# Использование JMS

JMS является еще одной технологией создания распределенных приложений, основанных на модели обмена сообщениями.

## Введение

*JMS (Java Messaging System)* представляет собой интерфейс к внешним системам, ориентированный на работу через сообщения. JMS является «старой» технологией – первая спецификация была опубликована в 1998г. В настоящее время пакет **javax.jms** входит в комплект **jdk**, а *Sun Application Server* реализует поддержку JMS в качестве одного из сервисов.

При разработке JMS в качестве основной задачи рассматривалось создание обобщенного Java API для приложений, ориентированных на работу с сообщениями (message-oriented application programming) и обеспечение независимости от конкретных реализаций соответствующих служб обработки сообщений.

Таким образом, программа, написанная с использованием JMS, будет корректно работать с любой системой сообщений, поддерживающей эту спецификацию (или имеющую соответствующие интерфейсы).

Поскольку JMS является лишь оболочкой или интерфейсом, описывающим доступные для приложения методы, для работы приложения понадобится определенная реализация этих интерфейсов JMS, называемая *провайдером JMS*. Такие реализации создаются независимыми производителями, и в настоящее время таких реализаций существует достаточно много (в том числе, например, реализация, включенная в Sun Application Server и распространяемая вместе с J2EE, а также MQSeries от IBM, служба JMS WebLogic от BEA, SonicMQ от Progress и другие).

Модель обмена сообщениями (и JMS) удобно использовать в том случае, если распределенное приложение обладает следующими характеристиками:

- взаимодействие между компонентами является асинхронным;
- информация (сообщение) должна передаваться нескольким или даже всем компонентам системы (семантика передачи от одного ко многим);
- передаваемая информация используется многими внешними системами, часть из которых не известна на момент проектирования системы или интерфейсы которых подвержены частым изменениям (концепция ESB – Enterprise Service Bus);
- обменивающиеся информацией (сообщениями) компоненты выполняются в разное время, что требует наличия посредника для промежуточного хранения переданной информации.

## Архитектура JMS

Архитектура JMS выглядит следующим образом (Рис. 7.1):

- Прикладные программы Java, использующие JMS, называются *клиентами JMS* (JMS client);
- Система обработки сообщений, управляющая маршрутизацией и доставкой сообщений, называется *JMS-провайдером* (JMS provider);



Курс подготовлен при поддержке Sun Microsystems
Правила использования материалов опубликованы на www.sun.ru

- *Приложение JMS* (JMS application) – это прикладная система, состоящая из нескольких JMS клиентов, и как правило одного JMS-провайдера. JMS-клиент, посылающий сообщение, называется *поставщиком* (producer). JMS-клиент, принимающий сообщение, называется *потребителем* (consumer). Один и тот же JMS клиент может быть одновременно и поставщиком и потребителем в разных актах взаимодействия;
- *Сообщения* (Messages) – это объекты, передающиеся и принимающиеся компонентами (клиентами JMS);
- *Средства администрирования* (Administrative tools) – средства управления ресурсами, использующимися клиентами.

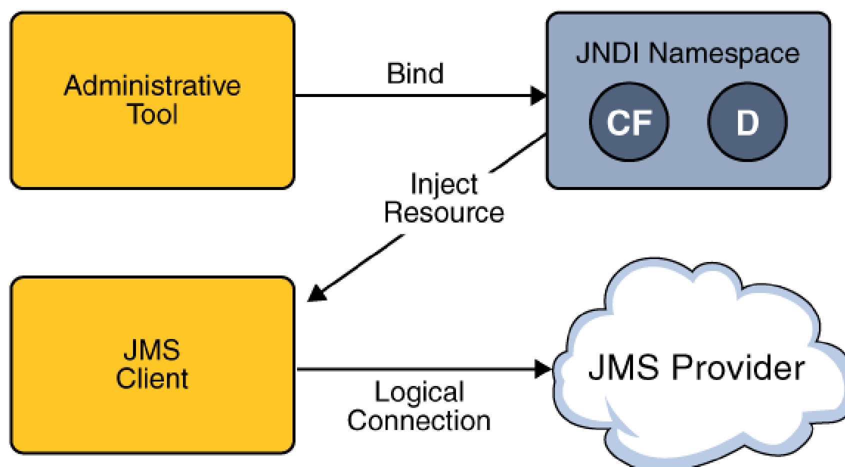


**Рис. 7.1. Архитектура JMS**

JMS предоставляет два подхода к передаче сообщений. Первый называется *«издание-подписка»* (publish an subscribe) (Рис. 7.3) и используется в том случае, если сообщение, отправленное одним клиентом должно быть получено несколькими.

Второй подход называется *«точка-точка»* (point to point) (Рис. 7.2) и служит для реализации обмена сообщениями между двумя компонентами.

Спецификация JMS называет эти два подхода *зонами сообщений* (messaging domains).

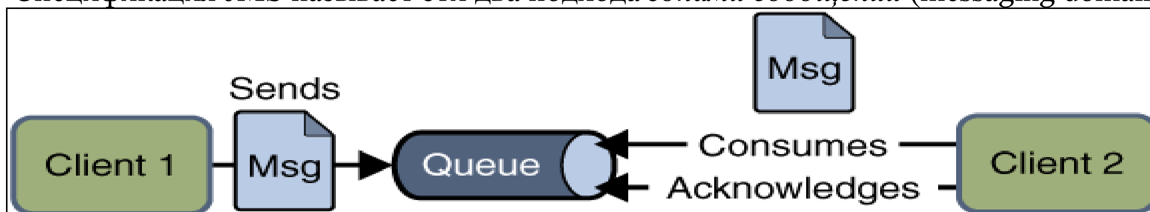


**Рис. 7.2. Модель взаимодействия «точка-точка»**



Курс подготовлен при поддержке Sun Microsystems
Правила использования материалов опубликованы на www.sun.ru

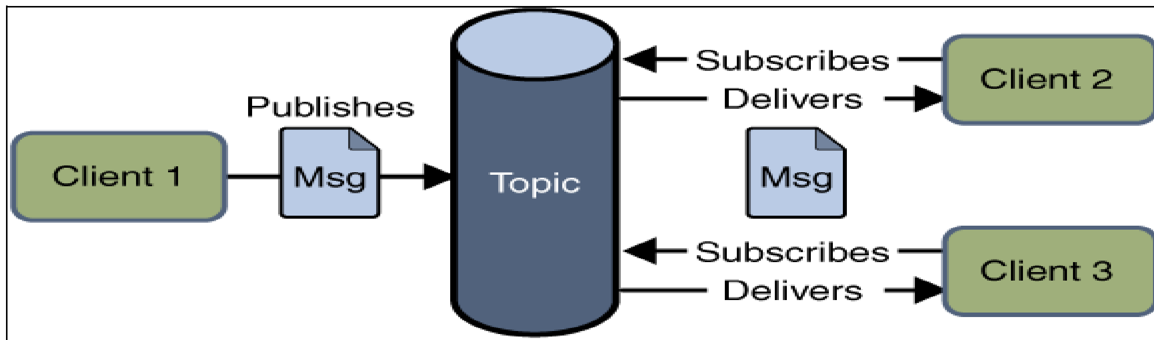


**Рис. 7.3. Модель взаимодействия «издание-подписка»**

### Модель взаимодействия точка-точка

Модель передачи сообщений «точка-точка» предоставляет возможность клиентам JMS посылать и принимать сообщения (как синхронно, так и асинхронно) через виртуальные каналы, называемые *очередями* (queues). Модель передачи сообщений «точка-точка» основывается на методе опроса, при котором сообщения явно запрашиваются (считываются) клиентом из очереди. Несмотря на то, что чтение из очереди могут осуществлять несколько клиентов, каждое сообщение будет прочитано только единожды - провайдер JMS это гарантирует.

### Модель взаимодействия издание-подписка

При использовании модели взаимодействия «издание-подписка» один клиент (поставщик) может посылать сообщения многим клиентам (потребителям) через виртуальный канал, называемый *темой* (topic). Потребители могут выбрать подписку (subscribe) на любую тему. Все сообщения, направляемые в тему, передаются всем потребителям данной темы. Каждый потребитель принимает копию каждого сообщения. Модель передачи сообщений издание-подписка, по существу, представляет собой модель, сервера, инициирующего соединение и «проталкивающего» информацию на клиента. В JMS эта концепция реализуется с помощью специальных *«слушателей»* (листенеров), регистрируемых в системе. При возникновении нового события листенер, закрепленный за данной темой, возбуждается.

Следует отметить, что при использовании модели «издание-подписка» клиенты JMS могут устанавливать долговременные подписки, позволяющие потребителям отсоединиться и позже снова подключиться и получать сообщения, поступившие во время отключения связи.

## *Первое знакомство*

Использование JMS предполагает выполнение разработчиком последовательности шагов. Ниже приведена общая схема использования JMS API.

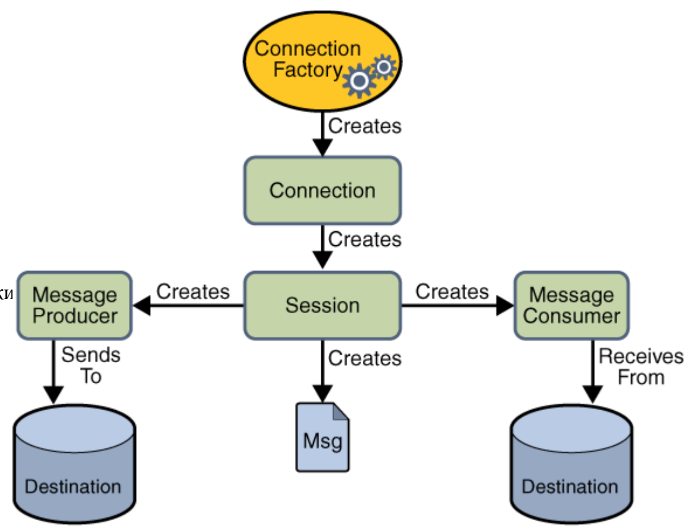

**Рис. 7.4. Общая схема использования JMS API**

Любой компонент, использующий JMS, прежде всего, должен создать соединение с JMS провайдером – собственно системой, обеспечивающей всю функциональности управления сообщениями.

Для этого используется специальная фабрика объектов - **ConnectionFactory**. **ConnectionFactory** на самом деле является интерфейсом, от которого наследуют **QueueConnectionFactory**, **TopicConnectionFactory**, **XAQueueConnectionFactory** и **XATopicConnectionFactory**. Таким образом, мы будем иметь дело с реализацией одного из этих интерфейсов. Для того, чтобы получить доступ к **ConnectionFactory**, программа должна извлечь соответствующую ссылку из JNDI. С использованием механизма аннотаций соответствующий код будет иметь следующий вид:

```
@Resource(mappedName="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
```

Указанный фрагмент кода извлекает ресурс с JNDI-именем **jms/ConnectionFactory** и связывает его с переменной **connectionFactory**.

Естественно, перед первым использованием **ConnectionFactory** должна быть создана:

```
asadmin.bat create-jms-resource --user admin --passwordfile admin-password.txt --host
localhost  --port 4848 --restype javax.jms.ConnectionFactory --enabled=true
jms/ConnectionFactory
```

Данная команда создает ресурс типа **javax.jms.ConnectionFactory** с именем **jms/ConnectionFactory**

Следующим шагом является создание соединения с JMS провайдером. Соответствующий код будет выглядеть следующим образом:

```
Connection connection = connectionFactory.createConnection();
```

Следует отметить, что соединение должно быть закрыто после того, как оно более не используется (**connection.close()**)

После того, как соединение создано, могут быть созданы виртуальные каналы, в рамках которых будет осуществляться передача сообщений. Существуют два типа таких каналов – *очередь* (Queue) и *тема* (Topic). Следует отметить, что создание виртуального канала требует наличия соответствующего *«пункта назначения»* (destination), созданного в JNDI. Таким образом, прежде чем программа сможет использовать очередь (или тему) соответствующий объект должен быть создан в JMS провайдере. Для *Sun Application Server* соответствующая команда будет иметь вид:

```
asadmin.bat create-jms-resource --user admin --passwordfile admin-password.txt --host
```



Курс подготовлен при поддержке Sun Microsystems
Правила использования материалов опубликованы на www.sun.ru

**localhost --port 4848 --restype javax.jms.Queue --enabled=true --property Name=PhysicalQueue jms/Queue**

где **restype** указывает тип объекта – в данном случае это **javax.jms.Queue**, а параметр **property** задает JNDI имя объекта - **jms/Queue**.

Для того, чтобы в программе получить ссылку на соответствующий объект, можно использовать код вида:

```
@Resource(mappedName="jms/Queue")
private static Queue queue;
```

для очереди, или, если нужно получить ссылку на объект типа тема:

```
@Resource(mappedName="jms/Topic")
private static Topic topic;
```

Еще раз обращаем внимание на то, что объекты с именами **jms/Queue** или **jms/Topic** должны быть предварительно созданы.

После того, как соединение с JMS провайдером установлено и получены ссылки на соответствующие пункты назначения (очередь или тема), может начаться собственно процесс обмена сообщениями.

Весь процесс обмена происходит в рамках сессии (Session), которая представляет собой однопотоковый контекст для обмена сообщениями. Сессия также предоставляет транзакционный контекст для обеспечения атомарности выполнения групп передач/приемов сообщений[1].

Сессия может быть создана следующим образом:

```
Session session = connection.createSession(false,  Session.AUTO_ACKNOWLEDGE);
```

где первый параметр указывает на то, что секция не транзакционна, а второй параметр указывает на то, что необходимо автоматически подтверждать успешную доставку сообщений.

В рамках сессии приложение может создать ряд объектов, обеспечивающих отправку и приемку сообщений. Это **Message Producers** и **Message Consumers**. Создаются эти объекты следующим образом:

```
MessageProducer producer = session.createProducer(queue);
MessageConsumer consumer = session.createConsumer(queue);
```

В качестве аргумента методы создания требуют указания пункта назначения. После создания посредством **MessageProducer** приложение может отправлять сообщения, а посредством **MessageConsumer** – принимать. Для отправки сообщения достаточно вызвать метод **send**, передав ему в качестве аргумента объект типа **Message**:

```
producer.send(message)
```

Для синхронного приема сообщения необходимо сначала запустить диспетчеризацию сообщения, затем вызвать метод **receive** соответствующего **MessageConsumer**. Метод **receive** является блокирующим, однако он может быть вызван с указанием количества миллисекунд, в течении которых он должен ожидать чтения сообщения.

```
connection.start();
Message m = consumer.receive();
```

при этом управление вернется в момент прихода сообщения, или

```
connection.start();
Message m = consumer.receive(2000);
```

при этом управление вернется в момент прихода сообщения или по истечению 2 секунд.

Для асинхронного приема сообщения может быть создан специальный класс –

[1] Сессии с поддержкой транзакционности в данном разделе не рассматриваются.



Курс подготовлен при поддержке Sun Microsystems
Правила использования материалов опубликованы на www.sun.ru

листенер, который должен быть зарегистрирован в рамках данного **MessageConsumer** и метод которого **onMessage** будет вызван в момент прихода сообщения.

```
Listener myListener = new Listener();
consumer.setMessageListener(myListener);
```

## Типы сообщений

Сообщение в JMS – это объект Java, состоящий из двух частей: *заголовка* (header) и *тела* (body) сообщения. В заголовке находится служебная информация, тело сообщения содержит в себе пользовательские данные, которые могут быть разной формы: текстовой, сериализуемые объекты, байтовые потоки и т. д.

JMS API определяет несколько типов сообщений :

- **BytesMessage** предназначен для передачи потока байт, который система никак не интерпретирует;
- **MapMessage** предназначен для передачи множества элементов типа «имя-значение», где имена являются объектами строкового типа, а значения – объектами примитивных типов данных Java;
- **ObjectMessage** предназначен для передачи сериализуемых объектов;
- **StreamMessage** предназначен для передачи множества элементов примитивных типов данных Java (они могут быть последовательно записаны, а затем прочитаны из тела сообщения этого типа);
- **TextMessage** предназначен для передачи текстовой информации.

## Первая программа

Приведем практический пример использования рассмотренных технологий и пройдем по шагам все этапы создания приложения с использованием JMS.

Во-первых, нам потребуется JMS провайдер. В качестве такового будет использован встроенный сервис Sun Java System Application Server, поставляемый в составе пакета J2EE. Инсталляция этого сервера приложений была рассмотрена ранее в разделе «».

Как уже говорилось ранее, первым шагом при разработке JMS приложения является создание необходимых ресурсов. Первый ресурс, который будет нами создан – **ConnectionFactory**. В нашем случае это делается командой

```
asadmin.bat create-jms-resource --user admin --passwordfile admin-password.txt --host
localhost  --port 4848 --restype javax.jms.ConnectionFactory --enabled=true
jms/Example1ConnectionFactory
```

Убедиться, что указанный ресурс создан можно с помощью административной консоли[2].

[2] Все действия, выполняемые нами с помощью утилиты asadmin, могут быть выполнены посредством административной консоли



Курс подготовлен при поддержке Sun Microsystems
Правила использования материалов опубликованы на www.sun.ru

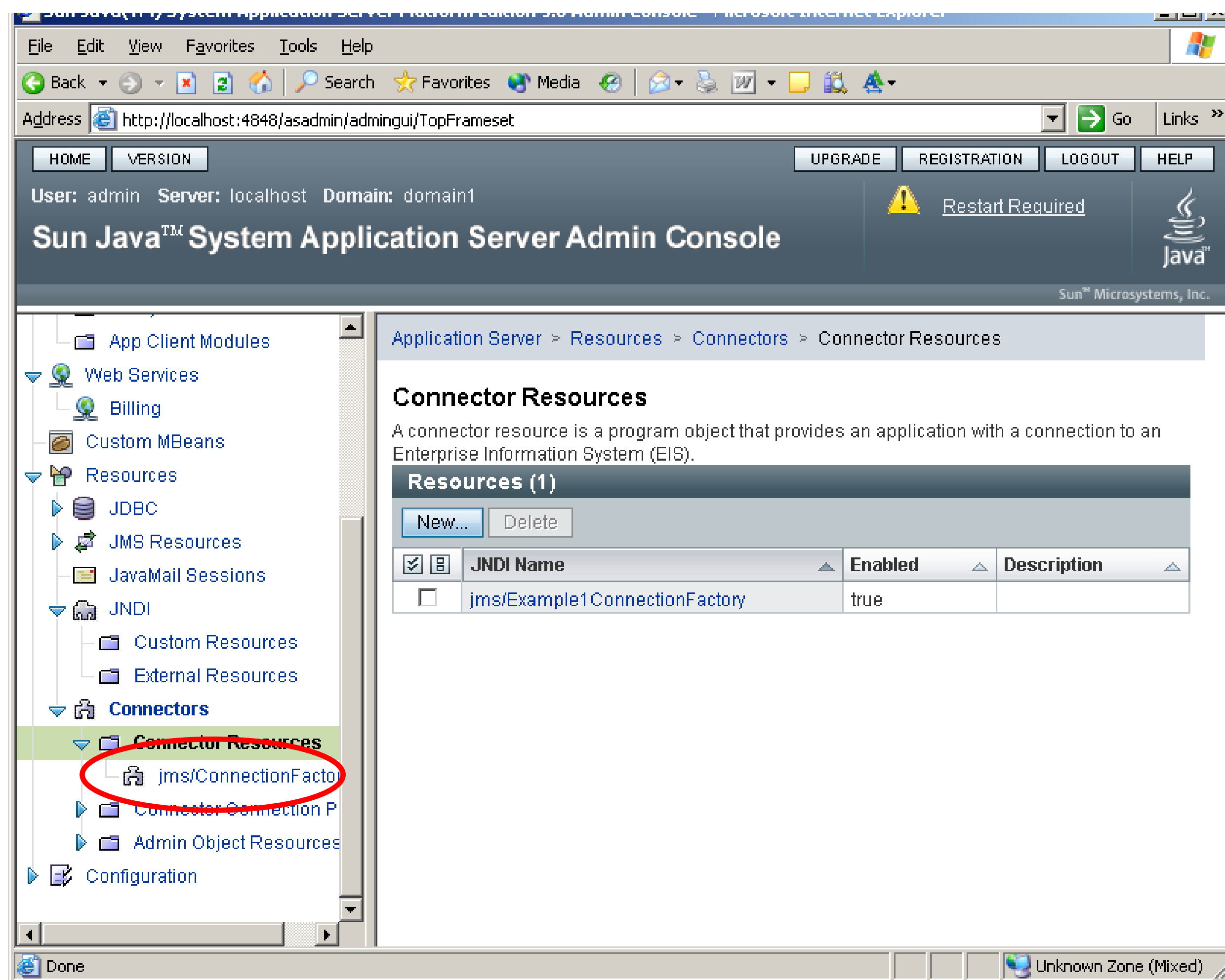

**Рис. 7.5. Административная консоль. Создание ConnectionFactory.**

Следующим ресурсом, который необходимо создать, является «пункт назначения». Для нашего примера будет создан пункт назначения типа очередь с именем **Example1Queue**
**asadmin.bat create-jms-resource --user admin --passwordfile admin-password.txt --host localhost --port 4848 --restype javax.jms.Queue --enabled=true --property Name=PhysicalQueue jms/Example1Queue**

Как и в случае с **ConnectionFactory,** созданный объект доступен в административной консоли.

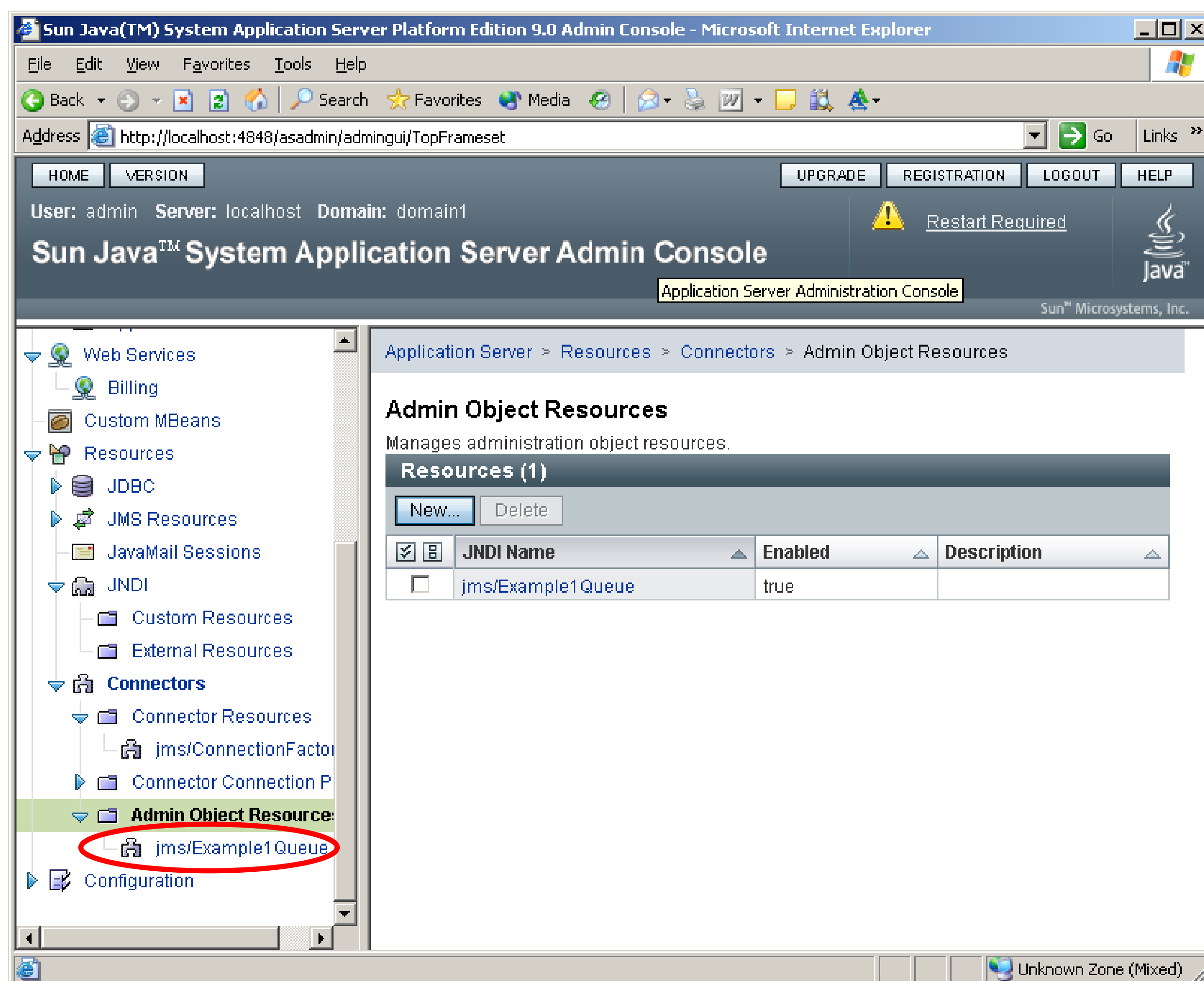


**Рис. 7.6. Административная консоль. Создание пункта назначения**

Теперь, после того, как все необходимые ресурсы созданы, можно приступить к созданию приложения. В качестве первого примера мы разработаем простое приложение, состоящие из двух компонентов, обменивающихся текстовыми сообщениями в синхронном режиме.

## JMSClient

Класс **JMSClient** после своего запуска создает соединение с JMS провайдером, создает сессию и **MessageProducer**, после чего, в ответ на каждый ввод пользователя посылает в очередь текстовое сообщение. Работа завершается, если пользователь ввел символ **q** (или **Q**).

---

```
1    import javax.jms.ConnectionFactory;
2    import javax.jms.Destination;
3    import javax.jms.Queue;
4    import javax.jms.Topic;
5    import javax.jms.Connection;
6    import javax.jms.Session;
```



Курс подготовлен при поддержке Sun Microsystems
Правила использования материалов опубликованы на www.sun.ru

```
7       import javax.jms.MessageProducer;
8       import javax.jms.TextMessage;
9       import javax.jms.JMSException;
10      import javax.annotation.Resource;
11      import java.io.InputStreamReader;
12      import java.io.IOException;
13      public class JMSClient {
14        @Resource(mappedName = "jms/Example1ConnectionFactory")
15        private static ConnectionFactory connectionFactory;
16        @Resource(mappedName = "jms/Example1Queue")
17        private static Queue queue;

18        public static void main(String[] args) {
19          Connection connection = null;

20          Destination dest = (Destination) queue;

21          try {
22            connection = connectionFactory.createConnection();
23      Session session =
        connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
24            MessageProducer producer = session.createProducer(dest);
25            TextMessage message = session.createTextMessage();
26      InputStreamReader inputStreamReader = new InputStreamReader(System.in);
27            char c = 'n';
28            int i = 0;
29            while (!((c == 'q') || (c == 'Q'))) {
30              try {
31                c = (char) inputStreamReader.read();
32                message.setText("This is message " + (i + 1));
33                System.out.println("Sending message: " + message.getText());
34                producer.send(message);
35                i++;
36              } catch (IOException e) {
37                System.err.println("I/O exception: " + e.toString());
38              }
39            }
40          } catch (JMSException e) {
41            System.err.println("Exception occurred: " + e.toString());
42          } finally {
43            if (connection != null) {
44              try {
45                connection.close();
46              } catch (JMSException e) {
47              }
48            }
49          }
50        }
```



Курс подготовлен при поддержке Sun Microsystems
Правила использования материалов опубликованы на www.sun.ru

```
51    }
```

**Описание 7.1. Класс JMSClient**

Программа демонстрирует весь цикл создания типичного JMS приложения. Сначала (строки 14-17) определяются необходимые ресурсы – в данном случае **ConnectionFactory** и **Queue** (эти ресурсы с соответствующими именами были ранее созданы).

Затем создается соединение с JMS провайдером (строка 22) и сессия (строка 23). Следующим шагом является создание **MessageProducer** (строка 24) и текстового сообщения (строка 25).

Цикл работы программы (строки 29-39) состоит в ожидании ввода пользователя (строка 31), формировании тела сообщения (строка 32) и его отправки в очередь (строка 34). В конце работы программа закрывает соединение с JMS провайдером (строка 45).

## JMSServer

Класс **JMSServer** служит для обработки сообщений, отправляемых **JMSClient**. Он синхронным образом считывает сообщения из очереди и печатает их.

```
1     import javax.jms.ConnectionFactory;
2     import javax.jms.Destination;
3     import javax.jms.Queue;
4
5     import javax.jms.Connection;
6     import javax.jms.Session;
7     import javax.jms.MessageConsumer;
8     import javax.jms.Message;
9     import javax.jms.TextMessage;
10    import javax.jms.JMSException;
11    import javax.annotation.Resource;
12    public class JMSServer {
13       @Resource(mappedName = "jms/Example1ConnectionFactory")
14       private static ConnectionFactory connectionFactory;
15       @Resource(mappedName = "jms/Example1Queue")
16       private static Queue queue;

17       public static void main(String[] args) {
18          String destType = null;
19          Connection connection = null;
20          Session session = null;
21          Destination dest = (Destination) queue;;
22          MessageConsumer consumer = null;
23          TextMessage message = null;


24          try {
25             connection = connectionFactory.createConnection();
26    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

```
27            consumer = session.createConsumer(dest);
28            connection.start();

29            while (true) {
30               Message m = consumer.receive(1);

31               if (m != null) {
32                  if (m instanceof TextMessage) {
33                     message = (TextMessage) m;
34                     System.out.println(
35                           "Reading message: " + message.getText());
36                  }
37               }
38            }
39         } catch (JMSException e) {
40            System.err.println("Exception occurred: " + e.toString());
41         } finally {
42            if (connection != null) {
43               try {
44                  connection.close();
45               } catch (JMSException e) {
46               }
47            }
48         }
49      }
50   }
```

**Описание 7.2. Класс JMSServer**

Отличие от предыдущего класса заключается в том, что создается не **MessageProducer**, а **MessageConsumer** (строка 27), поскольку мы собираемся не отправлять сообщения, а принимать их. Для того, чтобы начать обработку сообщений, необходимо вызвать метод **start** (строка 28).

Цикл работы программы (строки 29-38) состоит в ожидании прихода нового сообщения (строка 30). В данном случае ожидание продолжается 1 миллисекунду. Если пришедшее сообщение имеет текстовый тип (именно такие сообщения мы ожидаем получить), оно извлекается и печатается тело сообщения.

## Компиляция и запуск

Компиляция приложений JMS не имеет особенностей, единственное, что нужно сделать – указать соответствующие библиотеки в classpath.

**javac -classpath .;activation.jar;appserv-ws.jar;javaee.jar;jsf-impl.jar;appserv-jstl.jar; -d build JMSClient.java**

**javac -classpath .;activation.jar;appserv-ws.jar;javaee.jar;jsf-impl.jar;appserv-jstl.jar; -d build JMSServer.java**



Курс подготовлен при поддержке Sun Microsystems
Правила использования материалов опубликованы на www.sun.ru

Затем нужно собрать соответствующие пакеты (jar):
**jar -cvfm server.jar manifest2.mf -C ./build .**
**jar -cvfm client.jar manifest.mf -C ./build .**
При этом файлы манифеста имеют следующий вид: *manifest.mf* – Рис. 7.7 и *manifest2.mf* - Рис. 7.8.

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.6.5
Created-By: asw
Main-Class: JMSClient
```

**Рис. 7.7. Файл manifest.mf**

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.6.5
Created-By: asw
Main-Class: JMSServer
```

**Рис. 7.8. Файл manifest2.mf**

В результате будут созданы два пакета – **server.jar** и **client.jar**.
Запуск приложения производится с помощью утилиты **appclient**, соответственно:
**appclient -client client.jar**
для клиента (отправителя сообщений) и
**appclient -client server.jar**
для сервера (обработчика сообщений).

После запуска, клиент в ответ на ввод пользователя будет посылать сообщения в очередь, сервер будет их принимать и печатать.

Следует обратить внимание, что сервер и клиент не обязательно должны быть запущены одновременно. Так, клиент, будучи запущенным, может сгенерировать несколько сообщений и закончить работу. Сервер, будучи запущенным позднее, эти сообщения получит, поскольку JMS провайдер их сохранил.

Ниже приведен пример работы системы.
**appclient -client client.jar**
Запуск клиента приведет к выводу сообщений об отправке.
**Sending message: This is message 1**
**Sending message: This is message 2**
**Sending message: This is message 3**
**Sending message: This is message 4**
**Sending message: This is message 5**
**Sending message: This is message 6**
**Sending message: This is message 7**
**Sending message: This is message 8**
**Sending message: This is message 9**



Курс подготовлен при поддержке Sun Microsystems
Правила использования материалов опубликованы на www.sun.ru

**Sending message: This is message 10**

Запуск сервера:

**appclient -client server.jar**

приведет к выводу сообщений о приеме.

**Reading message: This is message 1**
**Reading message: This is message 2**
**Reading message: This is message 3**
**Reading message: This is message 4**
**Reading message: This is message 5**
**Reading message: This is message 6**
**Reading message: This is message 7**
**Reading message: This is message 8**
**Reading message: This is message 9**
**Reading message: This is message 10**

## *Пример*

Теперь, после того как основные концепции JMS проиллюстрированы, мы можем приступить к реализации системы, обеспечивающей решение поставленной нами задачи с использованием механизмов JMS.

Как обычно, система будет состоять из двух компонентов – клиентского, передающего данные о новых картах и операциях над ними, и серверного, принимающего и обрабатывающего эти данные.

Для передачи данных мы воспользуемся сообщениями типа **ObjectMessage** и транспортными классами, которые будут поддерживать механизм сериализации. Кроме этого, в серверном компоненте будет реализован асинхронный механизм обработки сообщений, с использованием соответствующего листенера.

Перед компиляцией и выполнением примера необходимо создать соответствующие ресурсы – **ConnectionFctory** и **Queue** с именами **jms/ConnectionFactory** и **jms/Queue** соответственно.

### Клиентский класс BillingClient

```
1     package com.asw.jms.ex1;
2
3     import javax.jms.*;
4     import javax.annotation.Resource;
5     import java.util.Date;
6
7     public class BillingClient {
8         @Resource(mappedName = "jms/ConnectionFactory")
9         private static ConnectionFactory connectionFactory;
10        @Resource(mappedName = "jms/Queue")
11        private static Queue queue;
```



Курс подготовлен при поддержке Sun Microsystems
Правила использования материалов опубликованы на www.sun.ru

```
12
13      public static void main(String[] args) {
14         Connection connection = null;
15         Destination dest = (Destination) queue;
16         try {
17            connection = connectionFactory.createConnection();
18            Session session = connection.createSession(false,
19                     Session.AUTO_ACKNOWLEDGE);
20            MessageProducer producer = session.createProducer(dest);
21            ObjectMessage message = session.createObjectMessage();
22            Card c = new Card ("Piter",new Date(),"1",0.0);
23            message.setObject(c);
24            producer.send(message);
25            c = new Card("Stefan",new Date(),"2",0.0);
26            message.setObject(c);
27            producer.send(message);
28            c = new Card("Nataly",new Date(),"3",0.0);
29            message.setObject(c);
30            producer.send(message);
31            int cnt = 1000;
32            for (int i = 0; i < cnt; i++) {
33   CardOperation co = new CardOperation ((i%3+1)+"",10+i,new Date());
34               message.setObject(co);
35               System.out.println("Sending message: " + message.getObject());
36               producer.send(message);
37            }
38            /*
39             * Send a non-text control message indicating end of
40             * messages.
41             */
42            producer.send(session.createMessage());
43         } catch (JMSException e) {
44            System.err.println("Exception occurred: " + e.toString());
45         } finally {
46            if (connection != null) {
47               try {
48                  connection.close();
49               } catch (JMSException e) {
50               }
51            }
52         }
53      }
54   }
```

**Описание 7.3. Класс BillingClient**

Отличие от рассмотренного ранее примера состоит в том, что создается сообщение типа **ObjectMessage** (строка 21). Для помещения объекта в тело этого сообщения необходимо

вызвать метод **setObject** (конечно, класс, объект которого помещается в сообщение, должен быть объявлен как реализующий интерфейс **Serializable**).

## Транспортный класс Card

Транспортные классы **Card** (Описание 7.4) и **CardOperation** (Описание 7.5) полностью аналогичны рассматриваемым ранее.

```
1     package com.asw.jms.ex1;
2
3     import java.io.Serializable;
4     import java.util.*;
5
6
7     public class Card implements Serializable{
8     public Card(String person, Date createDate, String cardNumber,double balance){
9                 this.person = person;
10                this.createDate = createDate;
11                this.cardNumber = cardNumber;
12                this.balance = balance;
13        }
14        public String person;
15        public Date createDate;
16        public String cardNumber;
17        public double balance;
18        public String toString(){
19    return "Card: 
      cardNumber="+cardNumber+"\tBalance="+balance+"\tPerson="+person+"\tCre
      ateDate="+createDate+"";
20        }
21    }
```

**Описание 7.4. Класс Card**

## Транспортный класс CardOperation

```
1     package com.asw.jms.ex1;
2
3     import java.util.*;
4     import java.io.*;
5
6     public class CardOperation implements Serializable {
```

```
7     public CardOperation(String card,double amount,Date operationDate){
8                   this.card = card;
9                   this.amount = amount;
10                  this.operationDate = operationDate;
11    }
12    public String card;
13    public double amount;
14    public Date operationDate;
15    }
```

**Описание 7.5. Класс CardOperation**

## Серверный класс BillingService

Серверный класс (Описание 7.6) содержит методы, обеспечивающие выполнение операций с картами и является контейнером для хранения карт.

Для обработки (получения) сообщений он регистрирует **ObjectListener**.

```
1     package com.asw.jms.ex1;
2
3     import javax.jms.*;
4     import javax.annotation.Resource;
5     import java.io.InputStreamReader;
6     import java.io.IOException;
7     import java.util.Hashtable;
8     import java.util.Enumeration;
9
10    public class BillingService {
11      @Resource(mappedName = "jms/ConnectionFactory")
12      private static ConnectionFactory connectionFactory;
13      @Resource(mappedName = "jms/Queue")
14      private static Queue queue;
15
16      Hashtable hash = new Hashtable();
17      public void addNewCard(Card c) {
18       hash.put(c.cardNumber, c);
19      };
20
21      public void performCardOperation(CardOperation co){
22       Card c = (Card)hash.get(co.card);
23       if (c==null) return;
24       c.balance+=co.amount;
25       hash.put(co.card,c);
26      };
27
28      public void printCards(){
29       for(Enumeration e = hash.elements();e.hasMoreElements();)
```



Курс подготовлен при поддержке Sun Microsystems
Правила использования материалов опубликованы на www.sun.ru

```
30                  System.out.println(e.nextElement());
31          }
32
33          public static void main(String[] args) {
34              String destType = null;
35              Connection connection = null;
36              Session session = null;
37              Destination dest = (Destination) queue;
38              MessageConsumer consumer = null;
39              ObjectListener listener = null;
40              TextMessage message = null;
41              InputStreamReader inputStreamReader = null;
42              char answer = '\0';
43              try {
44                  connection = connectionFactory.createConnection();
45             session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
46                  consumer = session.createConsumer(dest);
47                  listener = new ObjectListener(new BillingService());
48                  consumer.setMessageListener(listener);
49                  connection.start();
50                  System.out.println(
51                          "To end program, type Q or q, " + "then <return>");
52                  inputStreamReader = new InputStreamReader(System.in);
53                  while (!((answer == 'q') || (answer == 'Q'))) {
54                      try {
55                          answer = (char) inputStreamReader.read();
56                      } catch (IOException e) {
57                          System.err.println("I/O exception: " + e.toString());
58                      }
59                  }
60              } catch (JMSException e) {
61                  System.err.println("Exception occurred: " + e.toString());
62              } finally {
63                  if (connection != null) {
64                      try {
65                          connection.close();
66                      } catch (JMSException e) {
67                      }
68                  }
69              }
70          }
71      }
```

**Описание 7.6. Класс BillingService**

После создания соединения с JMS-провайдером (строка 44) и создания сессии (строка 45) создается **MessageConsumer** (строка 46). Затем создается **ObjectListener** (строка 47) и регистрируется в **MessageConsumer**, как принимающий сообщения (строка 48). Таким

образом, при приходе нового сообщения в очередь, будет вызываться метод **onMessage** листенера, который и будет обрабатывать сообщение. Метод **start** (строка 49) запускает обработку сообщений.

## Серверный листенер ObjectListener

**ObjectListener** (Описание 7.7) обрабатывает поступающие сообщения. При приходе нового сообщения он проверяет его тип (сообщение должно быть типа **ObjectMessage**, приход текстового сообщения интерпретируется как команда выдать балансы всех карт), извлекает объект, переданный в теле сообщения, проверяет его тип (ожидается, что пришедшие объекты могут быть двух типов – **Card** и **CardOperation**), после чего вызывает соответствующий метод **BillingService**, выполняющий нужную операцию.

```
1    package com.asw.jms.ex1;
2
3    import javax.jms.*;
4
5
6    public class ObjectListener implements MessageListener {
7       BillingService bs;
8       public ObjectListener (BillingService bs){
9        this.bs = bs;
10      }
11      public void onMessage(Message message) {
12         ObjectMessage msg = null;
13
14         try {
15            if (message instanceof ObjectMessage) {
16               msg = (ObjectMessage) message;
17                Object o = msg.getObject();
18               System.out.println("Reading message: " + o);
19                if (o instanceof Card) bs.addNewCard((Card)o);
20   else if (o instanceof CardOperation)
     bs.performCardOperation((CardOperation)o);
21            } else {
22               System.err.println("Message is not a ObjectMessage");
23                bs.printCards();
24            }
25         } catch (JMSException e) {
26            System.err.println("JMSException in onMessage(): " + e.toString());
27         } catch (Throwable t) {
28            System.err.println("Exception in onMessage():" + t.getMessage());
29         }
30      }
31   }
```

**Описание 7.7. Класс ObjectListener**



Курс подготовлен при поддержке Sun Microsystems
Правила использования материалов опубликованы на www.sun.ru

## Компиляция и запуск

Компиляция и выполнение не отличаются от рассмотренного ранее примера. Ниже приведен пример работы системы.

Запуск клиента:

**appclient -client BillingClient.jar**

Результат выполнения клиентской программы приведен на Рис. 7.9.

---

**Sending message: com.asw.jms.ex1.CardOperation@1e0bf98**

**…..**

**Sending message: com.asw.jms.ex1.CardOperation@11fb8c6**
**Sending message: com.asw.jms.ex1.CardOperation@35378d**
**Sending message: com.asw.jms.ex1.CardOperation@190c5c0**
**Sending message: com.asw.jms.ex1.CardOperation@1681629**
**Sending message: com.asw.jms.ex1.CardOperation@1c39aa6**
**Sending message: com.asw.jms.ex1.CardOperation@44c4a4**
**Sending message: com.asw.jms.ex1.CardOperation@469729**

---

**Рис. 7.9. Результат выполнения клиента**

Запуск сервера:

**appclient -client BillingService.jar**

Результат выполнения серверной программы приведен на Рис. 7.10.

---

**To end program, type Q or q, then <return>**
**Reading message: Card: cardNumber=1 Balance=0.0 Person=Piter CreateDate=Sun Dec 24 13:13:15 MSK 2006**
**Reading message: Card: cardNumber=2 Balance=0.0 Person=Stefan CreateDate=Sun Dec 24 13:13:15 MSK 2006**
**Reading message: Card: cardNumber=3 Balance=0.0 Person=Nataly CreateDate=Sun Dec 24 13:13:15 MSK 2006**
**Reading message: com.asw.jms.ex1.CardOperation@1dbe72f**
**Reading message: com.asw.jms.ex1.CardOperation@26efd3**
**Reading message: com.asw.jms.ex1.CardOperation@1c20611**
**Reading message: com.asw.jms.ex1.CardOperation@11c5c88**

**…**

**Reading message: com.asw.jms.ex1.CardOperation@11ce4fe**
**Reading message: com.asw.jms.ex1.CardOperation@1fe0d66**
**Card: cardNumber=3 Balance=169830.0 Person=Nataly CreateDate=Sun Dec 24 13:13:15 MSK 2006**

**Card: cardNumber=2 Balance=169497.0 Person=Stefan CreateDate=Sun Dec 24 13:13:15 MSK 2006**
**Card: cardNumber=1 Balance=170173.0 Person=Piter CreateDate=Sun Dec 24 13:13:15 MSK 2006**

**Рис. 7.10. Результат выполнения сервера**



Курс подготовлен при поддержке Sun Microsystems
Правила использования материалов опубликованы на www.sun.ru