JAVA MESSAGING SYSTEM (JMS)

CONTENTS

WHAT IS MESSAGING?

- Method of communication between software components or applications

WHY MESSAGING?

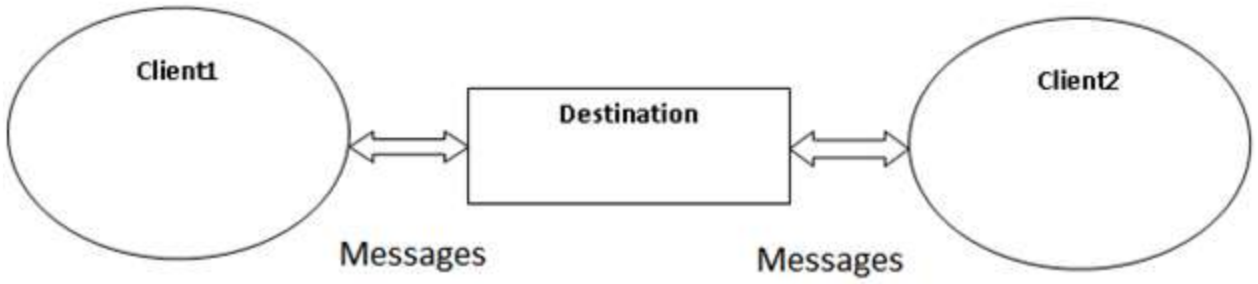
- Messaging enables distributed communication that is *loosely coupled*

- Messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application's methods

WHAT IS JMS

- The Java Message Service is a Java API that allows applications to create, send, receive, and read messages
- JMS API defines a common set of interfaces

JMS



JMS API COMPONENTS

- Provider
- Client
- Message
- Administered Objects

JMS API COMPONENTS

- A ***JMS provider*** is a messaging system that implements the JMS interfaces and provides administrative and control features
- ***JMS clients*** are the programs or components, written in the Java programming language, that produce and consume messages

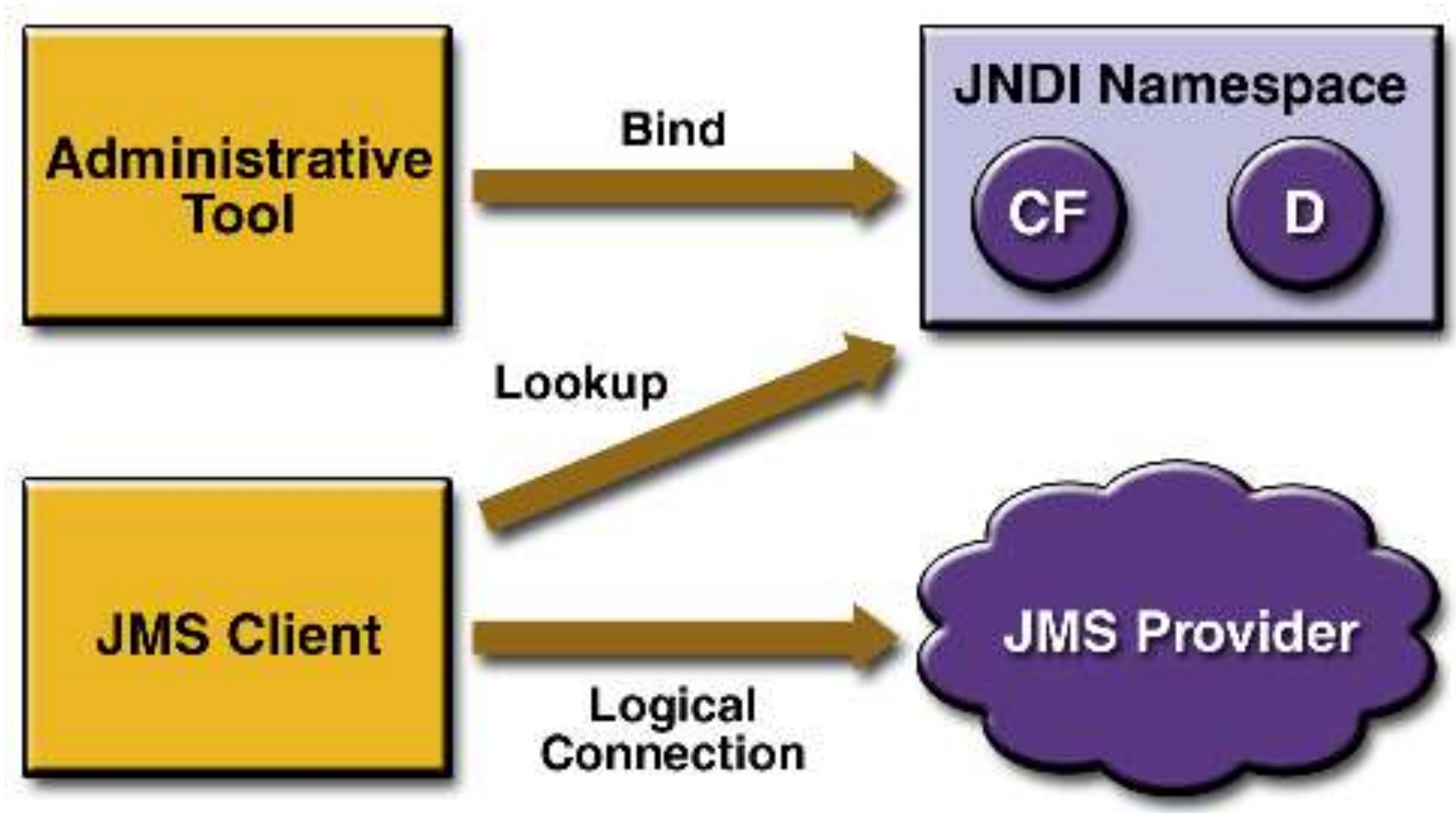
JMS API COMPONENTS

- ***Messages*** are the objects that communicate information between JMS clients
- Various types of messages exists
- Ex. TextMessage, ObjectMessage, etc

JMS API COMPONENTS

- ***Administered objects:*** Preconfigured JMS objects created by an administrator for the use of clients. (Ex Destinations & Connection factories)

JMS API COMPONENTS



MESSAGING DOMAIN

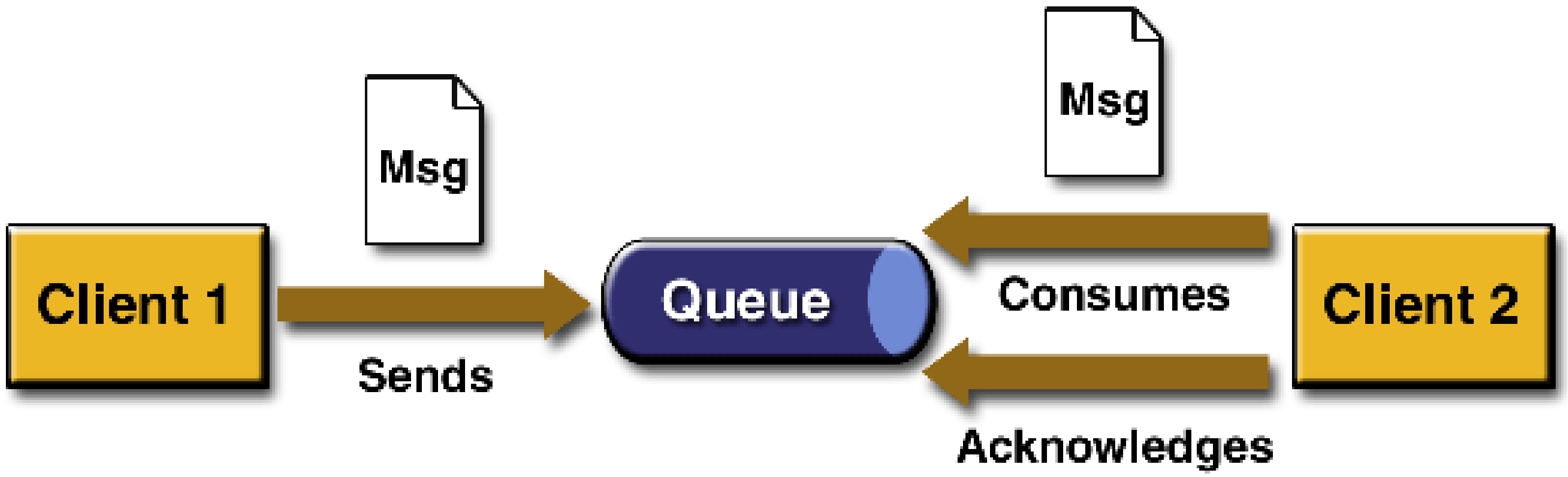
- Point to Point (Queue)
  - Each message is addressed to a specific queue
  - Receiving clients extract messages from the queue(s) established to hold their messages
  - One to one

- Publish / Subscribe
  - Each message is addressed to a specific topic
  - Subscribers extract messages from the topic(s)
  - One to many

POINT TO POINT

- Each message has only one consumer
- A sender and a receiver of a message have no timing dependencies
- The receiver can fetch the message whether or not it was running when the client sent the message
- The receiver acknowledges the successful processing of a message

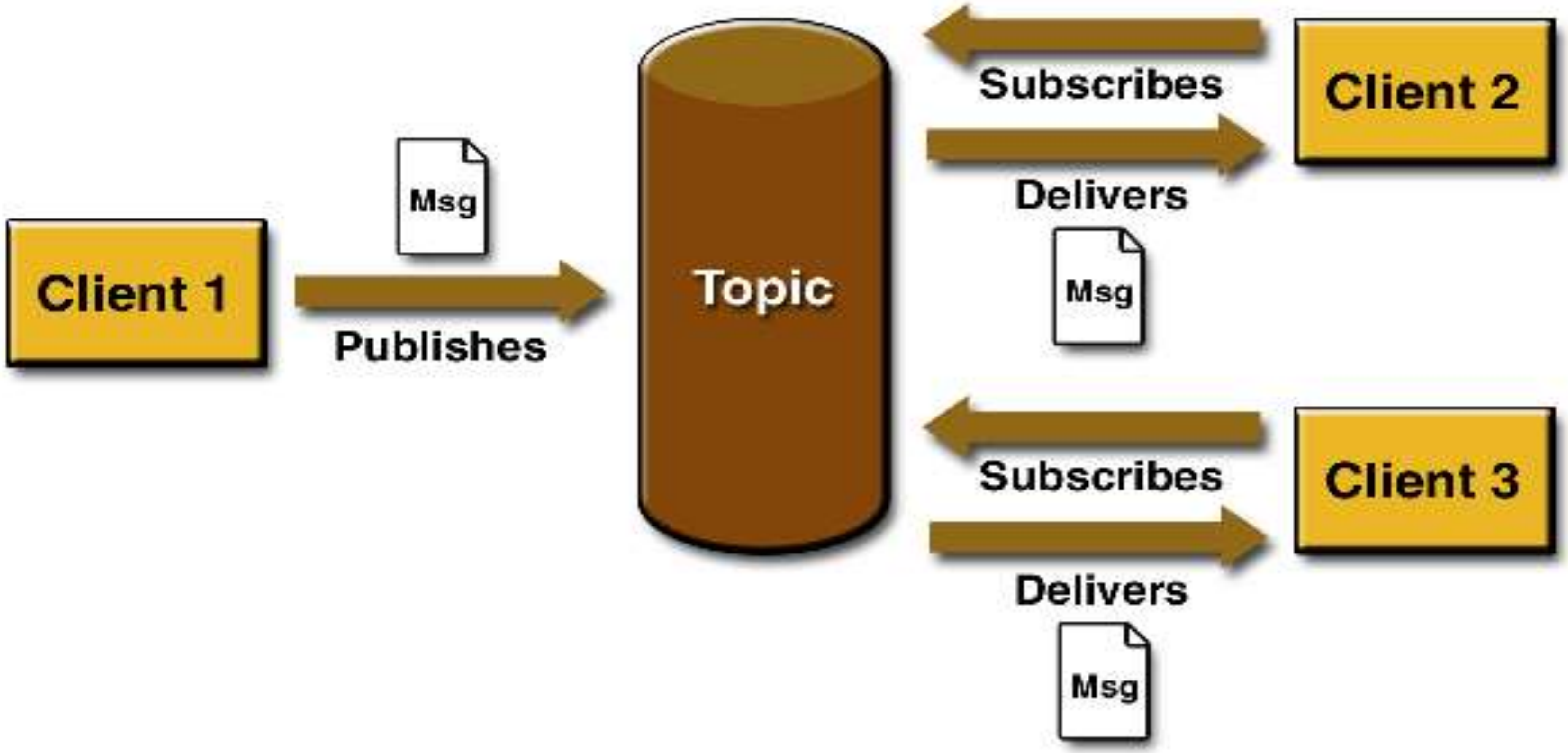
POINT TO POINT



PUBLISH/ SUBSCRIBE

- Each message may have multiple consumers
- Publishers and subscribers have a timing dependency
- A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages
- Only Durable subscriptions can receive messages sent while the subscribers are not active

PUBLISH / SUBSCRIBE



WHEN TO USE WHAT?

- Use PTP messaging when every message you send must be processed successfully only once by one and only consumer

- Use pub/sub messaging when each message can be processed by zero, one, or many consumers

MESSAGE CONSUMPTION

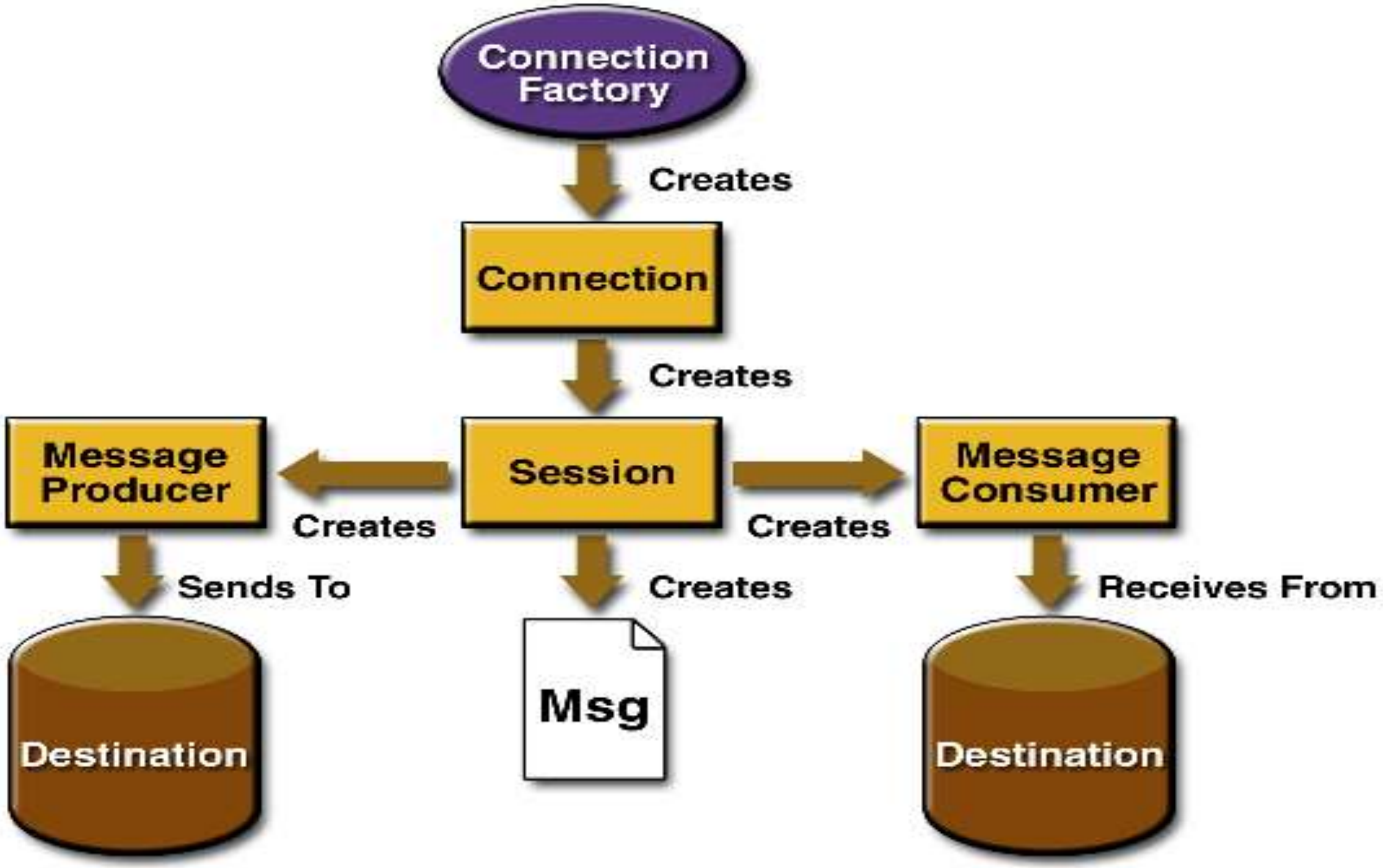
- Messages can be consumed in either of two way
  - **Synchronously.** A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method
  - **Asynchronously.** A client can register a *message listener* with a consumer

LAB

JMS API PROGRAMMING MODEL

- Administered Objects
  - Connection Factories
  - Destinations
- Connections
- Sessions
- Message Producers
- Message Consumers
  - Message Listeners
  - Message Selectors
- Messages
  - Message Headers
  - Message Properties
  - Message Bodies
- Exception Handling

PROGRAMMING MODEL



ADMINISTERED OBJECTS

- Two parts of a JMS application--destinations and connection factories--are best maintained administratively rather than programmatically

- Why administered?
  - The technology underlying these objects is likely to be very different from one implementation of the JMS API to another. Therefore, easy to manage through administrative tasks.

CONNECTION FACTORY

- A *connection factory* is the object a client uses to create a connection with a provider
- A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator
- Each connection factory is an instance of either the QueueConnectionFactory or the TopicConnectionFactory interface

SAMPLE CODE IN JEE ENV

- Context ctx = new InitialContext();
- QueueConnectionFactory queueConnectionFactory = (QueueConnectionFactory) ctx.lookup("QueueConnectionFactory");
- TopicConnectionFactory topicConnectionFactory = (TopicConnectionFactory) ctx.lookup("TopicConnectionFactory");

CODE IN STANDALONE

- **String connectionUri = "tcp://localhost:61616";**

- **ActiveMQConnectionFactory connectionFactory;**
- connectionFactory = **new**
- ActiveMQConnectionFactory(connectionUri);

DESTINATION

- A *destination* is the object a client uses to specify the target of messages it produces and the source of messages it consumes

- In the PTP messaging domain, destinations are called queues

- In the pub/sub messaging domain, destinations are called topics

- A JMS application may use multiple queues and/or topics.\

SAMPLE CODE

- Queue myQueue = (Queue) ctx.lookup("MyQueue");

- Topic myTopic = (Topic) ctx.lookup("MyTopic");

- In standalone,

Destination q= session.createQueue("testqueue1");

Destination t= session.createTopic("EVENTS.QUOTES");

CONNECTIONS

- A *connection* encapsulates a virtual connection with a JMS provider
- A connection could represent an open TCP/IP socket between a client and a provider service daemon
- You use a connection to create one or more sessions

SAMPLE CODE

- QueueConnection queueConnection = queueConnectionFactory.createQueueConnection();
- queueConnection.close();

- TopicConnection topicConnection = topicConnectionFactory.createTopicConnection();
- topicConnection.close();

SESSION

- A *session* is a single-threaded context for producing and consuming messages
- Use sessions to create message producers, message consumers, and messages
- Sessions serialize the execution of message listeners
- A session provides a transactional context with which to group a set of sends and receives into an atomic unit of work

SAMPLE CODE

- QueueSession queueSession = queueConnection.createQueueSession(true, 0);

- TopicSession topicSession = topicConnection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);

  Note:
  - The first boolean argument indicates that the session is transacted or not
  - the second indicates that message acknowledgment type 0 means “acknowledgment not specified for transacted sessions”

MESSAGE PRODUCER

- A *message producer* is an object created by a session and is used for sending messages to a destination.
- The PTP form of a message producer implements the QueueSender interface.
- The pub/sub form implements the TopicPublisher interface.

SAMPLE CODE

- QueueSender queueSender = queueSession.createSender(myQueue);
- queueSender.send(message);

- TopicPublisher topicPublisher = topicSession.createPublisher(myTopic);
- topicPublisher.publish(message);

MESSAGE CONSUMER

- A *message consumer* is an object created by a session and is used for receiving messages sent to a destination
- A message consumer allows a JMS client to register interest in a destination with a JMS provider
- The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination

SAMPLE CODE

- QueueReceiver queueReceiver = queueSession.createReceiver(myQueue);
- queueConnection.start();
- Message m = queueReceiver.receive();

SAMPLE CODE

- TopicSubscriber topicSubscriber = topicSession.createSubscriber(myTopic);
- topicConnection.start();
- Message m = topicSubscriber.receive(1000);
  - // time out after a second
- TopicSubscriber topicSubscriber = topicSession.createDurableSubscriber(myTopic);

MESSAGE LISTENER

- A *message listener* is an object that acts as an asynchronous event handler for messages.
- This object implements the MessageListener interface, which contains one method, onMessage.
- In the onMessage method, you define the actions to be taken when a message arrives.

SAMPLE CODE

- TopicListener topicListener = new TopicListener();

- topicSubscriber.setMessageListener(topicListener);

- topicConnection.start();

MESSAGE SELECTOR

- If your messaging application needs to filter the messages it receives, you can use a JMS API message selector, which allows a message consumer to specify the messages it is interested in.
- Message selectors assign the work of filtering messages to the JMS provider rather than to the application
- A message selector is a String that contains an expression that can be passed as argument to message consumer creation methods

SAMPLE CODE

- session.createReceiver(responseQueue, “color = ‘GREEN’”);

- The message consumer then receives only messages whose headers and properties match the selector i.e. color = ‘GREEN’

MESSAGES

A JMS message has three parts:

- A header
- Properties
- A body

MESSAGE HEADER

- Header contains a number of predefined fields that contain values that both clients and providers use to identify and to route messages

HEADER FIELDS

Header Field	Set By
JMSDestination	send or publish method
JMSDeliveryMode	send or publish method
JMSExpiration	send or publish method
JMSPriority	send or publish method
JMSMessageID	send or publish method
JMSTimestamp	send or publish method
JMSCorrelationID	Client
JMSReplyTo	Client
JMSType	Client
JMSRedelivered	JMS provider

MESSAGE PROPERTIES

- Can be used if you need values in addition to those provided by the header fields.
- You can use properties to provide compatibility with other messaging systems, or you can use them to create message selectors

MESSAGE BODY/ MESSAGE TYPES

Message Type	Body Contains
TextMessage	A java.lang.String object (for example, the contents of an Extensible Markup Language file).
MapMessage	A set of name/value pairs, with names as String objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
BytesMessage	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
StreamMessage	A stream of primitive values in the Java programming language, filled and read sequentially.
ObjectMessage	A Serializable object in the Java programming language.
Message	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

SAMPLE CODE - SENDER

```
TextMessage message =
   queueSession.createTextMessage();

message.setText(msg_text);      // msg_text is
   a String

queueSender.send(message);
```

SAMPLE CODE - RECEIVER

```
Message m = queueReceiver.receive();
if (m instanceof TextMessage) {
   TextMessage message = (TextMessage) m;
   System.out.println("Reading message: " +
         message.getText());
}
else {
   // Handle error
}
```

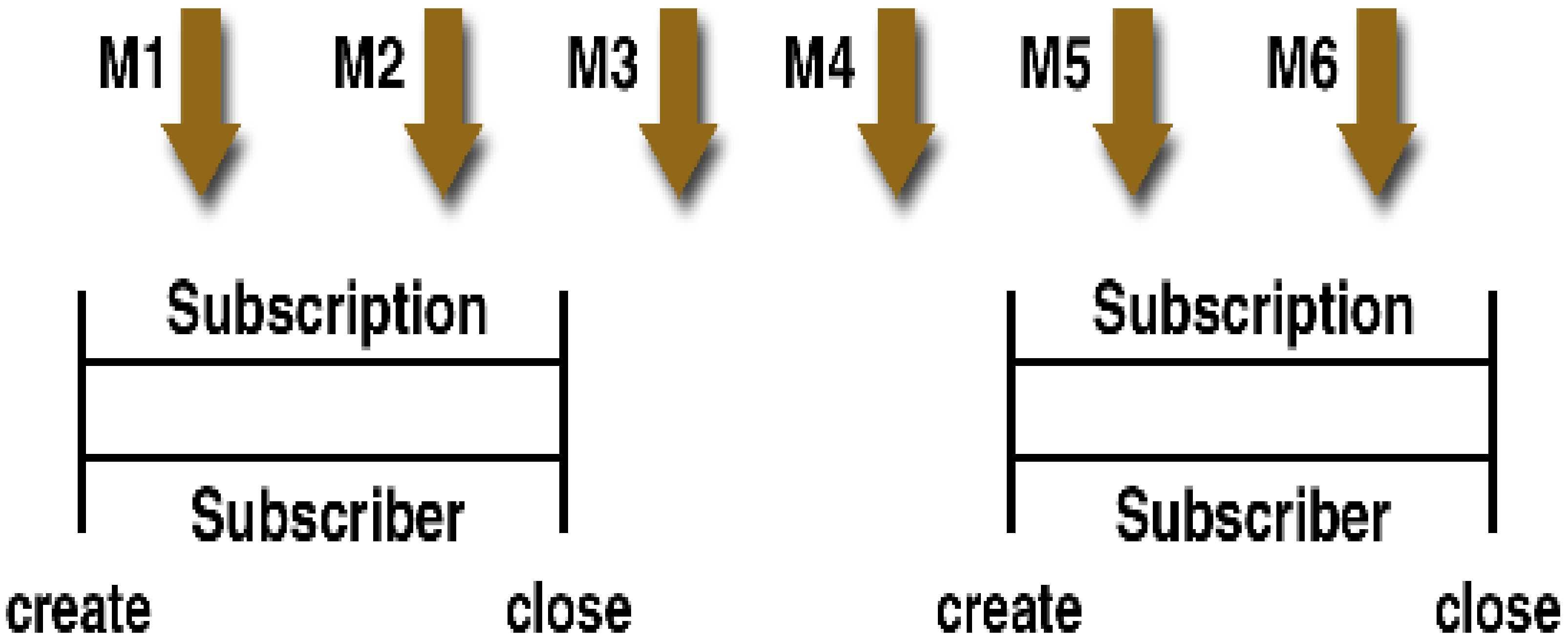
EXCEPTION HANDLING

- The root class for exceptions thrown by JMS API methods is JMSException.
- Catching JMSException provides a generic way of handling all exceptions related to the JMS API.

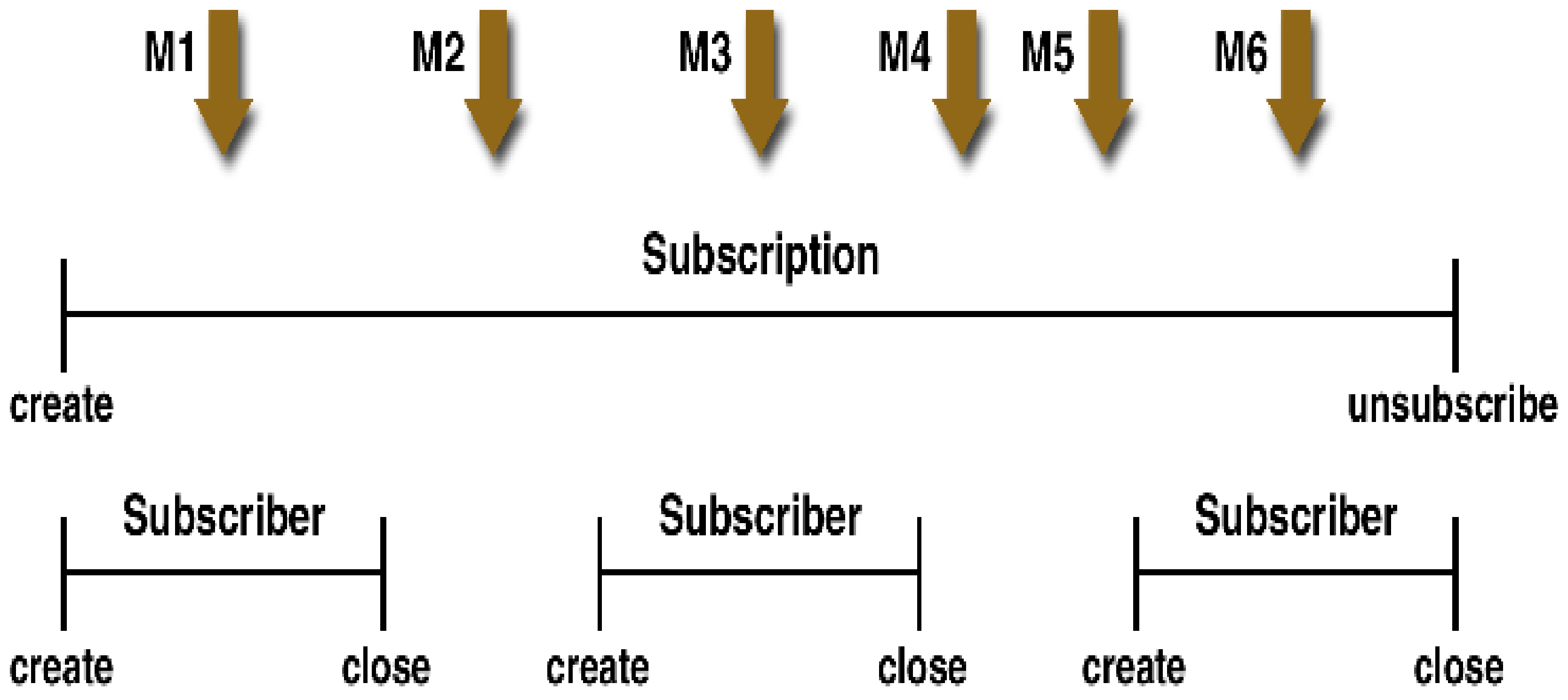
RELIABILITY

- Durable subscriptions
- Local transactions

NON DURABLE SUBSCRIPTION



DURABLE SUBSCRIPTION



DURABLE SUBSCRIPTION

- Establish the unique identity of a durable subscriber by setting the following:

  - A client ID for the connection
  - A topic and a subscription name for the subscriber

  - TopicSubscriber topicSubscriber = topicSession.createDurableSubscriber(myTopic, “MySubscription”);

LAB