PA 165 – Enterprise Java

Introduction to JMS

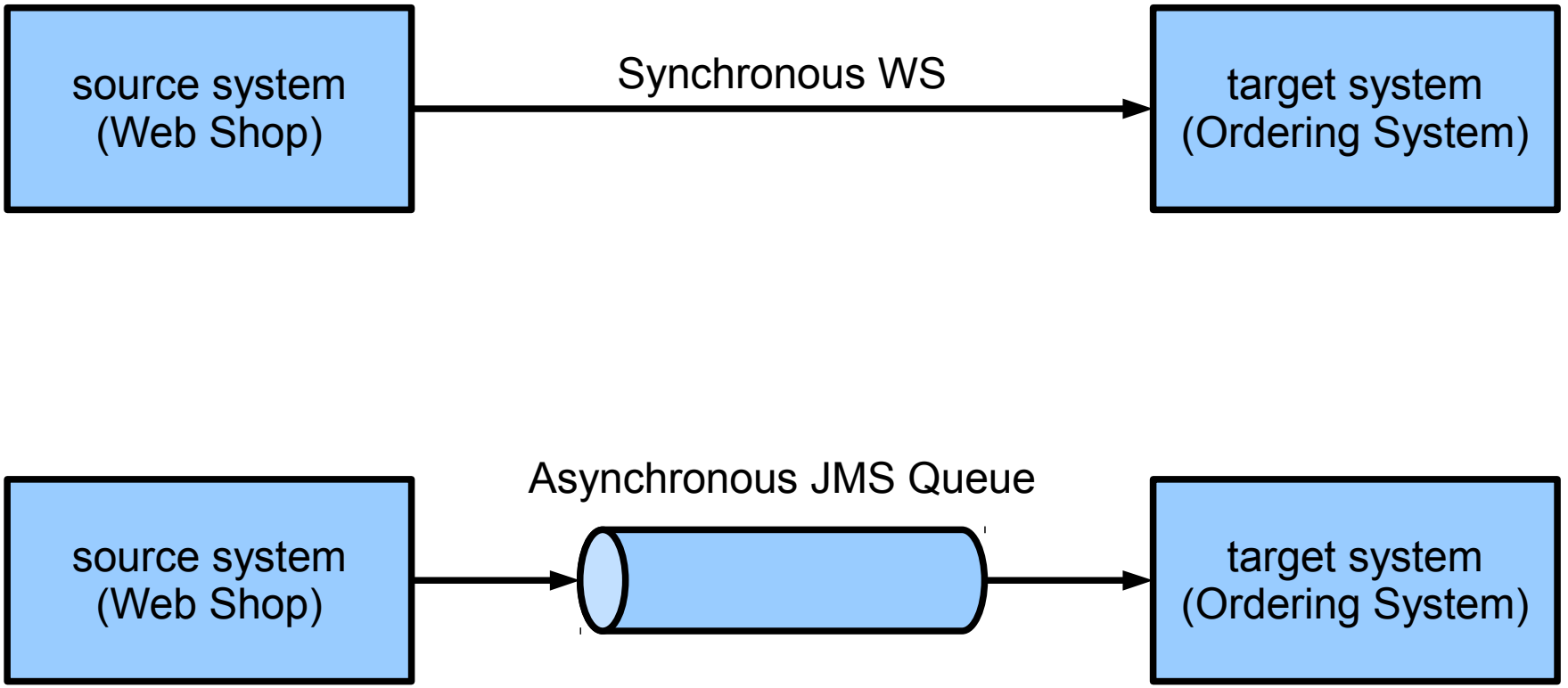
15th Dec 2015

JMS

- Java Message Service
- API for asynchronous messaging
  - JMS 1.0.2b (June 26, 2001)
  - JMS 1.1 (April 12, 2002, JSR 914, J2EE 1.4)
  - JMS 2.0 (May 21, 2013, JSR 343, Java EE 7

Why JMS

JMS versus synchronous WS



JMS versus synchronous WS

Aspect / Situation	Synchronous WS	Asynchronous JMS Queue
Operation in target system take some time	Source system is waiting for the response	Calling system does not need to wait for response
Result availability	Result is available in calling thread	Result must be sent back asynchronously
Threads utilization	Thread in source system is blocked until the operation in target system is finished	Thread in source system is block only until the message is saved
Target system is overloaded (**Scalability**)	Source system is waiting for the response, timeout can happen	Message is waiting in the queue and it is processed later
Target system is down or the network connection is not stable (**Reliability**)	Source system get timeout or error	Message is waiting in the queue and it is processed later

JMS versus synchronous WS

- Pros
  - Source system does not need to wait for response from target system.
  - Source system does not depend on availability and load of target system
  - Better performance, scalability and reliability
- Cons
  - You need MQ provider
  - Not well suitable for public internet API
  - You don't have the immediate result

Basic Rule

- Don't use synchronous WS for exchange of information which can be exchanged asynchronously
- Avoid The Hammer Syndrome – *When you hold a hammer, each problem looks like a nail.*

JMS API

JMS API

- Basic classes
  - ConnectionFactory, Destination, Message
- Classic API (JMS 1.0, JMS 1.1, JMS 2.0)
  - Connection, Session, MessageConsumer, MessageProducer
- Simplified API (JMS 2.0)
  - JMSContext, JMSConsumer, JMSProducer
- Legacy domain specific API (deprecated)
  - QueueConnectionFactory, QueueConnection, TopicConnectionFactory, QueueSession, QueueSender, QueueReceiver, TopicConnection, , TopicSession, TopicPublisher, TopicSubscriber.

Connection Factory

- Entry point for sending or receiving messages using JMS provider
- Factory for
  - Connection
  - JMS Context (JMS 2.0)
- Ususally provided as JNDI resource

Destination

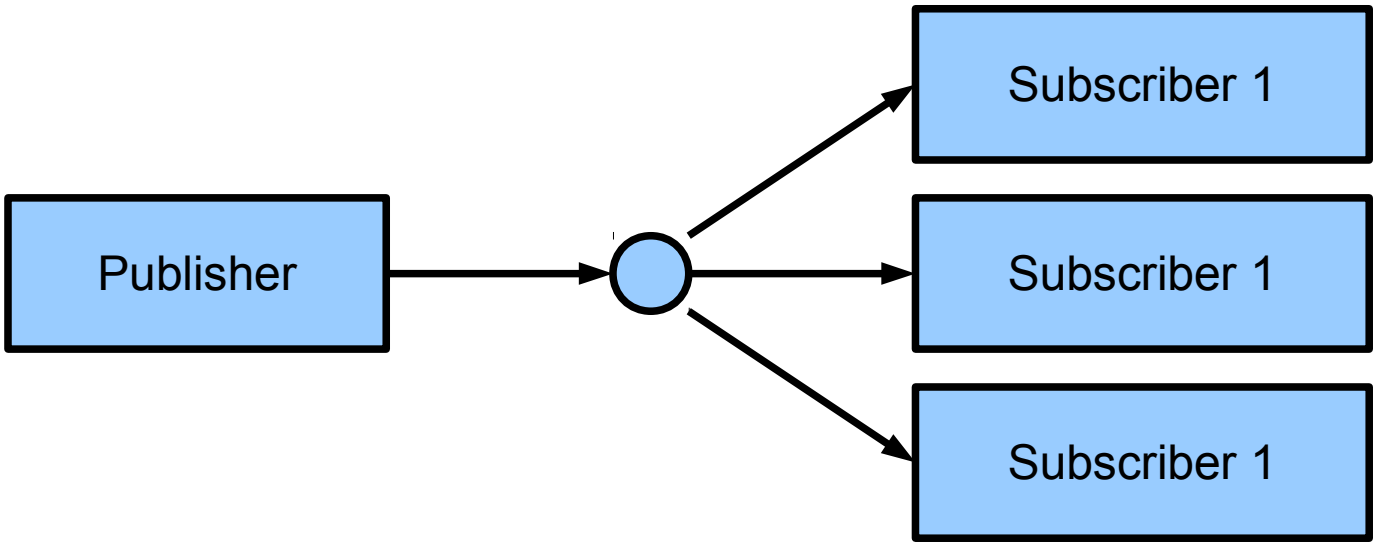
- Abstract representation of JMS channel
- Encapsulates a provider-specific address
- Standard destination types
  - Queue (point-to-point)
  - Topic (publisher-subscriber)

Queue (point-to-point)

- Each message is always delivered to single consumer
  - When there is no consumer, message is waiting in the queue
  - When there is multiple consumers, message is delivered to one (any) of them

Topic (publisher-subscriber)

- Each message is delivered to all subscribers
  - If there is no subscriber, message is dropped



How to get Destination instance

- As JNDI resource
  - standard and preffered way
- createQueue(String) or createTopic(String) of JMSContext or Session
  - Create Destination object with given name
  - Method does not create the physical queue or topic, only appropriate Destination instance!
- Create the instance directly with constructor
  - Usable for testing or simple example
  - Makes the code dependant on specific implementation, do not use in production code,

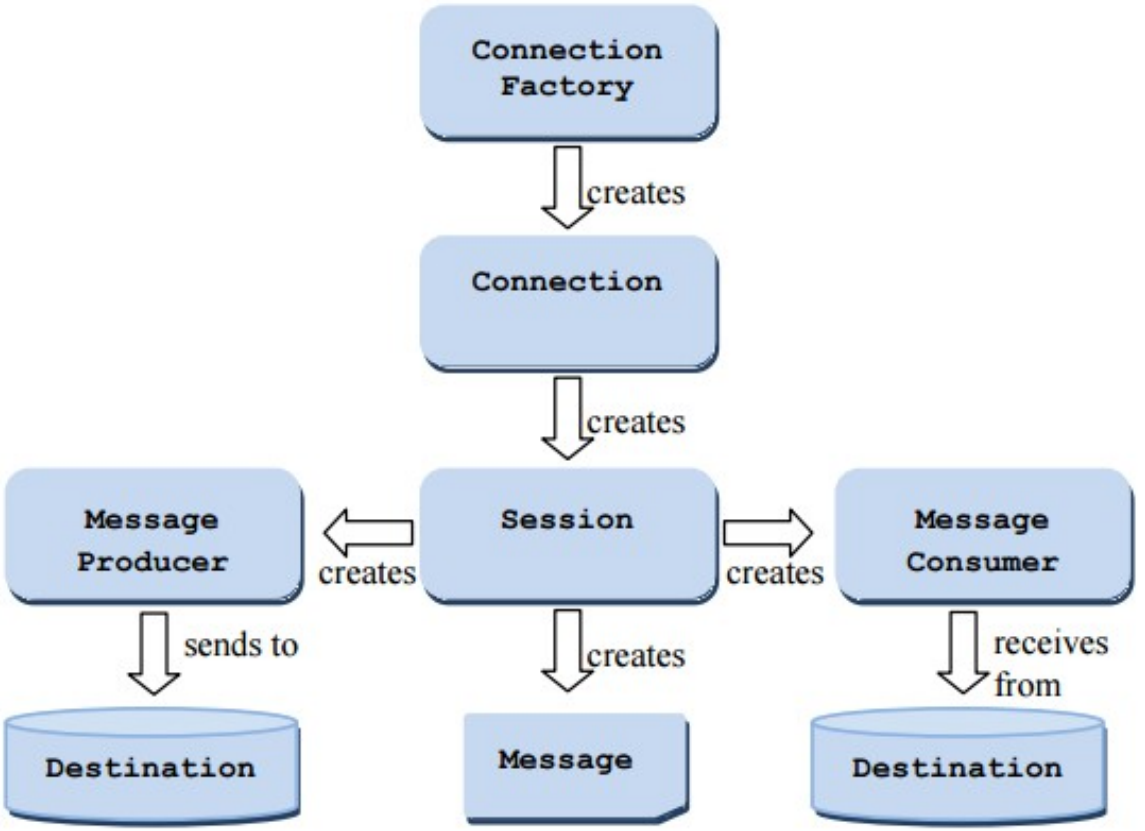
Get JMS resources with JNDI

```
// Using JNDI API

Context ctx = new InitialContext().lookup("java:/comp/env");

ConnectionFactory connectionFactory
    = (ConnectionFactory) ctx.lookup("jms/ConnectionFactory");

Queue requestQueue
    = (Queue) ctx.lookup("jms/RequestQueue");


// Using @Resource annotation

@Resource(name = "jms/ConnectionFactory")
private ConnectionFactory connectionFactory;

@Resource(name = "jms/RequestQueue")
private Queue requestQueue;
```

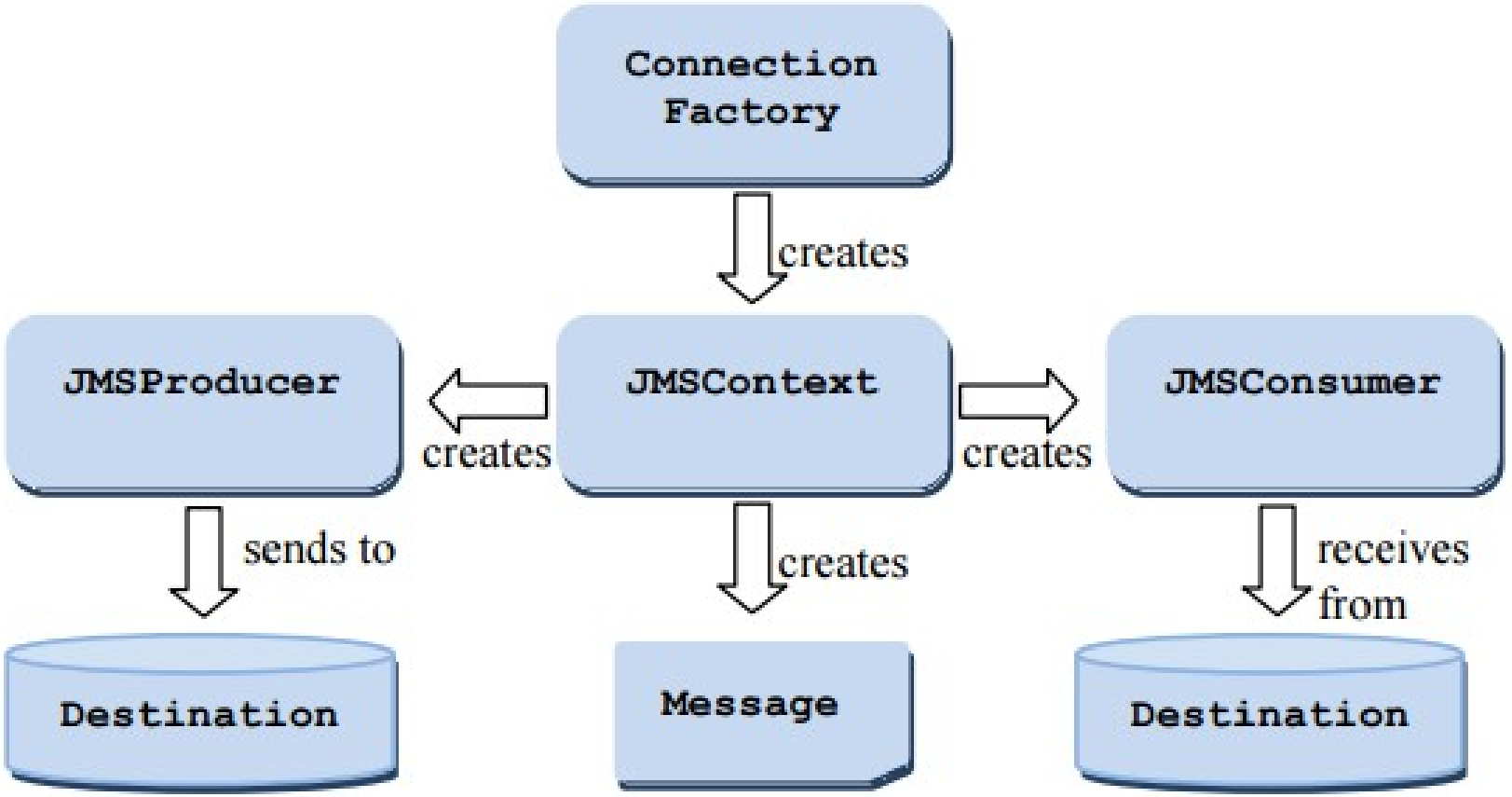
Message

- Header (defined by JMS)
  - Priority, destination, expiration, message id, delivery mode, timestamp, redelivered
- Properties (defined by application)
- Body(depends on message type)
  - StreamMessage – a stream of primitive values
  - MapMessage – a set of name-value pairs
  - TextMessage – a String
  - ObjectMessage – a Serializable Java object
  - BytesMessage – a stream of uninterpreted bytes.

Classic API


Connection Factory
creates
Connection
creates
Message Producer
creates
Session
creates
Message Consumer
sends to
creates
receives from
Destination
Message
Destination

Simplified API



Produce message (Classic API)

```
ConnectionFactory connectionFactory = ...
Destination destination = ...

Connection connection = connectionFactory.createConnection();
Session session = connection.createSession();
MessageProducer producer = session.createProducer(destination);

Message message = session.createTextMessage("MSG");

// Send the message. When the call is finisged,
// message is safely passed for delivery
producer.send(message);

// Don't forget to release all resources
// If you are using JMS 2.0, you can use try-with-resources
messageProducer.close();
session.close();
connection.close();
```

Produce message (Simplified API)

```
ConnectionFactory connectionFactory = ...
Destination destination = ...

try (JMSContext context = connectionFactory.createContext()) {

    // producer is lightweight object without close() method
    JMSProducer producer = context.createProducer();

    Message message = context.createTextMessage("Message Body");

    // Send the message. When the call is finisged,
    // message is safely passed for delivery
    producer.send(destination, message);

}
```

Consume message (Classic API)

```
ConnectionFactory connectionFactory = ...
Destination destination = ...

Connection connection = connectionFactory.createConnection();
Session session = connection.createSession();
MessageConsumer consumer = session.createConsumer(destination);

// Start to receive messages
connection.start();

// Get the next message from the queue. If the queue is empty,
// wait for the next message max 1000 milliseconds
Message message = consumer.receive(1000);

// Process the message
System.out.println(message);

// Don't forget to release all resources
// If you are using JMS 2.0, you can use try-with-resources
messageProducer.close();
session.close();
connection.close();
```

Consume message (Simplified API)

```
ConnectionFactory connectionFactory = ...
Destination destination = ...

try (JMSContext context = connectionFactory.createContext();
     JMSConsumer consumer = context.createConsumer(destination)) {

     // consumer is not lightweight object, close() must be called

     // Start to receive messages
     context.start();

     // Get the next message from the queue. If the queue is empty,
     // wait for the next message max 1000 milliseconds
     Message message = consumer.receive(1000);

     // Process the message
     System.out.println(message);

}
```

Asynchronous Message Handling

Message Consuming

- Synchronous message consuming
  - Client has to call receive method (examples above)
- Asynchronous message consuming
  - Incoming message is automatically processed with MessageListener, registered at appropriate MesageConsumer or JMSConsumer
- EJB Message Driven Bean
  - Another way of asynchronous message processing
  - Suggested way of message processing (if it is possible to use EJB)

Consume message (Simplified API)

```
ConnectionFactory connectionFactory = ...
Destination destination = ...

MessageListener messageListener = (Message message) -> {
    // Process the message
    System.out.println(message);
};

try (JMSContext context = connectionFactory.createContext();
    JMSConsumer consumer = context.createConsumer(destination)) {

    consumer.setMessageListener(messageListener);

    // Start to receive messages
    context.start();

}
```

Message Producing

- Synchronous
  -
- Asynchronous message sending (JMS 2.0)
  - Client is not waiting until the message is safely passed for delivery, but the send method returns imidiatelly and message is sent in background thread.

Delivery Mode

- Defines how reliable way will be the message delivered
  - PERSISTENT – Message will be stored to persistent storage to guarantee that it will not be lost in case of failure. This is the default mode.
  - NON_PERSISTENT – Message is held only in memory, less overhead, but message can be lost.

Must be set as parameter of MessageProducer.send(...) method or with MessageProducer.setDeliveryMode(int) method. Message.setJMSDeliveryMode(int) will not work (see Javadoc for explanation)!

Set DeliveryMode (Simplified API)

```
ConnectionFactory connectionFactory = ...
Destination destination = ...

try (JMSContext context = connectionFactory.createContext()) {

    JMSProducer producer = context.createProducer();

    // This is the right way how to set delivery mode
    producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

    Message message = context.createTextMessage("Message Body");

    // This will not work!
    message.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

    producer.send(destination, message);

}
```

Concurrecy in JMS

Concurrency support in JMS

Class in classic API	Class in simplified API	Supports concurrent use	Creating costs
Destination		yes	expensive
ConnectionFactory		yes	expensive
Connection	N/A	yes	expensive
Session	JMSContext	no	cheap
MessageProducer	JMSProducer	no	cheap
MessageConsumer	JMSConsumer	no	cheap