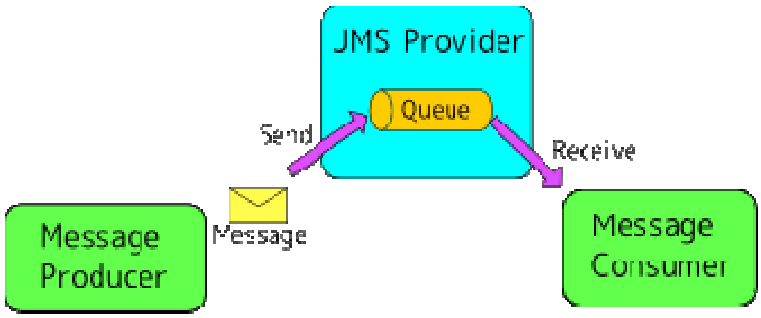
# Java Message Service with Apache ActiveMQ

- JMS let's you send messages containing data (for example a String, array of byte or a serializable Java object, from one program to another.
- It doesn't use a direct connection from Program A to Program B, instead the message is sent to a JMS provider and put there in a Queue where it waits until the other program receives it.
- The Java Message Service (JMS) API is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients. It is a Java API that allows application to create, send, receive, and read messages.
- The JMS API enables communication that is loosely coupled, asynchronous and reliable.




**Note:**

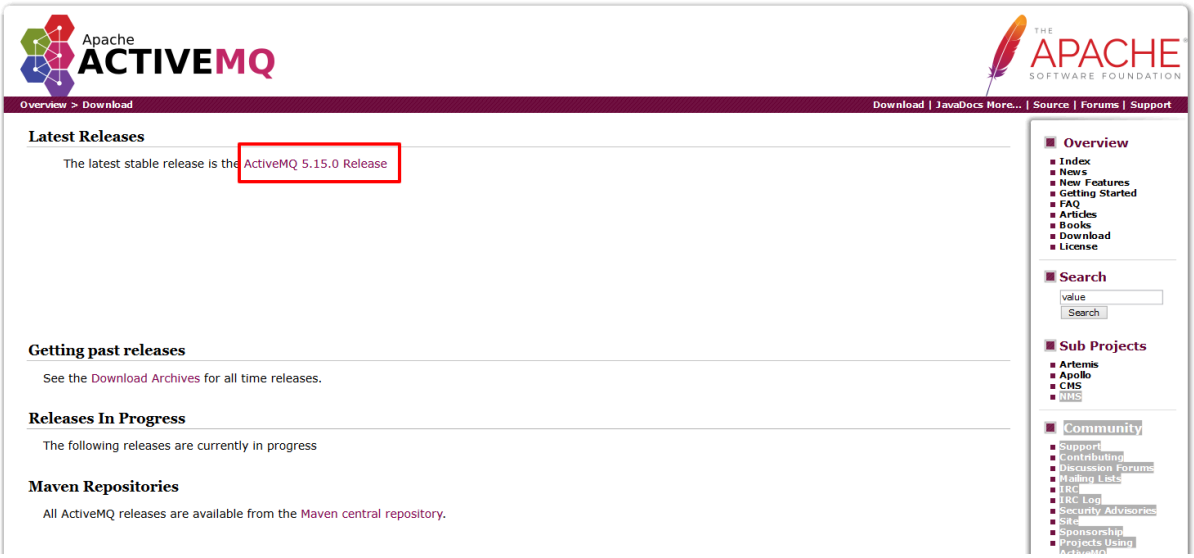
Before Working with JMS We need to install a JMS Provider to work with a JMS.

## Installing a JMS Provider (Apache ActiveMQ)

1. **Get the download link:**

activemq.apache.org/download.html

2. **Click on the Latest release Hyperlink to Download:**




3. **You'll get a zip file after downloading, extract the zip file to your preferred location.**
4. **NOTE: ( On Windows )**
   - **ActiveMQ requires Java 7 to run and to build**
   - **The JAVA_HOME environment variable must be set to the directory where the JDK is installed, e.g., `c:\Program Files\jdk.1.7.0_xx_xx`.**
   - **Maven 3.0 or greater (required when installing source or developer's releases).**
   - **JARs that will be used must be added to the classpath.**
5. **To run Apache ActiveMQ:**

### On Windows

From a console window, change to the installation directory and run `ActiveMQ`:

```
cd [activemq_install_dir]
```

where `activemq_install_dir` is the directory in which ActiveMQ was installed, e.g., `c:\Program Files\ActiveMQ-5.x`.

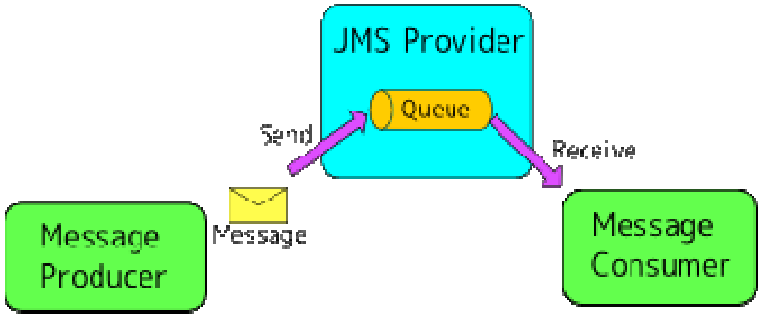
Then type (depending on ActiveMQ version):

**ActiveMQ 5.10 onwards**

```
bin\activemq start
```

**ActiveMQ 5.9 or older**

```
bin\activemq
```




- Here MessageProducer is a Java Program sending a JMS Message to a Queue on the JMS Provider.
- MessageConsumer is another program which receives that message.
- These two programs can run on separate machines and all they have to know to communicate is the URL of the JMS Provider.
- The Provider can be for example a Java EE server, like JBoss or Glassfish.
- We will use ActiveMQ which is lightweight and easy to use.

## Maven POM

```
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-all</artifactId>
    <version>${activemq.version}</version>
</dependency>
```

Once you have Apache ActiveMQ up and running, let's write our message producer and consumer programs.:

## Producer Program:

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.apache.activemq.ActiveMQConnection;
import org.apache.activemq.ActiveMQConnectionFactory;

public class Producer {

	/*
	 *  URL of the JMS server. DEFAULT_BROKER_URL will just mean
	 *  that JMS server is on localhost
	 */
	private static String url = "tcp://localhost:61616";
			//ActiveMQConnection.DEFAULT_BROKER_URL;
	// default broker URL is : tcp://localhost:61616"

	private static String subject = "praveen"; //Queue Name
	// You can create any/many queue names as per your requirement.

	public static void main(String[] args) throws JMSException {

		/*
		 *  Getting JMS connection from the server and starting it
		 */
		ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(url);
		Connection connection = connectionFactory.createConnection();
		connection.start();

		/*
		 * JMS messages are sent and received using a Session. We will
		 * create here a non-transactional session object. If you want
		 * to use transactions you should set the first parameter to 'true'
		 */
		Session session =   connection
		                    .createSession(false,Session.AUTO_ACKNOWLEDGE);

		/*
		 * Destination represents here our queue 'praveen' on the
		 * JMS server. You don't have to do anything special on the
		 * server to create it, it will be created automatically.
		 */
		Destination destination = session.createQueue(subject);
```

```
            /*
             *  MessageProducer is used for sending messages (as opposed
             *  to MessageConsumer which is used for receiving them)
             */
            MessageProducer producer = session.createProducer(destination);

            /*
             *  We will send a small text message saying 'Hello' in Japanese
             */
            TextMessage message = session
                        .createTextMessage("Hello World from ActiveMQ!");

            /*
             * Here we are sending the message!
             */
            producer.send(message);
            System.out.println("Message Sent: " + message.getText());

            connection.close();
        }
}
```

- There is a lot going on here. The javax.jms.Connection represents out connection with the JMS Provider – ActiveMQ. (Not to be confused with SQL's Connection )
- "Destination" represents the Queue on the JMS Provider that we will be sending messaged to. In our case, we will send it to Queue called 'praveen' it will be automatically created if it didn't exist yet).
- Note that there is no mention who will finally read the message. Actually, the Producer does not know where or who the consumer is. We are just sending a message into queue 'praveen' and what happens from there to the sent messages is not of Producer's interest any more.

**Consumer Program:**

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.apache.activemq.ActiveMQConnectionFactory;

public class Consumer {
        // URL of the JMS server
        private static String url = "tcp://localhost:61616";
                        //ActiveMQConnection.DEFAULT_BROKER_URL;
        // default broker URL is : tcp://localhost:61616"
```

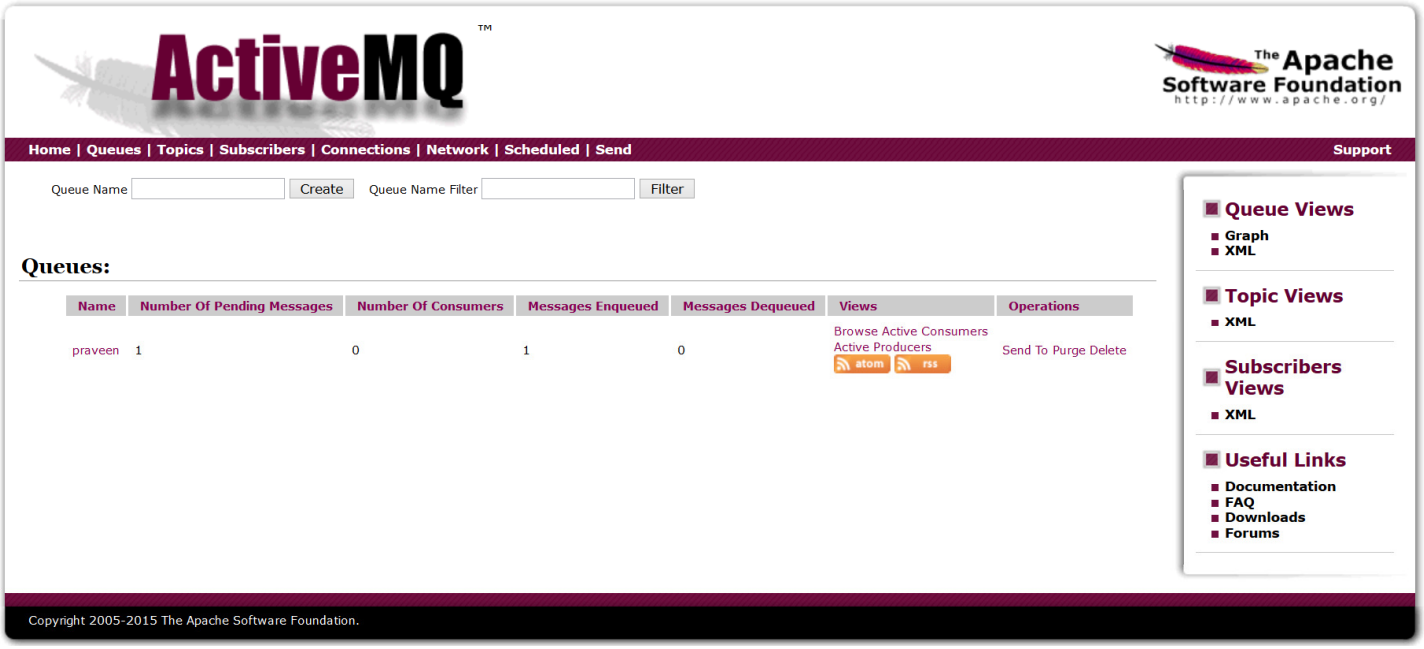
```
        // Name of the queue we will receive messages from
        private static String subject = "praveen";

        public static void main(String[] args) throws Exception {
                // Getting JMS connection from the server
                ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(url);
                Connection connection = connectionFactory.createConnection();
                connection.start();



                // Creating session for sending messages
                Session session = connection.createSession(false,
                                Session.AUTO_ACKNOWLEDGE);

                // Getting the queue
                Destination destination = session.createQueue(subject);

                // MessageConsumer is used for receiving (consuming) messages
                MessageConsumer consumer = session.createConsumer(destination);

                /*
                 *  Here we receive the message.
                 *  By default this call is blocking, which means it will wait
                 *  for a message to arrive on the queue.
                 */
                Message message = consumer.receive();

                /*
                 *  There are many types of Message and TextMessage
                 *  is just one of them. Producer sent us a TextMessage
                 *  so we must cast to it to get access to its .getText() method
                 */
                if (message instanceof TextMessage) {
                        TextMessage textMessage = (TextMessage) message;
                        System.out.println("Receive Message: " + textMessage.getText());
                }
                connection.close();
        }
}
```

- The consumer codes are pretty similar to the Producer's code before. Only few things are different, Instead of creating MessageProducer we are creating MessageConsumer and then use it's .receive() method instead of .send().
- You can see a downcast form Message to TextMessage but there is nothing we could do about it, because .receive() method just returns interface message (TextMessage interface extends Message) and there are no separate methods for receiving just TextMessage's.

**Compile and run both programs and remember to add ActiveMQ's JAR file to the classpath. Before running them be also sure the ctiveMQ's instance is running (for example in a separate terminal) .**

**If You see the code compiled successfully, then run the Producer Program and check you ActiveMQ console in your Browser:**

http://localhost:8161/admin/queues.jsp

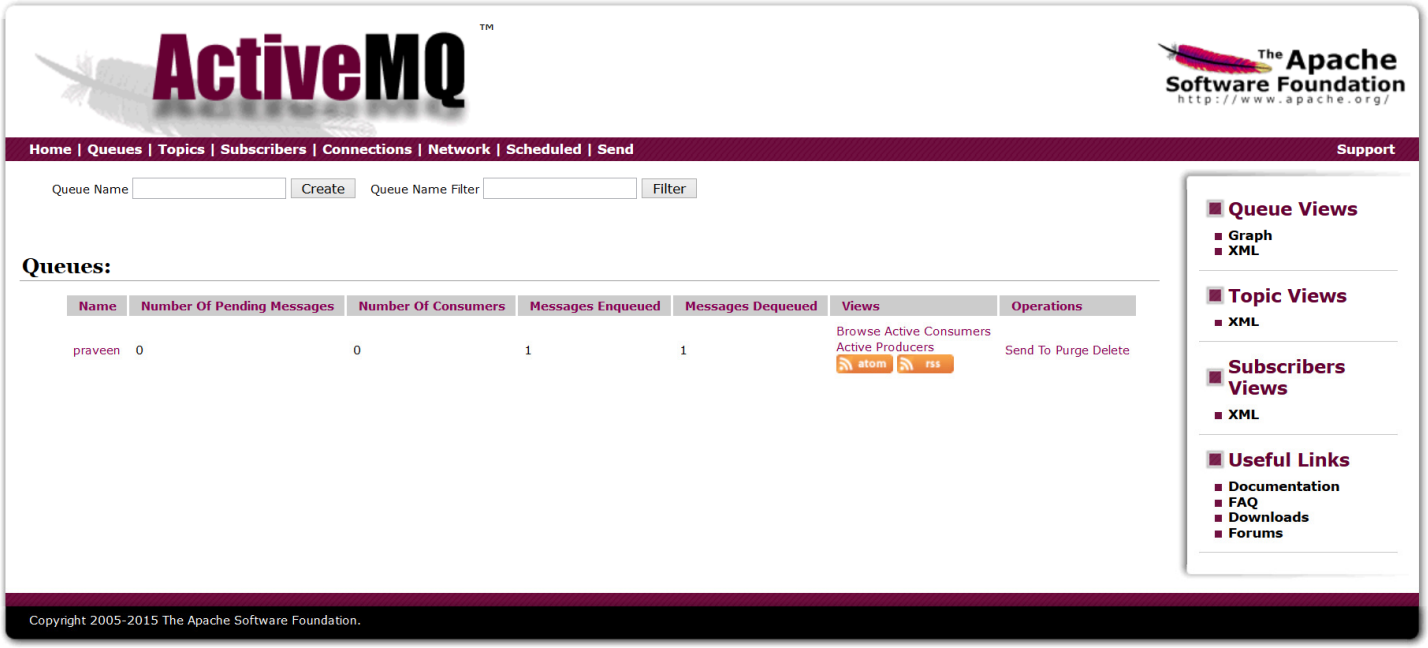



You will see that one message is in the queue by name ‘praveen’

Now run the **Consume Program**

After running the consume program you will see a message in the console which will say the Message is received:

And then check the ActiveMQ Console in the Browser:




If You are getting the proper output in the console (similar to above) then everything went ok. The message successfully received.

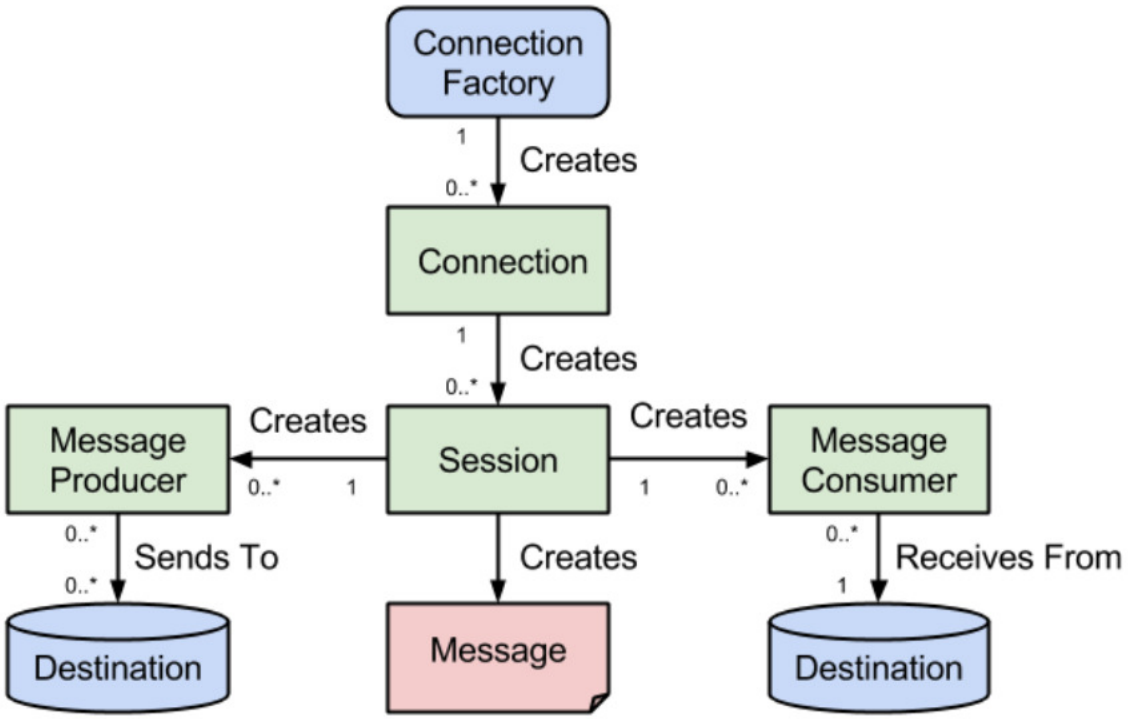
## Why JMS?

- Communication using JMS is **asynchronous.** The producer just sends a message and goes on with his business. If you called a method using RMI you would have to wait until it returns, and there can be cases you just don't want to lose all that time.
- Take a look at how we run the example above. At the moment we run the producer to send a message, there was yet no instance of Consumer running. The message was delivered at the moment the consumer asked ActiveMQ for it. Now compare it to RMI. If we tried to send any request through RMI to a server that is not running yet, we would get a RemoteException.
- Using JMS let's you send requests to services that may be currently unavailable. The message will be delivered as soon as the service starts.
- JMS is a way of abstracting the process of sending messages. As you can see in the examples above, we are using some custom Queue with name 'praveen'. The producer just know it has to send to that queue and the

consumer takes it from there. Thanks to that we can decouple the producer from the consumer.

- When a message gets into a queue, we can do a full bunch of stuff with it, like sending it to other queues, copying it, saving in a database, routing based on its contents and much more.

# The JMS API Programming Model




- The basic building of blocks of the JMS API programming model is shown above. At the top we have the *ConnectionFactory* object which is the object a client uses to create a connection to a JMS provider. A connection factory encapsulates a set of connection configuration parameters like of example the broker URL. A connection factory is a JMS administered object that is typically created by an administrator and later used by JMS clients.
- When you have a *ConnectionFactory* object, you can use it to create a connection. A *Connection* object encapsulates a virtual connection with a

JMS provider. For example, a connection could represent an open TCP/IP socket between a client and a provider service daemon. Before an application completes, it must close any connections that were created. Failure to close a connection can cause resources not to be released by the JMS provider.

- Closing a connection also closes its sessions and their message producers/message consumers.

- A session is a single-threaded context for producing and consuming messages. A session provides a transactional context with which to group a set of sends and receives into an atomic unit of work.

  **Note: - As mentioned above, it is important to note that everything from a session down is single threaded!**

- A ***MessageProducer*** is an object that is created by a session and used for sending messages to a destination. You use a Session object to create a message producer for a destination.
  **Note:** It is possible to create an unidentified producer by specifying a null *Destination* as argument to the *createProducer()* method. When sending a message, overload the send method with the needed destination as the first parameter.
- A ***MessageConsumer*** is an object that is created by a session and used for receiving messages sent to a destination. After you have created a message consumer it becomes active, and you can use it to receive messages. Message delivery does not begin until you start the connection you created by calling its start() method.
  **Note:** Remember to always call the ***start()*** *method* on the Connection object in order to receive messages!
- A ***Destination*** is the object a client uses to specify the target of messages it produces and the sources of messages it consumes. In point-to-point messaging domain, destinations are called queues. **In publish/subscribe messaging domain, destinations are called topics.**

- The ***create( )*** method will first create an instance of the ***ConnectionFactory*** which is in turn used to create a connection to ActiveMQ using the default broker URL.
- Using the ***Connection*** instance, a ***Session*** is created which is used to create a ***Destination*** and ***MessageProducer*** objects are automatically closed when calling this method.
- The class also contains a ***close()*** method which allows to correctly release the resources at the JMS string.Using the session a JMS ***TextMessage*** is created on which the string is set. Using the message producer the message is sent to the JMS provider.
- For receiving messages, a ***Consumer*** class is defined which has the same ***create()*** and ***close()*** methods.
- The main difference is that in the case of a message consumer the connection is started.
- A timeout parameter is passed to the **receive()** method in order to avoid waiting for an indefinite amount of time in case no message is present on the destination. As a result a check is needed to see if ***receive()*** method returned a message or null.