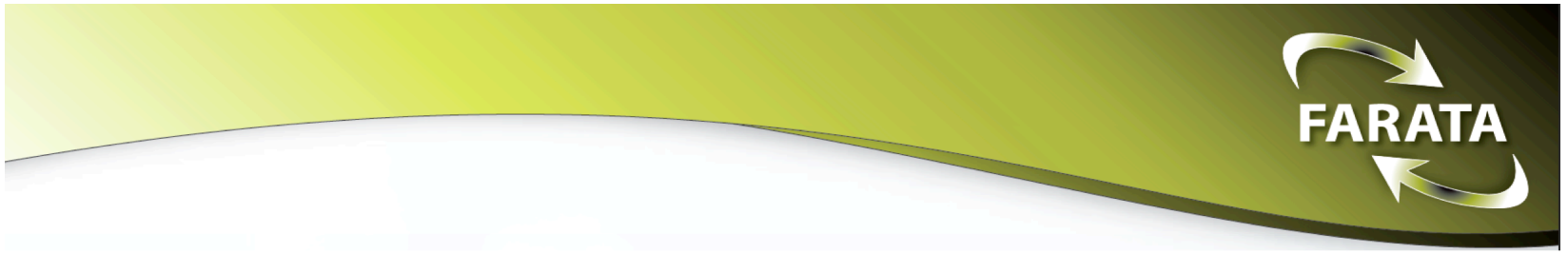


Java Programming
Unit 16

JNDI.
Java Messaging Service.

Java Naming and Directory Interface (JNDI)

Java Naming and Directory Interface

Naming and directory servers are registries of objects.

JNDI helps Java objects in finding required resources (e.d. data source, message queue, etc.).

Every Java app server creates internal JNDI tree of objects.

Administrator *binds* resources to the names in the JNDI tree. This is done via Admin Console, using scripts, or XML deployment descriptors.

Java Naming and Directory Interface

- JNDI `InitialContext` is the root of JNDI tree
- If your Java code runs inside Java EE server, it can *inject* the entries from JNDI to your code using `@Resource` annotation.
- Your program can also run a `lookup()` on JNDI tree to find resources.
- Standalone Java programs can only invoke `lookup()` to find the objects.

Getting InitialContext

Java program inside the app server:

```
Context namingContext = new InitialContext();
```

Java program outside of the app server (a Glassfish-specific example):

```
final Properties env = new Properties();

// JNDI properties are not the same in every Java EE server
env.put("java.naming.factory.initial”,
  "com.sun.enterprise.naming.SerialInitContextFactory");
props.setProperty("java.naming.factory.url.pkgs","com.sun.enterprise.nami
ng");
props.setProperty("java.naming.factory.state”,"com.sun.corba.ee.impl.pres
entation.rmi.JNDIStateFactoryImpl");
props.setProperty("org.omg.CORBA.ORBInitialHost", "localhost");
props.setProperty("org.omg.CORBA.ORBInitialPort", "8080");

Context namingContext = new InitialContext(env);
```

JMS Administered Objects

- JMS destinations (queues, topics) and connection factorues are typically maintained by adminitrators.
- Administrators configure (bind) administered objects to naming servers (JNDI, LDAP).
- Connection factory provides connectivity to MOM server.
- Connection factory is an instance of `ConnectionFactory`, `QueueConnectionFactory`, `TopicConnectionFactory`
- Destinations are instances of `Topic` or `Queue`.

Resource Injection

- Injection decouples your code from implementation of its dependencies

- Resource injection allows to inject any JNDI resource into a container-managed object, e.g. servlet, ejb, REST endpoint.

```
@Resource(name="java:comp/DefaultDataSource)
private javax.sql.DataSource myDataSource;
```

```
@Resource(lookup ="java:/ConnectionFactory")
    ConnectionFactory connectionFactory;

    @Resource(lookup = "queue/test")
    Queue testQueue;
```

Resource Lookup

- Finding JMS Connection factory:

```
ConnectionFactory connectionFactory = (ConnectionFactory)
          namingContext.lookup(connectionFactoryString);
```

- Finding a JMS destination (e.g. a msg queue *test*):

```
Destination destination = (Destination)
                      namingContext.lookup("jms/queue/test");
```

Java Messaging Service (JMS)

JMS and MOM

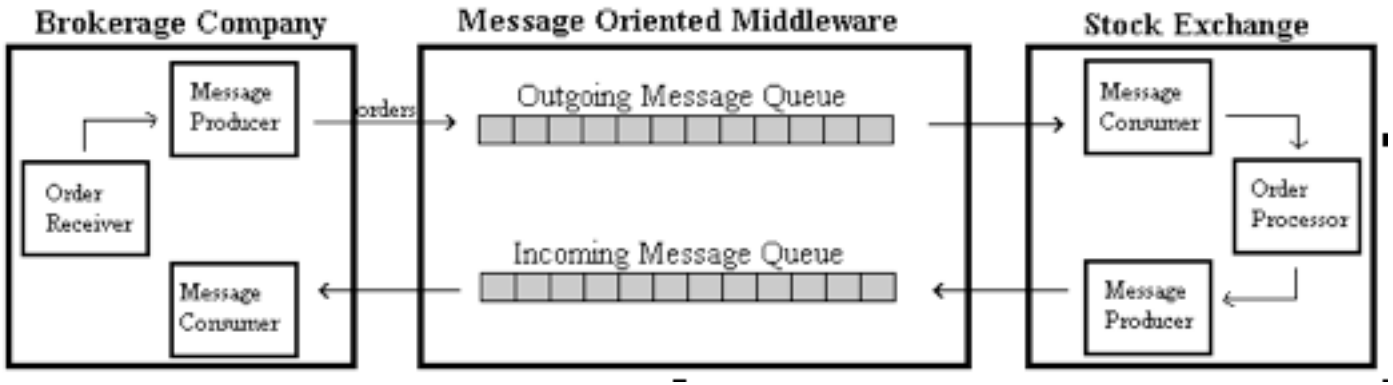
- Message Oriented Middleware (MOM) is a transport for messages, e.g. EMS, WebSphereMQ, ActiveMQ, et al. MOM is also known as JMS Provider

- JMS stands for Java Messaging Service

- JMS is an API for working with one of the MOM servers

- MOM is like a postal service

Point-to-Point (a.k.a. P2P) Architecture

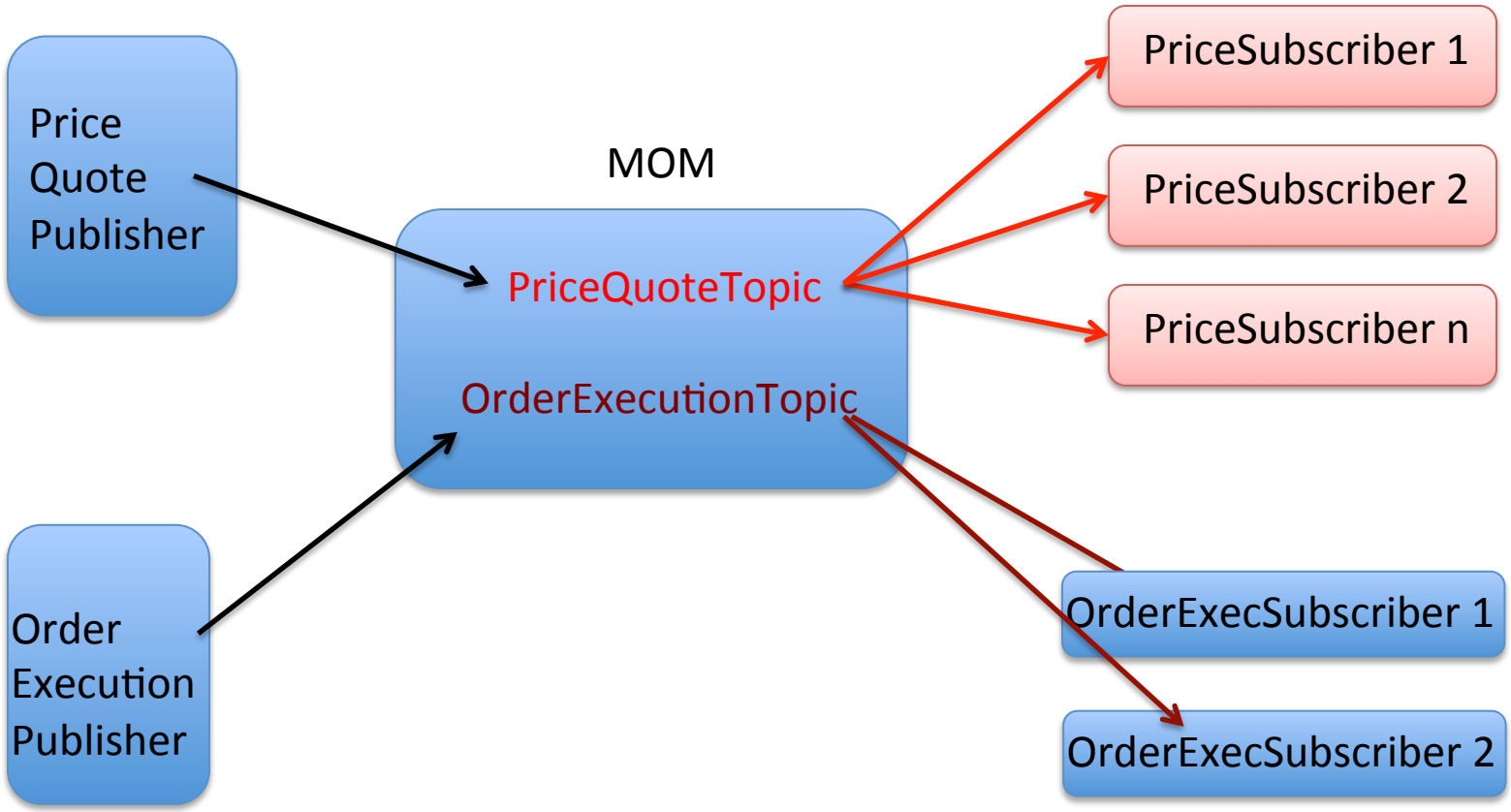


Point-to-Point messaging is when a program *sends* a message to a particular queue and and a *single consumer* receives the message from this queue.

In this mode the message is removed from a queue (*de-queued*) as soon as it's successfully delivered to the consumer.

Publish/Subscribe (a.k.a. Pub/Sub) Architecture

If a program publishes a message for multiple consumers, it's called *Publish/Subscribe messaging*.



JMS 1.1 API Overview

- Queue is a place to put in (or get from) your messages. **A message producer** (sender) puts messages in a queue and **a message consumer** (receiver) de-queues them.
- QueueConnection is an interface that represents connection to MOM.
- QueueConnectionFactory is an object that creates Connection objects.
- QueueSession is an object that represents a particular session between the client and MOM server. QueueConnection creates a session object.
- QueueSender is an object that actually sends messages.
- QueueReceiver receives messages.
- TopicPublisher publishes messages (it has similar functionality to QueueSender).
- TopicSubscriber is an object that receives messages (similar to QueueReceiver).
- Topic is an object that is used in Pub/Sub mode to represent some application event.
- TopicPublisher publishes messages to a topic so the TopicSubscriber(s) could subscribe for it.
- Message is an object that serves as a wrapper to an application objects that can be placed in a JMS queue or published to a topic.

Message Types

- `TextMessage` is an object that can contain any Java String.
- `ObjectMessage` can contain any `Serializable` Java object.
- `BytesMessage` contains an array of bytes.
- `StreamMessage` has a stream of Java primitives.
- `MapMessage` contains key/value pairs, e.g. `"id", 123`.

How to send a message (JMS 1.1)

1. Create (or get) a `ConnectionFactory` object.
2. Create a `Connection` object and call its method **start()**.
3. Create a `Session` object.
4. Create a `Queue` object.
5. Create a `MessageProducer` object.
6. Create one of the `Message` objects (e.g. `TextMessage`) and put some data in it.
7. Call the method `send()` on the `QueueSender`.
8. Close `QueueSender`, `Session` and `Connection` objects to release system resources.

Sending a message (JMS 1.1)

```
Session session=null;
        ConnectionFactory factory;
        QueueConnection connection=null;

        try{
            factory = new com.sun.messaging.ConnectionFactory();
            factory.setProperty(ConnectionConfiguration.imqAddressList, "mq://127.0.0.1:7677,mq://127.0.0.1:7677");
            connection = factory.createQueueConnection("admin","admin");

            connection.start();

            session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
            Queue ioQueue = session.createQueue("TestQueue");
            MessageProducer queueSender = session.createProducer(ioQueue);

            // Buy 200 shares of IBM at market price
            TextMessage outMsg = session.createTextMessage("IBM 200 Mkt");

            queueSender.send(outMsg);

            System.out.println("Sucsessfully placed an order to purchase 200 shares of IBM");
          }
          catch (JMSException e){
               System.out.println("Error: " + e.getMessage());
          }
          finally{…}
```

How to receive a message (JMS 1.1)

A message consumer doesn't need to request a message. The asynchronous callback method onMessage() will be called immediately when a message appears in the queue.

1. Create (or get from some naming server) the QueueConnectionFactory object.
2. Create a Connection object and call its method start().
3. Create a Session object.
4. Create a Queue object.
5. Create a QueueReceiver object.
6. If your class implements MessageListener write implementation for the callback method onMessage(). If you decide to get messages synchronously, just call the method QueueReceiver.receive().
7. Close the Session and Connection objects to release the system resources.

Receiving a message (JMS 1.1)

```
// in the constructor
 factory = new com.sun.messaging.ConnectionFactory();  // MOM-specific
factory.setProperty(ConnectionConfiguration.imqAddressList,
                                "mq://localhost:7677,mq://localhost:7677");
connection = factory.createQueueConnection("admin","admin");

connection.start();

Session session = connection.createQueueSession(
                    false, Session.AUTO_ACKNOWLEDGE);

Queue ioQueue = session.createQueue( "TestQueue" );

consumer = session.createConsumer(ioQueue);
consumer.setMessageListener(this);

System.out.println("Listening to the TestQueue...");

// Don't finish -  wait for messages
Thread.sleep(100000);
```

```
public void onMessage(Message msg){
        String msgText;
        try{
           if (msg instanceof TextMessage){
                 msgText = ((TextMessage) msg).getText();
                System.out.println("Got from the queue: " + msgText);
           }else{
             System.out.println("Got a non-text message");
           }
        }
        catch (JMSException e){
            System.out.println("Error while consuming a message: ”
                                                          + e.getMessage());
        }
}
```

Message Acknowledgements

Message acknowledgment mode is defined at the time of creation of the `Session` object. The method `createSession()` has two arguments.

If the first argument of `createSession()` **is true**, the session is transacted and the message could be either committed, or rolled back by the consumer.

Invokin **`commit()`** removes the message from the queue. The method `rollback()` leaves the message in the queue.

Message Acknowledgements (cont.)

If the first argument of `createSession()` **is false**, the second argument defines the acknowledgement mode.

- `AUTO_ACKNOWLEDGE` mode sends the acknowledgement back as soon as the method `onMessage()` is successfully finished.
- `CLIENT_ACKNOWLEDGE` mode requires explicit acknowledgement: `msg.acknowledge()`.
- `DUP_OK_ACKNOWLEDGE` – in case of server's failure the same message may be delivered more than once.

Publishing a Message (JMS 1.1)

```
TopicConnection connection = connectionFactory.createTopicConnection();

TopicSession pubSession = connection.createTopicSession(false,
                                                    Session.AUTO_ACKNOWLEDGE);

Topic myTopic = pubSession.createTopic ("Price_Drop_Alerts");

TopicPublisher publisher= pubSession.createPublisher(myTopic);

connection.start();

TextMessage message = pubSession.createTextMessage();

message.setText("The sale in Apple stores starts  tomorrow");

publisher.publish(message);
```

Subscribing to a Topic (JMS 1.1)

```
TopicSession subSession =  connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);

subSession.createTopic("Price_Drop_Alerts");

TopicSubscriber subscriber = subSession.createSubscriber(topic);

connection.start();

subscriber.setMessageListener(this);

public void onMessage(Message message) {

String msgText;
 try{
    if (msg instanceof TextMessage){
           msgText = ((TextMessage) msg).getText();
          System.out.println("Got " + msgText);
    }else{
       System.out.println("Got a non-text message");
    }
 }
 catch (JMSException e){
     System.out.println("Error: " + e.getMessage());
 }
}
```

Walkthrough 1 (start)

1. **Start the Open MQ server:** Open command (or Terminal) window, change (cd) to *glassfish4/mq/bin* directory, and start the Open MQ broker. In Windows OS run imqbrokerd.exe, in MAC OS do ./imqbrokerd -port 7677

2. Start Open MQ Admin Console: Open another Command or Terminal window, go to glassfish4/mq/bin directory again, and start the admin GUI tool imqadmin to create the required messaging destinations.

3.Create new message broker: Right-click on Brokers and add a new broker and named StockBroker, change the port to 7677, enter the password admin, and press OK.

4. Connect to the StockBroker (right-click menu), and add broker destination (right-click menu) named TestQueue.

On MAC you might get the error "Unsupported major.minor version 51.0".
To fix it, export JAVA_HOME
export JAVA_HOME=`/usr/libexec/java_home`
and start the imqbrokerd (and imqadmin) with parameter, for example:
./imqbrokerd -javahome $JAVA_HOME -port 7677
./imqadmin -javahome $JAVA_HOME

Walkthrough 1 (end)

5. Download the code for Lesson 30 and import it into Eclipse

6. Go to the project Properties | Java Build Path and add two jars in the Library section: imq.jar and jms.jar located in the glassfish4/mq/lib.

7. Review the code of the MessageSender – it connects to the MOM that runs on port 7677.

8. Start MesageReceiver and it'll print
Listening to the TestQueue...

9. Run MessageSender. It'll place the message in TestQueue and will print
Successfully placed an order to purchase 200 shares of IBM.

10. Check the console of the MessageReceiver. It received the message and printed Got from the queue: IBM 200 Mkt

11. Go back to Slide 3 and review the diagram. It should make more sense now.

Java EE 7 Includes JMS 2.0 (JSR 343)

JMS API 1.1 was not updated for more than 10 years.

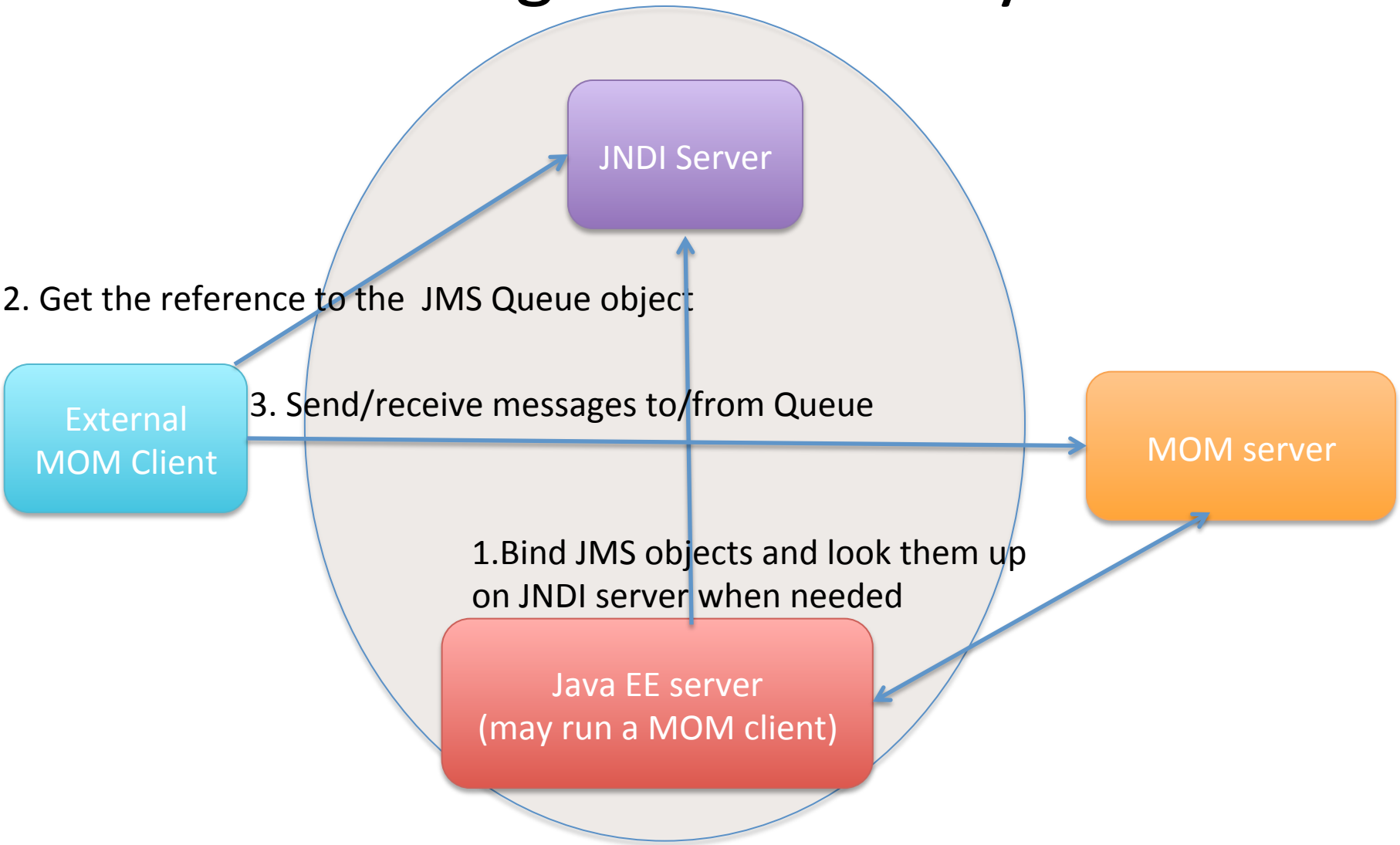
JMS 2.0 has simplified API, but the old JMS 1.1 code will still work.

The new `JMSContext` encapsulates both `Connection` and `Session`. It implements `AutoCloseable`.

JMS 2.0 has `JMSProducer`, `JMSConsumer`

`JMSException` is replaced with `JMSRuntimeException`

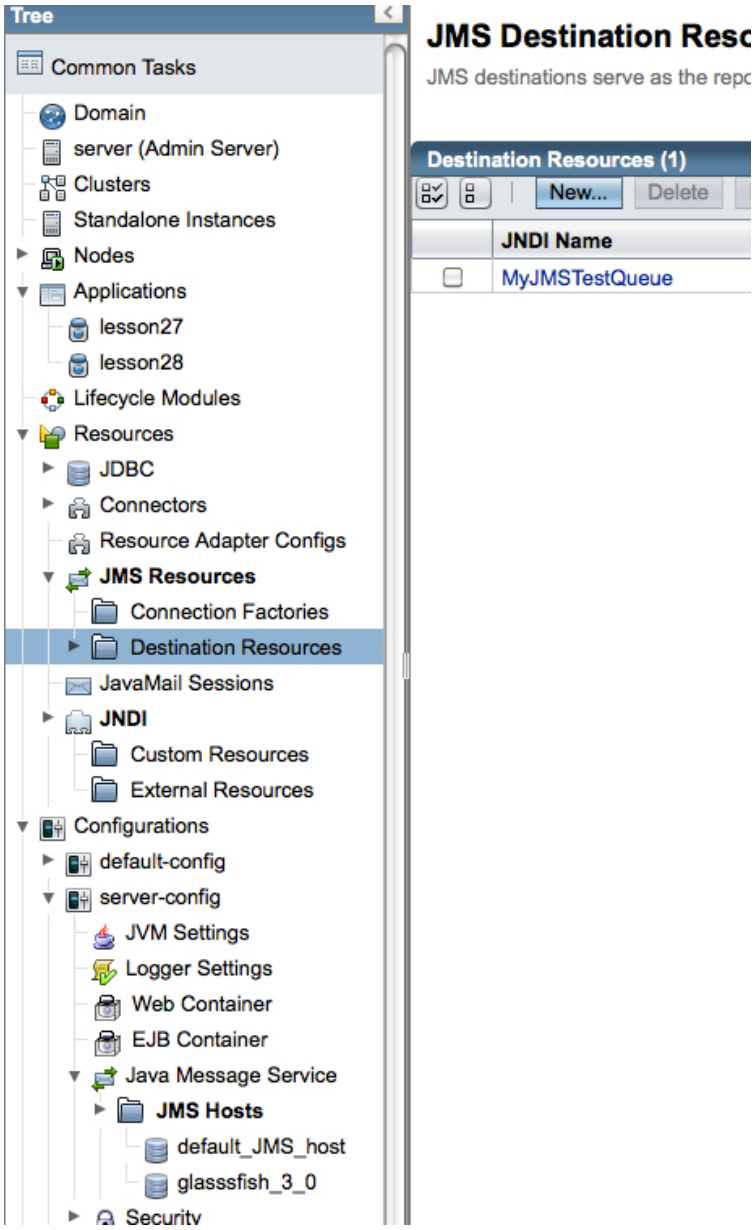
Java Naming and Directory Interface



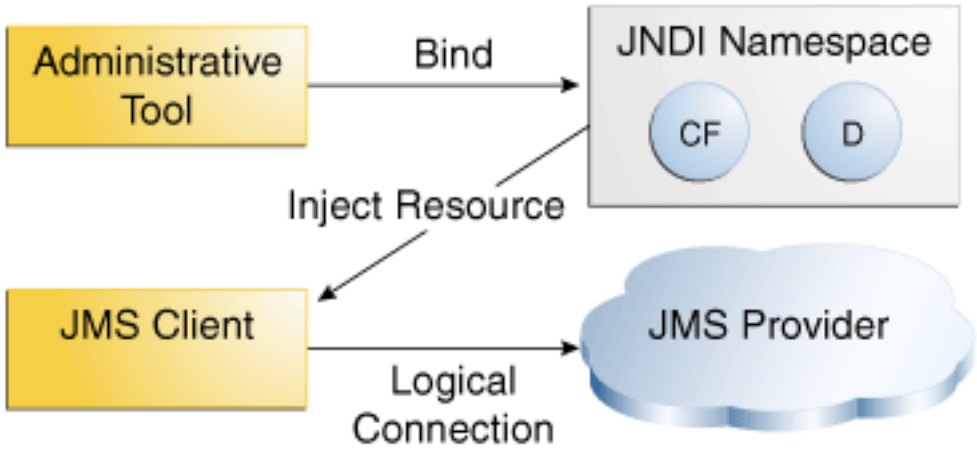
To replace MOM, just rebind new admin objects (queues, topics) to JNDI server (e.g. LDAP)

Configuring Resources with GlassFish

Start GlassFish and open its admin console at http://localhost:4848/



Binding JMS objects to JNDI



This diagram is taken from Oracle's Java EE 7 tutorial: http://bit.ly/193QHGW

Binding JMS admin objects to the external server allows quickly redirect the JMS client to another MOM.

Homework

1. Study the materials from the lessons 30 and 31 from the textbook.

2. Do the assignments from the Try It sections of the lesson 31.