



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

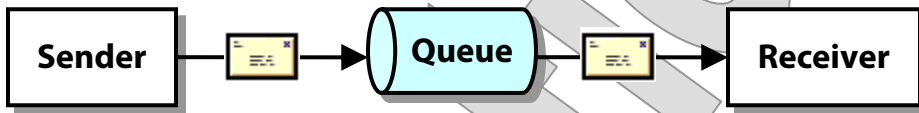
- The Java 2 Enterprise platform includes the **Java Message Service**, or **JMS** – at version 1.1 as of this writing.
- JMS defines the message **destination**, which acts as an intermediary between the **producer** and **consumer** of a message.



- 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

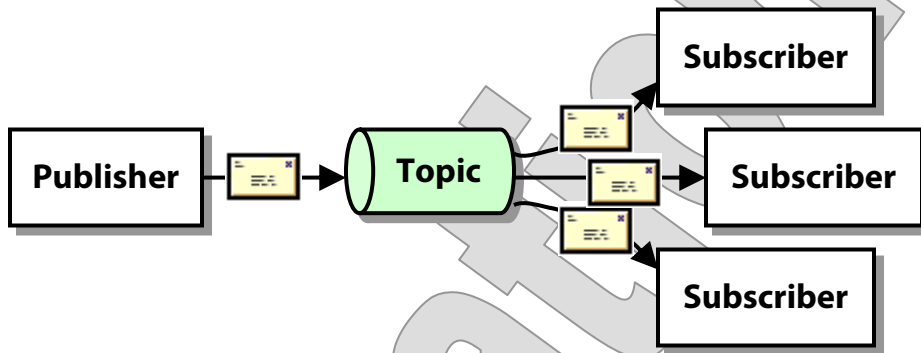
- JMS offers two distinct messaging styles: **point-to-point** and **publish-and-subscribe**.
- The point-to-point model is the more intuitive, and the more commonly used:



- 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

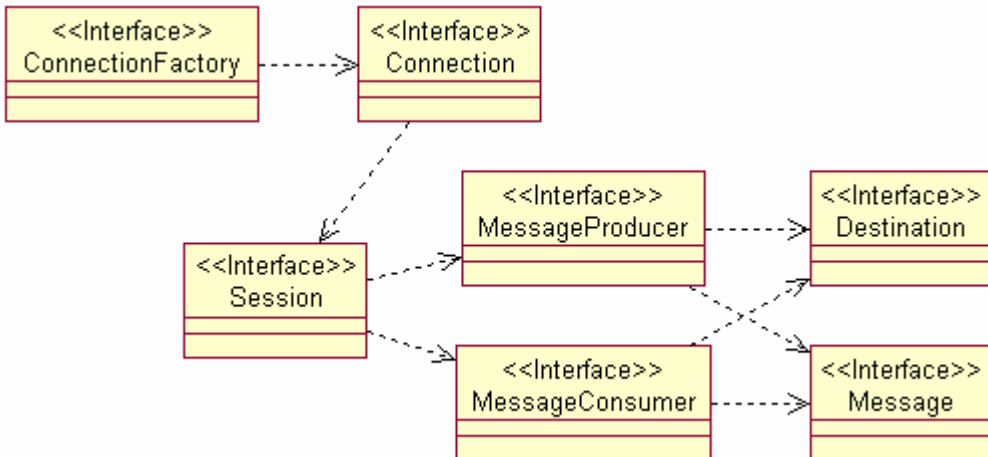
- In the publish-and-subscribe style, multiple parties can receive a message from the JMS destination.



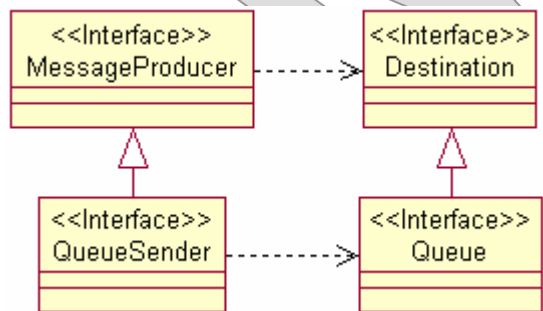
- The producer is called a **publisher**.
 - The destination is called a **topic**.
 - Consumers are called **subscribers**, and (in the simplest usage anyway) they must be **active** and listening for messages at the time they are published, in order to receive them.
- **Publish-and-subscribe offers 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 and Environment Setup

- For our hands-on exercises in JMS, we'll rely on the message-queue service bundled with the **J2EE 1.4 SDK**.
 - This SDK includes a **Java standard SDK**, version 5.0.
 - It also includes the **Ant** make utility, and custom targets for server administration that can be included by invoking the tool with the prepared script **asant**.
- These tools have already been installed on your systems.
- For the Ant targets to work, assure that certain environment settings are accurate:
 - An environment variable **CC_MODULE** must be set to the root directory of the lab installation; this is typically

```
CC_MODULE=c:\Capstone\JMS
```

- The executable path must include the Java SDK's **bin** directory and the J2EE SDK's **bin** directory. A typical path for Windows systems will be:

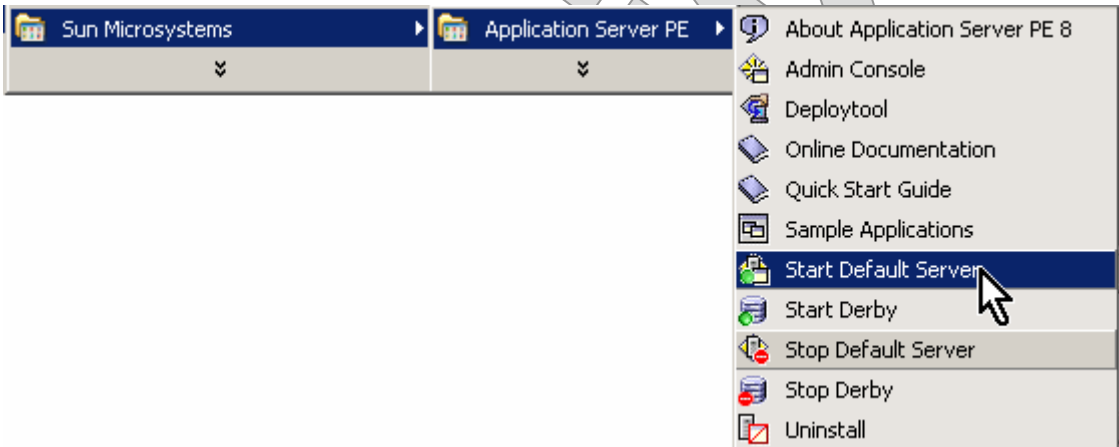
```
PATH=c:\Windows;c:\Windows\System32;  
c:\Sun\AppServer\jdk\bin;c:\Sun\AppServer\bin
```

- Standard environment variables for Java and J2EE must be set:

```
JAVA_HOME=c:\Sun\AppServer\jdk  
J2EE_HOME=c:\Sun\AppServer
```

Starting and Stopping the Server

- On Windows, a program menu is created as part of the J2EE 1.4 SDK installation, with some of the most useful tasks.
- To start the application server, choose **Start Default Server** from this menu as shown below:



- A console appears to inform you that the server is starting, then has started. Hit a key to clear this console.
- You won't see the server process on the desktop anywhere.
- You can leave this running, for the most part, throughout the course; when you want to shut down, just choose the **Stop Default Server** item from the same menu.

Ant Tasks

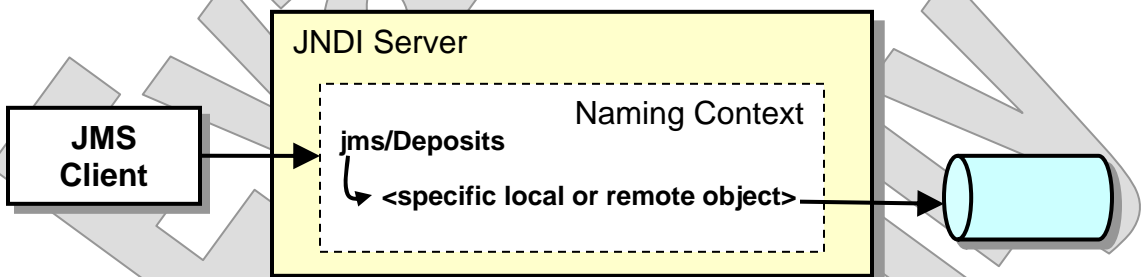
- To simplify repetitive tasks for our exercises, we take advantage of prepared Ant build files.
- We shouldn't dive into every detail of the structure of these Ant files, but there are a couple that deserve some attention.
- The target **build-appclient** creates a JAR file containing:
 - All the compiled **.class files**
 - A deployment descriptor **application-client.xml**, necessary to announce the JAR to the J2EE 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

- The target **run-appclient** launches the ACC and asks it to run a specific JAR.
- The upshot is that building with **ant** and then running one of the prepared scripts to invoke the **run-appclient** target 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 details to carry out exercises in this course.
- But, if you're curious, you might want to review the files in **c:\Capstone\JMS\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 J2EE 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.

- 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.

- 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 J2EE server's **application-client container (ACC)**.
 - This gives it full visibility to the JNDI service, without having to make a TCP connection from outside the server process.

asant

Report

jdbc

__TimerPool

__TimerPool__pm

UserTransaction

ejb

mgmt

MEJB

__SYSTEM

pools

__TimerPool

descriptors

__xa

jmsra

- The exact output for your machine may vary slightly, as any configured resource (JDBC, JMS, etc.) or deployed application (servlets, JSPs, EJB, etc.) will appear here in some form.

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;

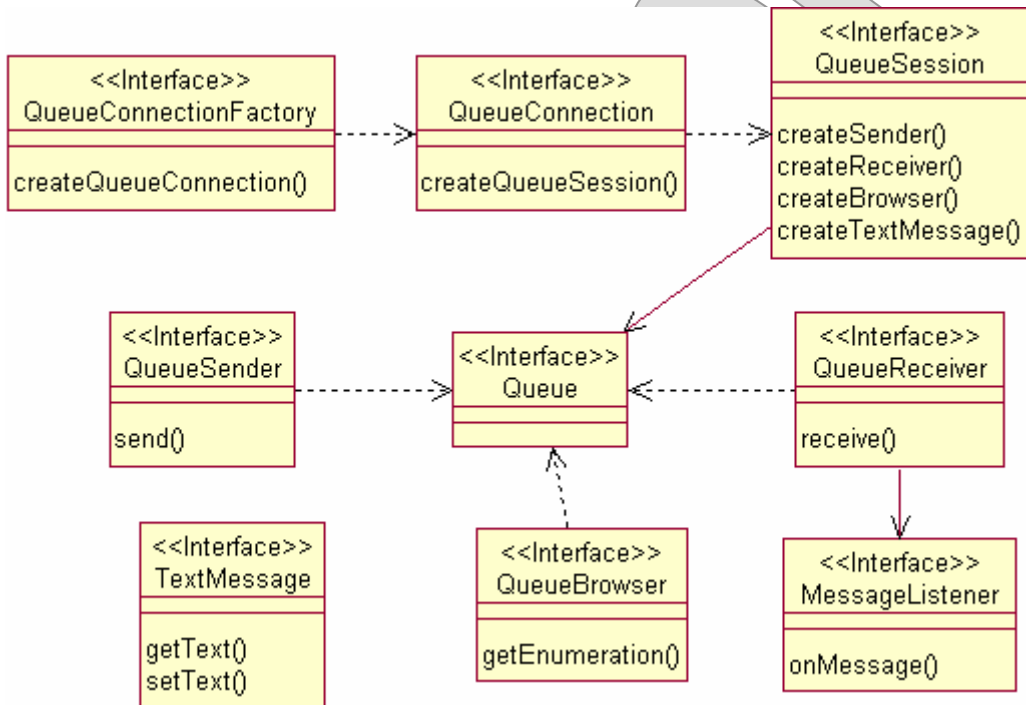
    public Message createMessage ();
    public MessageProducer createProducer
        (Destination dest);
    public MessageProducer createConsumer
        (Destination dest, MessageSelector selector);

    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:

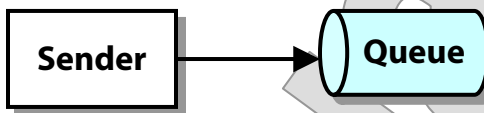


- We'll also focus on **TextMessages**, though there are various **Message** subtypes as well (which we'll consider later).
- There is only one way to send to a queue, but there are actually three distinct means of consuming messages:
 - A **QueueReceiver** can **receive** any messages on the queue – 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]);
}
```

- **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) {}
    }
    ...
}
```

- 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) {}
}
}
```

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

asant

Send

```
com.sun.enterprise.appclient.Main <init>
WARNING: ACC003: Application threw an exception.
javax.naming.NameNotFoundException: QueueFactory
not found
```

- 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

```
create-jms-resource --interactive=true
--secure=false --passwordfile
C:\Capstone\JMS\Ant\AS8.2.password --terse=false --
user admin --enabled=true --target server --host
localhost --echo=true --port 4848 --restype
javax.jms.QueueConnectionFactory jms/QueueFactory
Command create-jms-resource executed successfully.
create-jms-resource --interactive=true --
secure=false --passwordfile
C:\Capstone\JMS\Ant\AS8.2.password --terse=false -
-user admin --enabled=true --target server --
property imqDestinationName=Queue --host localhost
--echo=true --port 4848
--restype javax.jms.Queue jms/Queue
Command create-jms-resource executed successfully.
```

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

Report

```
...
jms
  Queue
  QueueFactory
__SYSTEM
pools
  ...
  jms
    QueueFactory
...
```

- **Sending should now work:**

Send

Sent message to queue Queue:

```
Text:           Testing: 1, 2, ...
Class:          com.sun...TextMessageImpl

getJMSMessageID():      ID:...-2177-1163 794455328

getJMSTimestamp():     1163794455328
getJMSCorrelationID(): null
JMSReplyTo:           null
JMSDestination:       Queue
getJMSDeliveryMode(): PERSISTENT
getJMSRedelivered():  false
getJMSType():          null
getJMSExpiration():   0
getJMSPriority():      4
Properties:            null
```

- 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);
```

- 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.

- Build and test, using the script **Receive**:

Receive

Receiving message(s) from queue Queue:

```
Text:           Testing:  1, 2, ...
Class:          com.sun...TextMess

getJMSMessageID():      ID:...-2177-1163

getJMSTimestamp():     1163794455328
getJMSCorrelationID(): null
JMSReplyTo:           null
JMSDestination:       Queue
getJMSDeliveryMode():  PERSISTENT
getJMSRedelivered():   false
getJMSType():          null
getJMSExpiration():   0
getJMSPriority():      4
Properties:            null
```

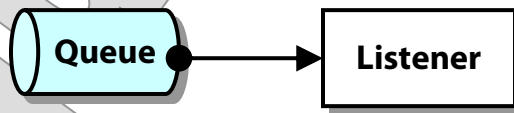
- Try again, and of course the queue is exhausted:

Receive

Receiving message(s) from queue 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.

- 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.

- 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) {}
```

- 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.

- 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 Queue:  
(Send "SHUTDOWN" message to shut down.)  
Text:      One  
...  
Text:      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:

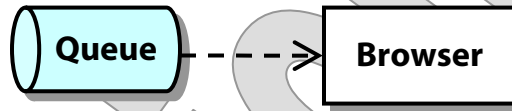
```
Send Three  
Send SHUTDOWN
```

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

```
Text:      Three  
Text:      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 queueing, 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.

Suggested time: 30 minutes

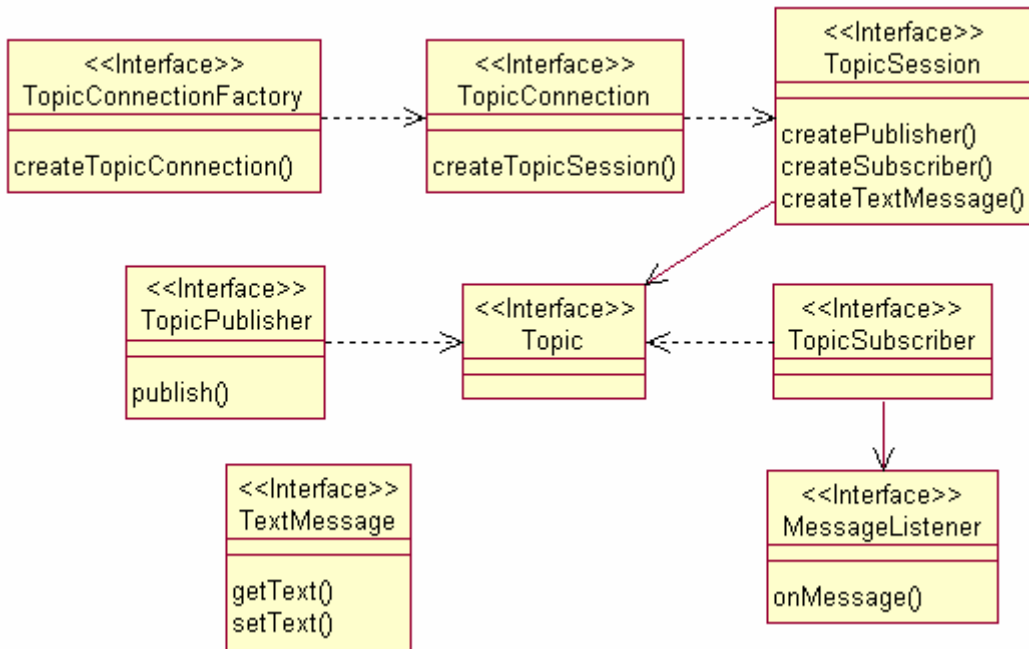
In this lab you will refactor common JMS client tasks into a reusable utility called the **PointToPointClient**. You may have noticed in the previous examples that a great deal of JMS code is repetitive – which means boring, and time-consuming, and especially error-prone. Let's factor things like JNDI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

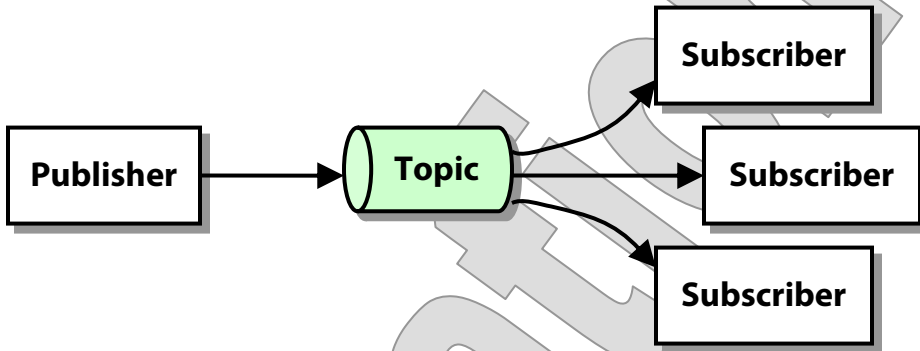
Working with Topics

- 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.

- 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.

- 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
asant
Publish One
Publish Two
```

- Now run the subscriber in a separate console:

```
Subscribe
Receiving messages from topic 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:

```
Text :      Three
... 
```

- 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?
 - Don't try it!
 - In theory the results are undefined; perhaps which listener got any given message would be a random matter of thread scheduling.
 - In practice the first listener gets all messages; latter listeners don't get any, even after the first one shuts down.
 - This makes it quite unpleasant to clean up after the experiment, because the listener process hangs, which hangs the ACC.

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);
```

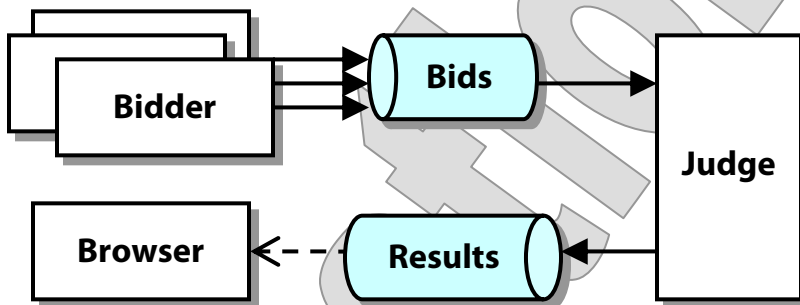
- These just delegate to other overloads that take three additional parameters:

```
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.

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.