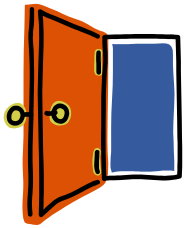
ESIR SR

Unit 10a: JGroups

François Taïani

Overview of the Session

- What is JMS
- Messages vs. RPC
- Interaction Styles
- Main JMS Classes
- Advanced Features



See lecture on indirect communication

What is JMS?

- "Java Message Service"  http://java.sun.com/products/jms/
- An **API** that is part of the J2EE standard (since Java 1.3)
  - ➔ Relevant classes and interface in javax.jms
  - ➔ Provided my most J2EE implementations (I.e. JBoss, Sun's J2EE SDK)
- **JMS** = Message Oriented Middleware (MOM) + Publish Subscribe (Pub/Sub)
  - ➔ Clients communicate via "queues" and "topics"
  - ➔ degrees of guarantees: persistence, atomicity, blocking or not
  - ➔ A central broker: possibly distributed / replicated
- Other MOM products
  - ➔ Websphere MQ (IBM), Microsoft's MSMQ and Oracle's Streams Advanced Queuing AQ, Apache ActiveMQ

Messages vs. RPC

Why uses Messages?

- Messages provides loose time & space coupling
  - ➔ In JMS "Messages" used both for MOM and Pub/Sub
  - ➔ Large scale systems: RPC often too tightly coupled
  - ➔ Asynchronous interaction possible
    - Sender up while receiver down (or disconnected) and vice-versa
- Messages are highly flexible
  - ➔ Anything can be a message in JMS: string, object, XML
  - ➔ Arbitrary interaction patterns possible
    - 1-1, n-n, with replies, with no replies
- Messages don't hide distribution
  - ➔ Sometimes this is needed to control side effects of distribution
  - ➔ Particularly true for large scale entreprise-wide systems

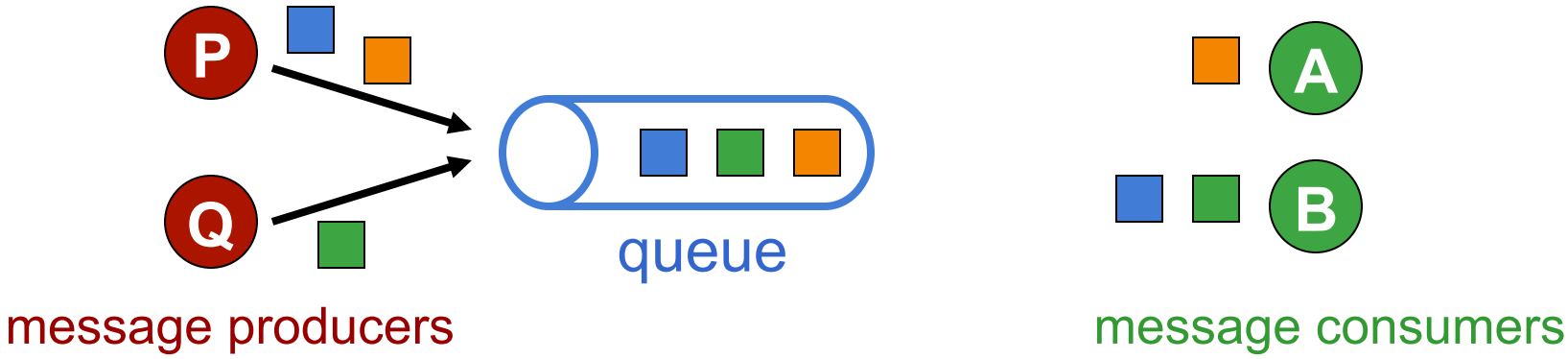
Interaction Styles in JMS

Covered in lecture on indirect communication

- Two main **modes of communication**
  - ➔ Message queuing (aka 1-to-1 communication)
  - ➔ Publish-Subscribe (aka 1-to-many communication)
- Two main **mode of message consumption** on the receiver
  - ➔ Blocking (aka synchronous, aka pull mode) with **MessageConsumer.receive( )**
  - ➔ Non-blocking (aka asynchronous, aka push mode) with **MessageListener.onMessage(..)**
- Both dimensions can be combined
  - ➔ Four possibilities
  - ➔ Usually blocking reception used with message queuing
  - ➔ And non-blocking reception used with publish-subscribe

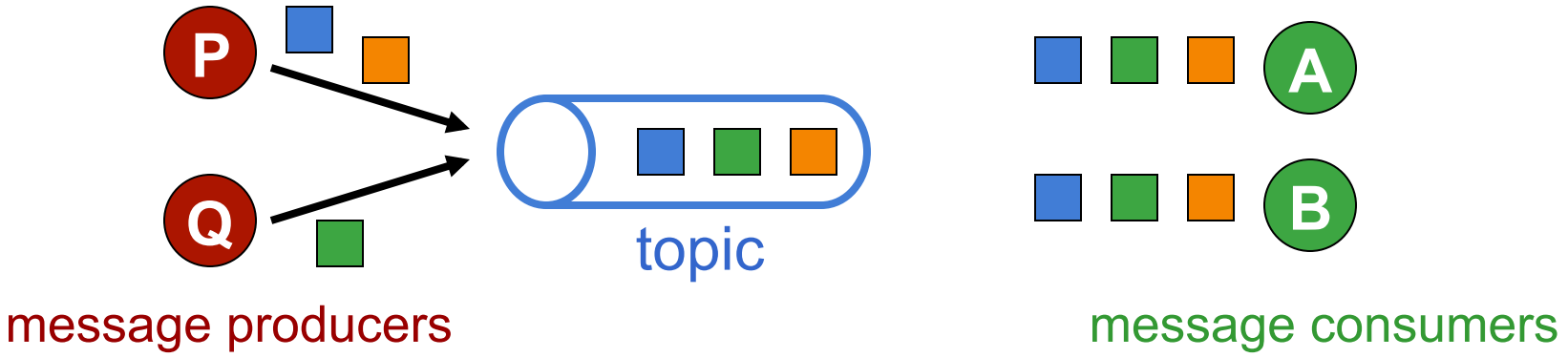
1-to-1 communication

- **Queues**:
  - Messages are sent to a queue object on the server
  - They are received from the queue by message consumers (clients)
  - **One** message can only be received by **one** clients
  - But several clients can be writing to / reading from the same queue concurrently
    - How message are dispatched is implementation dependent



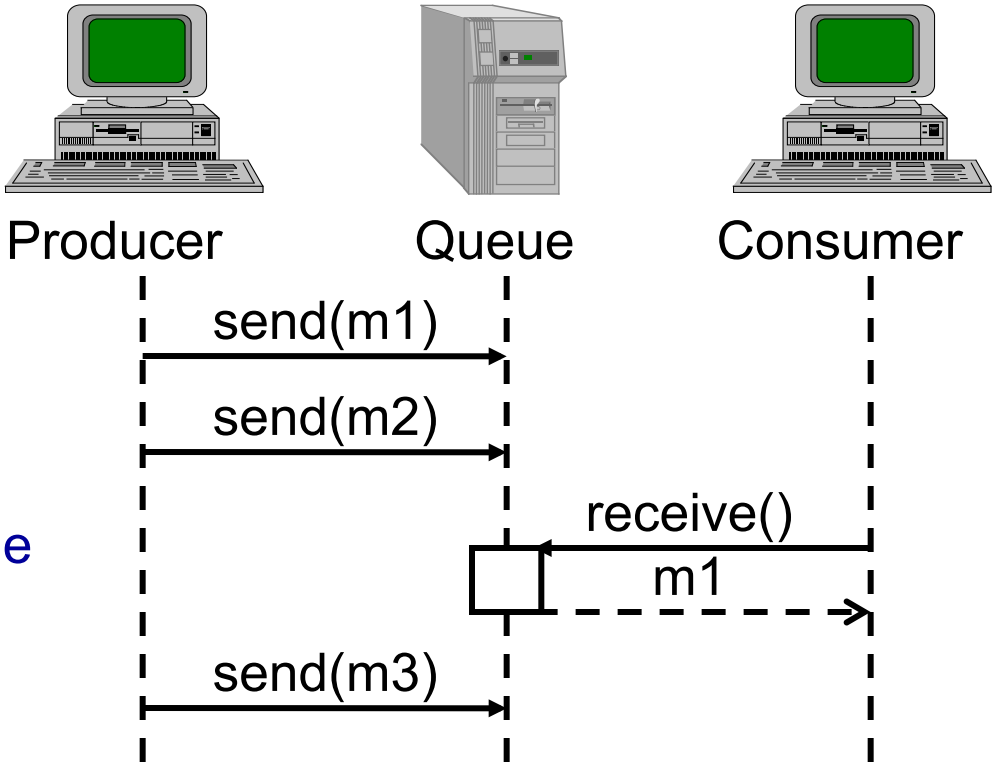
Publish / Subscribe (1-many)

- **Topics**:
  - Messages are “published” relative to a topic object on the server
  - They are received by message consumers (clients) that have subscribed to the topic
  - **One** message is received by **all subscribers**
  - If clients subscribe/ unsubscribe while messages are sent, results are undefined (they may or may not get the messages)



Blocking Reception (Pull)

- Aka Active Reception
- Typically used with point-to-point queues



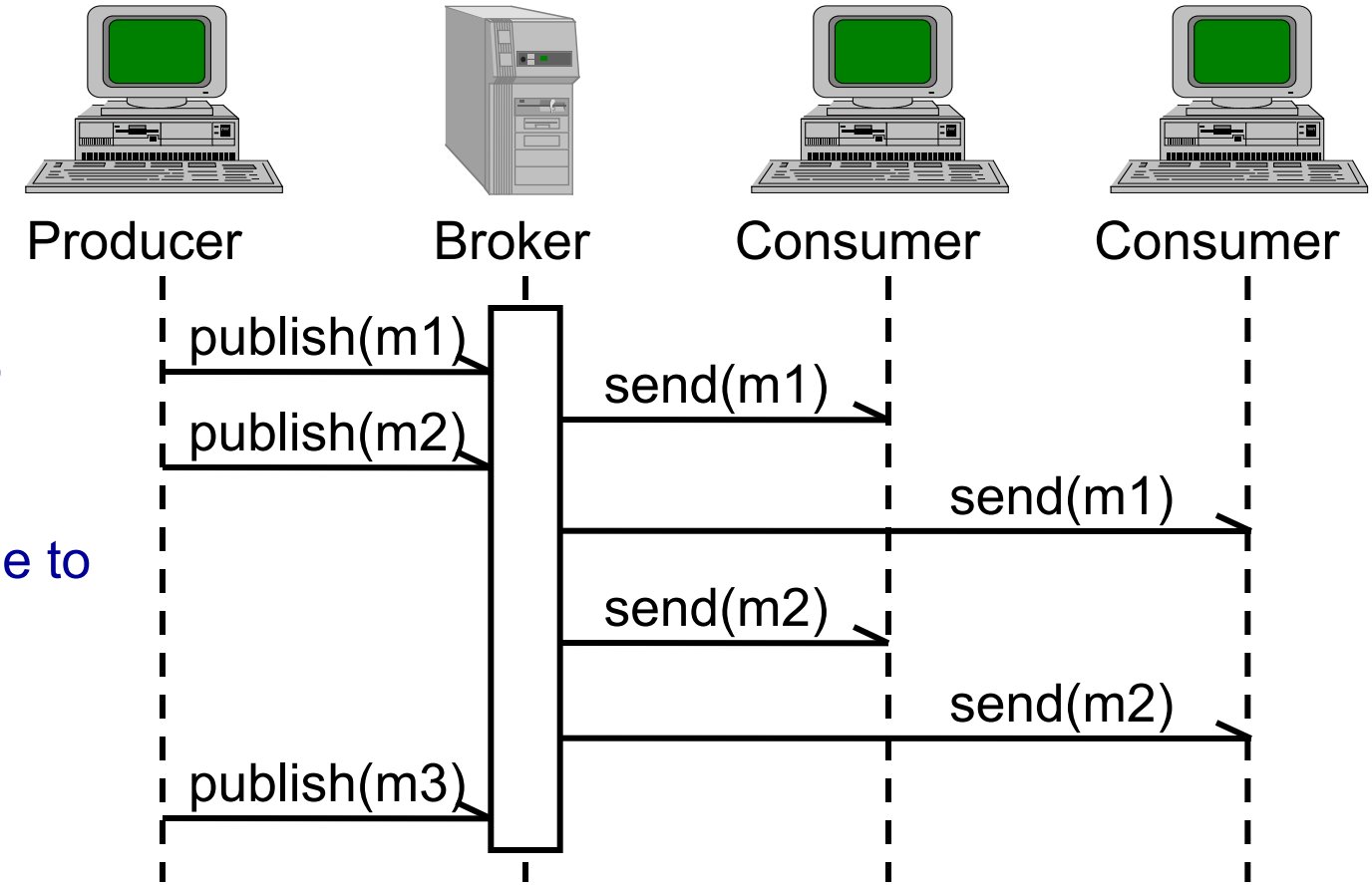
Put message into queue

Consume message

Source: Kelly & Maffeis

Non-Blocking Reception (Push)

- Aka passive reception
- Typically used with 1-n communication (Publish/Subscribe)



Pass message to broker

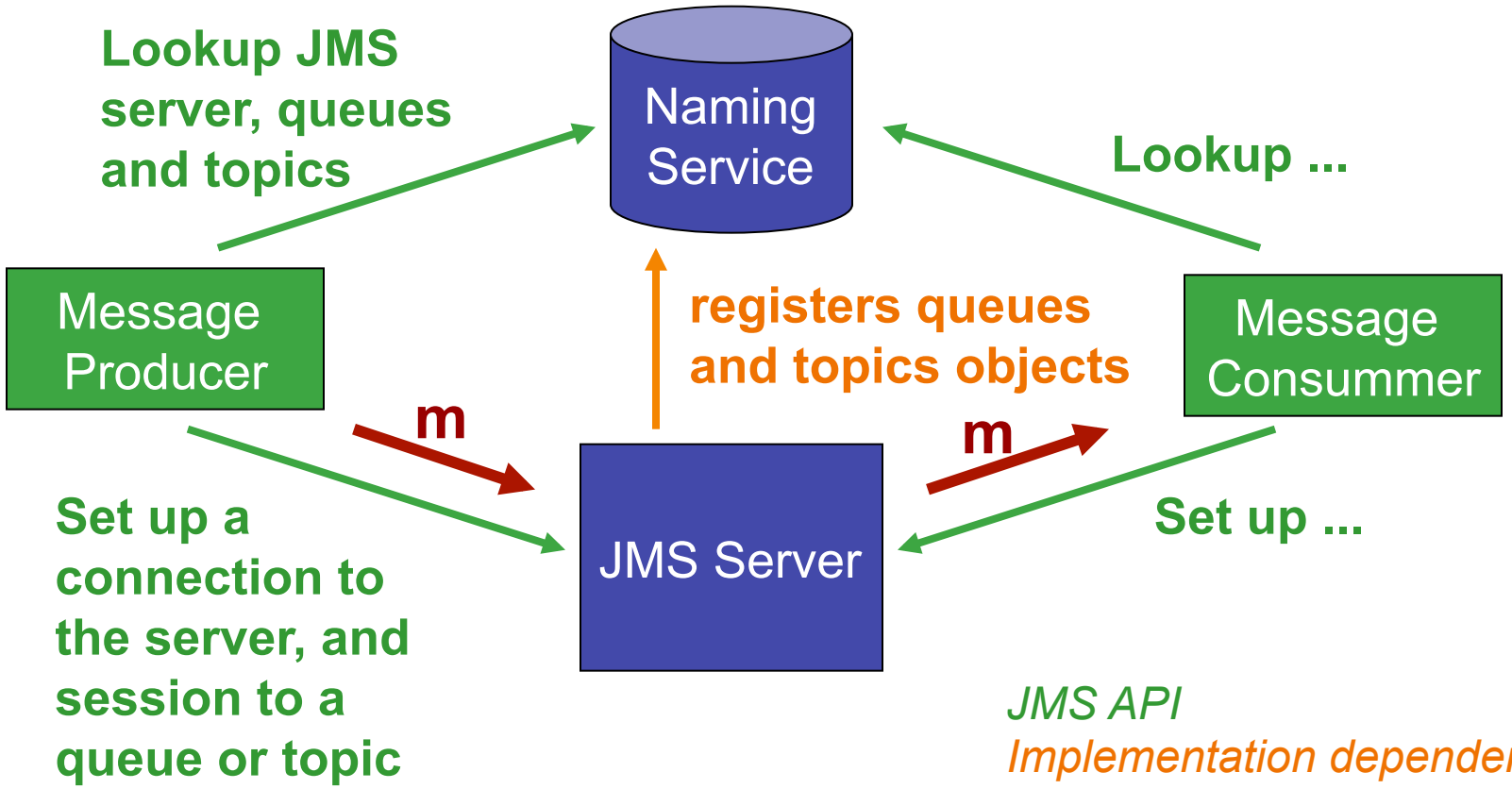
Dispatch message to all consumers

Source: Kelly & Maffeis

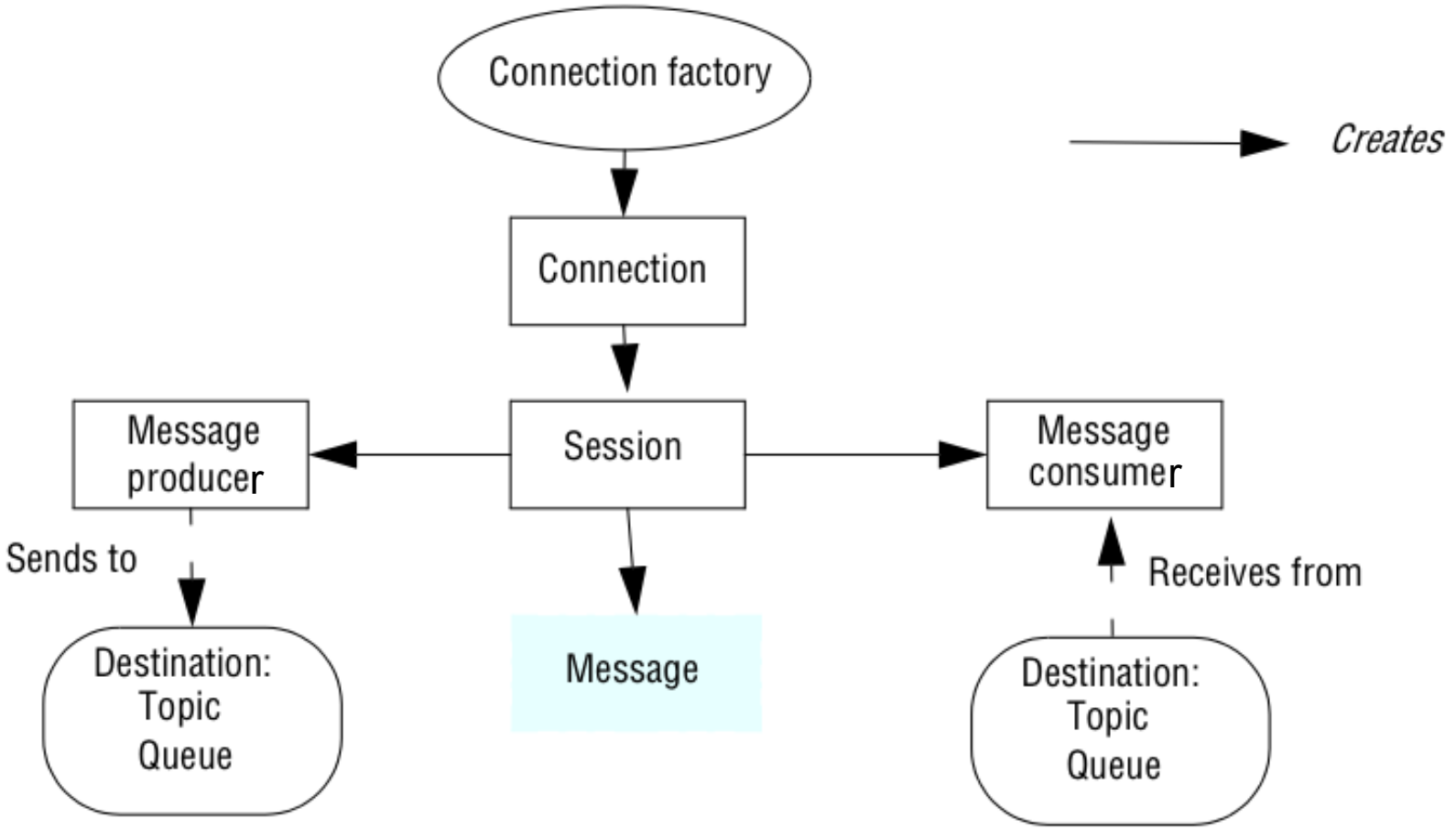
Important Note

- The JMS API **only** provide interfaces to program **the clients** of a messaging system
  - ➔ It provides means to retrieve references to Queue and Topic objects through a naming service (usually JNDI)
  - ➔ It does not permit the creation of Queues and Topics ('Destinations') on the server
- The **creation** of Queues and Topic on the server ("JMS Provider") is implementation dependant
  - ➔ In SUN's implementation done using the J2EE admin console
- **Goal**: The same client code can be used with different server implementations.

Typical JMS Architecture



The Main JMS Classes



The Main JMS Classes

- Actually they are not classes but interfaces
  - As a user you don't get to see the implementation classes
- Typical set up of a JMS client process
  - create a **Connection**
  - one or more **Sessions**
  - a number of **MessageProducers** and **MessageConsumers**
- A **Connection** object:
  - Encapsulates an open connection with a JMS provider
    - typically an open TCP/IP socket between a client and the JMS server
  - Its creation is where client authentication takes place
  - It is created from a **ConnectionFactory**
    - the connection factory is typically retrieved from the naming service
    - ConnectionFactory cf = NamingService.lookup("someName");

Session Objects

- A **Session** object is a single-threaded context for producing and consuming messages
  - ➔ It is created using **Connection.createSession(..)**
- A **Session** object serves several purposes
  - ➔ It is a factory for its message producers and consumers.
  - ➔ It is a scoping unit to perform atomic transactions that span its producers and consumers
  - ➔ It enforces a serial order for the messages it consumes and the messages it produces across all its producers and consumers
  - ➔ It retains messages until they have been acknowledged

Queue and Topic Objects

- **Queue** and **Topic** are both sub-interface of Destination
- Not created by clients but retrieved from a Naming Service
  - Actually what is retrieved is a *reference* to a topic or a queue
  - Same mechanisms as ConnectionFactories
    - Queue myQueue = NamingService.lookup("myQueue")
  - Queues, Topic + Connection Factories = "administered object"
    - They Must be set up in an implementation dependent manner
- They are needed to create **Message Consumer** and **Message Producer** from a Session object
  - MessageConsumer Session.createConsumer (Destination)
  - MessageProducer Session.createProducer (Destination)

Message Consumer and Producers

- They provide methods to send and receive messages either to/from a queue or about a topic
- **Message production** by Producer
  - ➔ send(Message message) + variant with fine tuning
- **Message reception** by Consumer
  - ➔ Synchronous: receive(), receive(long timeout), receiveNoWait()
  - ➔ Asynchronous: Listener mechanism setMessageListener(..)

Example: Fire Alarm (Publisher)

```
import javax.jms.*;
import javax.naming.*;

public class FireAlarmJMS {
  public void raise() {
    try {
      Context ctx = new InitialContext();
      TopicConnectionFactory topicFactory =
        (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
      Topic topic =       (Topic)ctx.lookup("Alarms");
      TopicConnection topicConn =
        topicFactory.createTopicConnection();
      TopicSession    topicSess  =
        topicConn.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
      TopicPublisher  topicPub =
        topicSess.createPublisher(topic);
      TextMessage     msg =
        topicSess.createTextMessage();
      msg.setText("Fire!");
      topicPub.publish(message);
    } catch (Exception e) {
    }
  } // EndMethod raise()
} // EndClass FireAlarmJMS
```

Jini Lookup

Connection, Session, and Publisher

message published

Example: Fire Alarm (Consumer)

```
import javax.jms.*;
import javax.naming.*;

public class FireAlarmConsumerJMS {
  public String await() {
    try {
      Context ctx = new InitialContext();
      TopicConnectionFactory topicFactory =
        (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
      Topic topic =      (Topic)ctx.lookup("Alarms");
      TopicConnection topicConn =
        topicFactory.createTopicConnection();
      TopicSession    topicSess   =
        topicConn.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
      TopicSubscriber topicSub =
        topicSess.createSubscriber(topic);
      topicSub.start();
      TextMessage msg
        = (TextMessage) topicSub.receive();
      return msg.getText();
    } catch (Exception e) {
    }
  } // EndMethod await()
} // EndClass FireAlarmConsumerJMS
```

Jini Lookup

Connection, Session, and Publisher

message received (blocking)

Using the code

- On the consumer: waiting for an alarm

```
FireAlarmConsumerJMS
  alarmCall = new FireAlarmConsumerJMS();
String msg = alarmCall.await();
```

- On the publisher's side: raising an alarm

```
FireAlarmJMS alarm = new FireAlarmJMS();
alarm.raise();
```

Advanced Aspects of JMS

- **Reliability**
  - By default message are sent in `PERSISTENT` mode
  - JMS server takes extra care to prevent message loss
  - In particular messages sent in this mode this are logged to stable storage when sent
  - Possible to switch this off to gain performance
- **Durability**
  - Default: consumers only receive messages sent while active
  - Possible to create "durable subscription": like asking a neighbour to record your favourite TV program while you're on holyday

Advanced Topics (cont.)

- **Message Expiration**
  - By default messages never expire
  - Possible to set expiration time
  - Messages not received after this time are destroyed

- **Transaction**
  - Grouping of a sequence of client operations (sending, receiving) into one atomic unit of work
  - If anything goes wrong, work done is rolled back and transaction can be started all over again

Summing Up

At the end of this Session:

- You should understand what **Message Oriented middleware** is about
- You should know what the **Java Messaging Service** is
- You should know the difference between **point-to-point** and **publish-subscribe** message communication, and between blocking and non-blocking reception
- You should have some idea of what the main classes of the JMS are and what they do

References

- J2EE API Documentation on JMS
  - ➔ http://docs.oracle.com/javaee/6/api/javax/jms/package-summary.html
- Java Message Service - What and Why?
  - ➔ Bill Kelly, Silvano Maffeis
  - ➔ www.jug.ch/events/slides/000508_jmsintro-english.pdf
- Chapter 45 of the J2EE Tutorial on Oracle's web site
  - ➔ http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html