any WebLogic domain is designated as the administrative server.
  - That server hosts the console application and implements administrative changes for the whole domain, and possibly for individual servers including itself.
- All of our administrative tasks are scripted, so that we can stay focused on development concepts and skills.
  - We deploy and undeploy applications using an Ant task.
  - We create JMS resources using scripting.
- Still, the console application can be illustrative.
- We'll fire it up now and then, to get a closer look at the server configuration that accompanies our application development.

## The WebLogic Administration Console

---

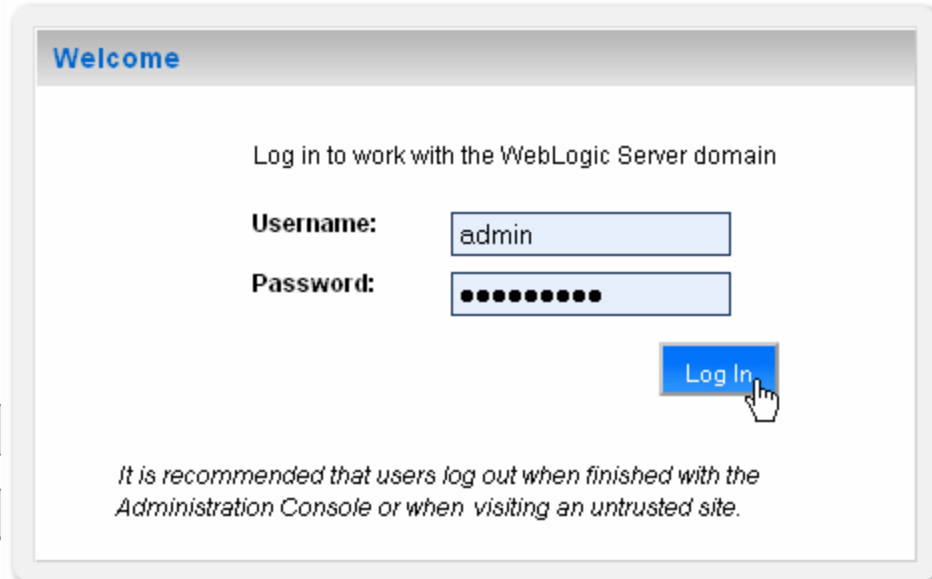
- Visit the console now at the following URL:

`http://localhost:8080/console`

### Deploying application for /console.....

This application is deployed on the first access. You can change this application to instead deploy during startup.

Refer to instructions in the On-Demand Deployment documentation.



Welcome

Log in to work with the WebLogic Server domain

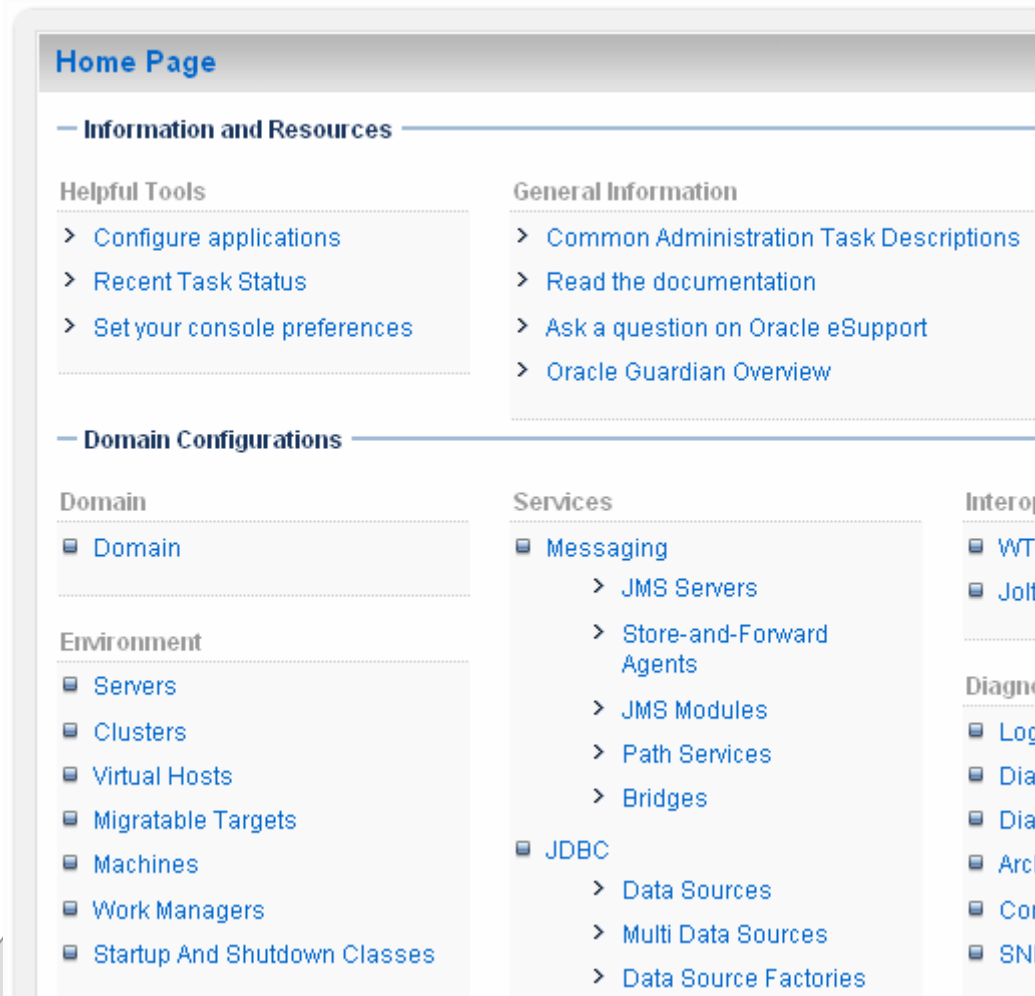
Username:

Password:

*It is recommended that users log out when finished with the Administration Console or when visiting an untrusted site.*

- Login using the prepared admin account, **admin/ccstudent**.

## The WebLogic Administration Console



- There's far too much to show on the page, and far too many functions available in the console for us to visit it all.
- And, ultimately, this isn't a course in server administration.

## JMS Configuration

---

- But let's see how JMS resources are configured:
  - Under “Services” and “Messaging,” click the JMS Modules link:



- Here are the two JMS modules we've created so far – one queue and one topic:

**JMS Modules**

New Delete Showing 1 to 2 of 2 Previous | Next

<input type="checkbox"/>	Name	Type
<input type="checkbox"/>	QueueSystemResource	System
<input type="checkbox"/>	TopicSystemResource	System

New Delete Showing 1 to 2 of 2 Previous | Next

## JMS Configuration

- Click the QueueSystemResource module and see the resources configured within it:

**Summary of Resources**

New Delete Showing 1 to 3 of 3 Previous | Next

<input type="checkbox"/>	Name	Type	JNDI Name	Subdeployment	Targets
<input type="checkbox"/>	Queue	Queue	jms.Queue	DeployToJMSServer	QueueServer
<input type="checkbox"/>	QueueFactory	Connection Factory	jms.QueueFactory	DeployToJMSServer	QueueServer
<input type="checkbox"/>	QueueTemplate	Template	N/A	N/A	N/A

New Delete Showing 1 to 3 of 3 Previous | Next

- Notice that two of these are JNDI-named objects (which we've already observed through the JNDIReport application), and these are the two that we look up in our **PointToPointClient**, or really in any JMS client code.
  - The third is a WebLogic-specific object that our administration script happens to create; we don't use JMS templates in this course.
  - You may want to click on the Queue and QueueFactory objects, and poke around their settings a bit; you'll see various configuration options that relate to topics found later in this course.
- Click the Log Out link near the top of the page, and close the browser.

## Expiration

---

- Among several attributes that can be assigned to JMS messages, we'll consider **expiration** now – in advance of the next chapter, which will describe them all.
- You can tell a sender or publisher to set a finite **time to live** on a message, in the act of sending or publishing.

– We've been using the simple versions of production methods:

```
QueueSender.send (Message msg);  
TopicPublisher.publish (Message msg);
```

– These just delegate to other overloads that take three additional parameters:

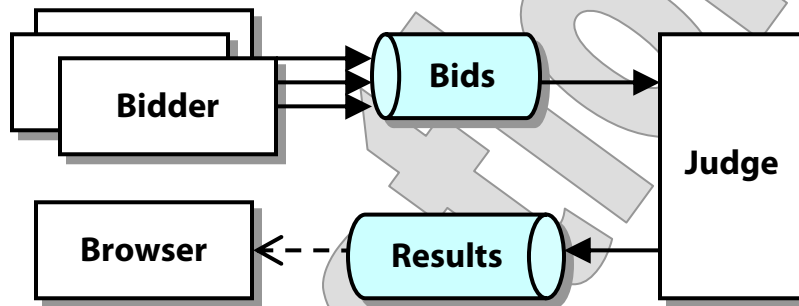
```
QueueSender.send (Message msg,  
    int deliveryMode, int priority, long timeToLive);  
TopicPublisher.publish (Message msg,  
    int deliveryMode, int priority, long timeToLive);
```

- **Delivery mode** is either persistent or non-persistent.
  - **Priority** is just what it sounds like.
  - **Time to live** tells the JMS queue to remove the message after a certain time, even if it hasn't been properly consumed. The default is zero, which means no expiration should be set.
  - For topics, the meaning here is vague at best.
- The queue sender will add the time to live to the time of sending, and stamp that into the message as its expiration.

## The Auction Case Study

---

- For several of this course's exercises, we'll work with the following model of an auction house that uses JMS queues to receive bids and to post auction results:



- The **Bidder** uses a **QueueSender** to send bids on lots available at auction, stating lot, bidder name, and price in a formatted text message.
- The **Judge** acts as a consumer of bids, using a **QueueReceiver**, and a producer of results.
- To make results available to all parties, but still enjoy the durability of a queue, the **Judge** sets the result messages to expire in a finite time. For convenience in the lab, this time will be one minute, though practically it would probably be several days or weeks.
- The **Browser** can check the results queue to see what auction lots have been judged and whose bids won the auctions.

## Judging the Auction

LAB 1B

**Suggested time: 45 minutes**

In this lab you will complete the initial implementation of the Auction case study. The **Bidder** and **Browser** clients are already implemented; you will add code to the **Judge** to read bids, judge the auction by selecting the high bidder, and post the results.

Detailed instructions are found at the end of the chapter.

Evaluation  
Only

## SUMMARY

- **The Java Message Service fills a simple need not met by other Java EE technology: to send and receive messages using disparate components that are not bound to one another by their semantics nor required to operate synchronously.**
- **Two different message styles suit different sorts of applications:**
  - Queues offer absolute reliability and deferred message processing; they are more obviously the model that motivates JMS.
  - Topics offer broadcasting ability, but typically call for immediate processing, and messages are not durable.
- **Any sort of Java component can be a JMS client.**
  - Components living in Java EE containers automatically have access to the JMS API and to the server's JNDI context, in which JMS queues, topics, and factories will be published.
  - Standalone (Java SE) applications have a harder road, but can make remote connections to a JNDI context and find JMS objects “from outside” the Java EE server.

## A Point-to-Point Utility

**LAB 1A**

In this lab you will refactor common JMS client tasks into a reusable utility called the **PointToPointClient**. You may have noticed in the previous examples that a great deal of JMS code is repetitive – which means boring, and time-consuming, and especially error-prone. Let’s factor things like JNDI lookup and creation of connections and sessions into a utility.

You will be instructed to carry out some of the first and most obvious refactoring tasks. This should be a useful exercise and help you get more familiar with JMS API details. However it’s probably not worth the lab time to nail down every last bit of this utility class; at a certain point we break off from detailed instructions, and you’re free to go as far along with the rest of the task as you like, or simply review the final utility class, which we will use in upcoming labs to simplify work there.

**Lab workspace:** Labs\Lab1A  
**Backup of starter code:** Examples\Queue\Step1  
**Answer folder(s):** Examples\Queue\Step2  
**Files:** src/cc/jms/PointToPointClient.java (to be created)  
src/cc/jms/Sender.java  
src/cc/jms/Receiver.java  
src/cc/jms/Listener.java

**Instructions:**

1. Define a new class **cc.jms.PointToPointClient**. Declare protected fields **factory**, **queue**, **connection**, and **session**, of the appropriate types for working with queues. You can run **ant** at any time to check that your classes compile.
2. Create a constructor that takes a single string as a parameter: this will be the name of a JMS queue.
3. Using the **Sender** class as a template, implement this constructor to look up the **queue** and **factory** objects. One simplifying assumption we’ll make for our utility is that there will be a queue connection factory published at a name that is the queue name plus the suffix “Factory”.
4. Create an **open** method that takes a **boolean** for transaction mode and an **int** for acknowledgement mode – the same two parameters defined on **Connection.createSession** – with **void** return type. Use the **factory** to create the **connection**, and use the **connection** to create the **session** with the given parameters. You’ll have to **catch JMSException** and close up the connection if the session can’t be created.

**A Point-to-Point Utility****LAB 1A**

5. Create a **close** method that safely closes the **connection**, taking the **try/catch** headaches off the caller's hands.
6. Now let's extend a bit more convenience to the caller. Create a method **openToSend** that calls **open** but then goes farther by creating a **QueueSender** and returning it to the caller. Again, **catch** the **JMSEException**, and call **close** in that case.
7. Create a method **openToReceive** that does the analogous things for a receiver.
8. Now you can start to take advantage of your new utility. Open the **Sender.java** source file and make it **extend PointToPointClient**. (You can use this utility via delegation, too, but inheriting it is especially nice.)
9. Give **Sender** a default constructor that calls the superclass constructor and passes the name "Queue".
10. Now, in **write**, you can remove a bunch of code: the four declarations at the beginning; the entire **try/catch** system that does the JNDI lookup; and the first three assignments in the next **try** block. Just replace this last bit with a single line of code like this:

```
QueueSender sender = openToSend ();
```

11. Rip out all the code from the **finally** block and replace with a call to **close**.
12. You should be able to use the **Sender** application now, just as you did before – but of course with less code, the reusable stuff now in a reusable class.
13. Do the same refactoring for **Receiver** and **Listener**.

**Optional Steps** – carry out as many or as few of these as you like, but in any case take a moment to review the final version in **Step2**.

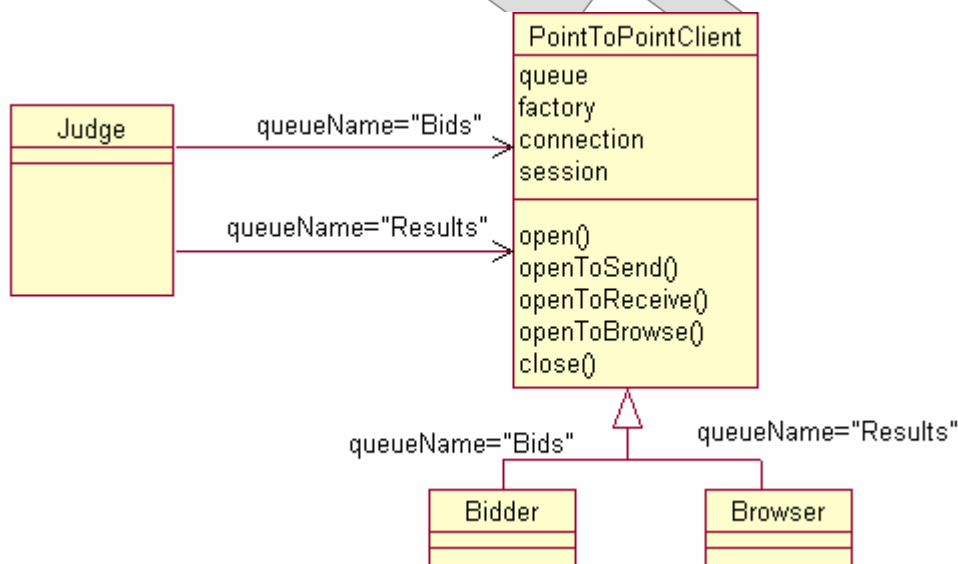
14. You might want to enhance **PointToPointClient** with overloads of **open**, **openToSend**, and **openToReceive** that take no arguments and use the default values of **false** for transacted and **Session.AUTO\_ACKNOWLEDGE** for acknowledgement mode.
15. Create accessors for all four protected fields.
16. In **open** and **close**, check that the connection is in the right state coming in. That is, don't let a caller **open** twice in a row; if **connection** is not **null**, throw an **IllegalStateException**; and in **close** throw the exception if the connection is already **null**, and be sure to set it to **null** after closing.
17. Create a method **openToBrowse**.

## Judging the Auction

LAB 1B

In this lab you will complete the initial implementation of the Auction case study. The **Bidder** and **Browser** clients are already implemented; you will add code to the **Judge** to read bids, judge the auction by selecting the high bidder, and post the results.

You'll see that the first two of those classes extend the **PointToPointClient** from the previous lab. The **Judge** has to work a little differently, because it can't be dedicated to just one queue. Fortunately the utility you built can be used almost as easily as a delegate, and so the **Judge** will create two **PointToPointClient** instances, one to act as its proxy for each of the bid and result queues:



### Lab workspace:

Labs\Lab1B

### Backup of starter code:

Examples/Auction/Step1

### Answer folder(s):

Examples/Auction/Step2

### Files:

src/cc/auction/Judge.java

### Instructions:

1. Review the **Bidder** class, which you'll see is quite similar to the **Sender** from the example, except that it has a convenience method that sends three distinct messages to the bids queue.
2. Review the **Browser** class, and here you'll see a concrete example of queue browsing.

**Judging the Auction****LAB 1B**

3. Build and test the **Bidder** class, also creating the necessary queues for the application:

```
CreateQueue Auction_Bids
CreateQueue Auction_Results
ant
Bid
Sent three bids.
```

4. Open the **Judge.java** source file, and see that there is a **judge** method in need of implementation; also a **main** method that calls it and a **parseBid** helper method that knows how to break the text messages from the bids queue into objects of type **Bid**, which is a simple **JavaBean**.
5. To begin your implementation of **judge**, create a new **PointToPointClient** called **receiving**, and initialize it by passing the name “Auction\_Bids” to the constructor.
6. Declare a **Bid** reference **winningBid**, initialized to **null**.
7. Using **Examples/Queue/Step2/src/cc/jms/Receiver.java** as a template for your code: enter a **try/catch/finally** system; create a queue receiver; enter a loop that reads all available messages; **catch** and log any exceptions to the console, and **finally** call **close** on the **receiving** delegate. (Notice that you’ll have to call **receiving.openToReceive**, not just **openToReceive**, since this class doesn’t extend **PointToPointClient**.)
8. Inside the loop, call **parseBid** to get a **Bid** object that represents the message content.
9. Optionally, dump this content to the console for your own testing purposes.
10. If this bid has a higher price than the current **winningBid**, or if **winningBid** is **null**, set **winningBid** to refer to this bid.
11. After the **try/catch/finally**, if **winningBid** is **null**, you didn’t get any messages, or there were errors processing them. Print a message and shut down the application.
12. Otherwise, it’s time to send a message to the second queue. Create another **PointToPointClient** instance, this one called **sending** and dedicated to the queue “Auction\_Results”.
13. In a **try** block, create a queue sender.
14. Create a text message that expresses who won the auction; the exact format doesn’t matter in this case.
15. Send the message, and set a time to live of one minute while you’re at it:

```
sender.send (myMessage, Message.DEFAULT_DELIVERY_MODE,
            Message.DEFAULT_PRIORITY, 60000);
```

16. Again, **catch** and log any exceptions, and **finally** be sure to **close** your delegate.

**Judging the Auction****LAB 1B**

17. Build again, and test your new class. You should see output similar to the following, depending of course on how you chose to write information to the console:

**ant**

**Judge**

Checking for bids ...

Bid:

Who: Will

Lot: 49

Bid: 100

Bid:

Who: Leah

Lot: 49

Bid: 110

Bid:

Who: Christy

Lot: 49

Bid: 50

**Check**

Returning winning bid:

Who: Leah

Lot: 49

Bid: 110