JMS

JMS Tutorial

JMS (Java Message Service) is an API that provides the facility to create, send and read messages. It provides loosely coupled, reliable and asynchronous communication.

JMS is also known as a messaging service.

Understanding Messaging

Messaging is a technique to communicate applications or software components.

JMS is mainly used to send and receive message from one application to another.

Requirement of JMS

Generally, user sends message to application. But, if we want to send message from one application to another, we need to use JMS API.

Consider a scenario, one application A is running in INDIA and another application B is running in USA. To send message from A application to B, we need to use JMS.

Advantage of JMS

1) **Asynchronous:** To receive the message, client is not required to send request. Message will arrive automatically to the client.

2) **Reliable:** It provides assurance that message is delivered.

Messaging Domains

There are two types of messaging domains in JMS.

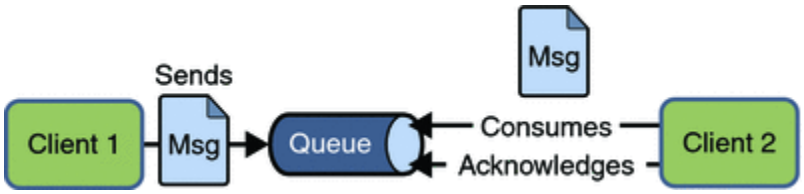
1. Point-to-Point Messaging Domain
2. Publisher/Subscriber Messaging Domain

1) Point-to-Point (PTP) Messaging Domain

In PTP model, one message is **delivered to one receiver** only. Here, **Queue** is used as a message oriented middleware (MOM).

The Queue is responsible to hold the message until receiver is ready.

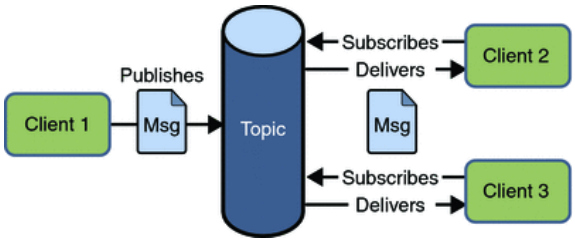
In PTP model, there is **no timing dependency** between sender and receiver.



2) Publisher/Subscriber (Pub/Sub) Messaging Domain

In Pub/Sub model, one message is **delivered to all the subscribers**. It is like broadcasting. Here, **Topic** is used as a message oriented middleware that is responsible to hold and deliver messages.

In PTP model, there is **timing dependency** between publisher and subscriber.



DTD

A Document Type Definition **(DTD)** describes the tree structure of a document and something about its data. It is a set of markup affirmations that actually define a type of document for the SGML family, like GML, SGML, HTML, XML.
A DTD can be declared inside an XML document as inline or as an external recommendation. DTD determines how many times a node should appear, and how their child nodes are ordered.

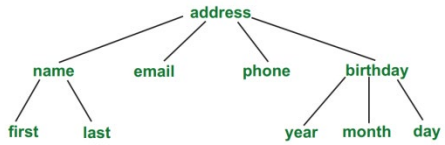
There are 2 data types, PCDATA and CDATA

- PCDATA is parsed character data.
- CDATA is character data, not usually parsed.

Syntax:

```
<!DOCTYPE element DTD identifier
[
  first declaration
  second declaration
  .
  .
  nth declaration
]>
```

Example:



DTD for the above tree is:

XML document with an internal DTD:

- XML

```
<?xml version="1.0"?>
<!DOCTYPE address [
<!ELEMENT address (name, email, phone, birthday)>
<!ELEMENT name (first, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT birthday (year, month, day)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT month (#PCDATA)>
<!ELEMENT day (#PCDATA)>
]>

<address>
   <name>
      <first>Rohit</first>
```

```
    <last>Sharma</last>
  </name>
  <email>sharmarohit@gmail.com</email>
  <phone>9876543210</phone>
  <birthday>
    <year>1987</year>
    <month>June</month>
    <day>23</day>
  </birthday>
</address>
```

The DTD above is interpreted like this:

- !DOCTYPE address defines that the root element of this document is address.
- !ELEMENT address defines that the address element must contain four elements: “name, email, phone, birthday”.
- !ELEMENT name defines that the name element must contain two elements: “first, last”.
    - !ELEMENT first defines the first element to be of type “#PCDATA”.
    - !ELEMENT last defines the last element to be of type “#PCDATA”.
- !ELEMENT email defines the email element to be of type “#PCDATA”.
- !ELEMENT phone defines the phone element to be of type “#PCDATA”.
- !ELEMENT birthday defines that the birthday element must contain three elements “year, month, day”.
    - !ELEMENT year defines the year element to be of type “#PCDATA”.
    - !ELEMENT month defines the month element to be of type “#PCDATA”.
    - !ELEMENT day defines the day element to be of type “#PCDATA”.

XML document with an external DTD:

- XML

```
<?xml version="1.0"?>
<!DOCTYPE address SYSTEM "address.dtd">
<address>
```

```
    <name>
        <first>Rohit</first>
        <last>Sharma</last>
    </name>
    <email>sharmarohit@gmail.com</email>
    <phone>9876543210</phone>
    <birthday>
        <year>1987</year>
        <month>June</month>
        <day>23</day>
    </birthday>
</address>
```

address.dtd:

- <!ELEMENT address (name, email, phone, birthday)>
- <!ELEMENT name (first, last)>
    - <!ELEMENT first (#PCDATA)>
    - <!ELEMENT last (#PCDATA)>
- <!ELEMENT email (#PCDATA)>
- <!ELEMENT phone (#PCDATA)>
- <!ELEMENT birthday (year, month, day)>
    - <!ELEMENT year (#PCDATA)>
    - <!ELEMENT month (#PCDATA)>
    - <!ELEMENT day (#PCDATA)>

Enterprise Java Beans (EJB)

Enterprise Java Beans (EJB) is one of the several Java APIs for standard manufacture of enterprise software. EJB is a server-side software element that summarizes business logic of an application. Enterprise Java Beans web repository yields a runtime domain for web related software elements including computer reliability, Java Servlet Lifecycle (JSL) management, transaction procedure and other web services. The EJB enumeration is a subset of the Java EE enumeration.

The EJB enumeration was originally developed by IBM in 1997 and later adopted by Sun Microsystems in 1999 and enhanced under the Java Community Process.

The EJB enumeration aims to provide a standard way to implement the server-side business software typically found in enterprise applications. Such machine code addresses the same types of problems, and solutions to these problems are often repeatedly re-implemented by programmers. Enterprise Java Beans is assumed to manage such common concerns as endurance, transactional probity and security in a standard way that leaves programmers free to focus on the particular parts of the enterprise software at hand.

To run EJB application we need an application server (EJB Container) such as Jboss, Glassfish, Weblogic, Websphere etc. It performs:

1. Life cycle management
2. Security
3. Transaction management
4. Object pooling

Types of Enterprise Java Beans
There are **three** types of EJB:
1. *Session Bean:* Session bean contains business logic that can be invoked by local, remote or webservice client. There are two types of session beans: (i) Stateful session bean and (ii) Stateless session bean.

- **(i) Stateful Session bean :**
  Stateful session bean performs business task with the help of a state. Stateful session bean can be used to access various method calls by storing the information in an instance variable. Some of the applications require information to be stored across separate method calls. In a shopping site, the items chosen by a customer must be stored as data is an example of stateful session bean.

- **(ii) Stateless Session bean :**
  Stateless session bean implement business logic without having a persistent storage mechanism, such as a state or database and can used shared data. Stateless session bean can be used in situations where information is not required to used across call methods.

2. *Message Driven Bean:* Like Session Bean, it contains the business logic but it is invoked by passing message.
3. *Entity Bean:* It summarizes the state that can be remained in the database. It is deprecated. Now, it is replaced with JPA (Java Persistent API). There are two types of entity bean:

- **(i) Bean Managed Persistence :**
  In a bean managed persistence type of entity bean, the programmer has to write the code for database calls. It persists across multiple sessions and multiple clients.

- **(ii) Container Managed Persistence :**
  Container managed persistence are enterprise bean that persists across database.

In container managed persistence the container take care of database calls.

When to use Enterprise Java Beans
1.Application needs Remote Access. In other words, it is distributed.
2.Application needs to be scalable. EJB applications supports load balancing, clustering and fail-over.
3.Application needs encapsulated business logic. EJB application is differentiated from demonstration and persistent layer.

Advantages of Enterprise Java Beans
1. EJB repository yields system-level services to enterprise beans, the bean developer can focus on solving business problems. Rather than the bean developer, the EJB repository is responsible for system-level services such as transaction management and security authorization.
2. The beans rather than the clients contain the application's business logic, the client developer can focus on the presentation of the client. The client developer does not have to code the pattern that execute business rules or access databases. Due to this the clients are thinner which is a benefit that is particularly important for clients that run on small devices.
3. Enterprise Java Beans are portable elements, the application assembler can build new applications from the beans that already exists.

Disadvantages of Enterprise Java Beans
1. Requires application server
2. Requires only java client. For other language client, you need to go for webservice.
3. Complex to understand and develop EJB applications

1) XML

What is xml

- **Xml** (eXtensible Markup Language) is a mark up language.
- XML is designed to store and transport data.
- Xml was released in late 90's. it was created to provide an easy to use and store self describing data.
- XML became a W3C Recommendation on February 10, 1998.
- XML is not a replacement for HTML.
- XML is designed to be self-descriptive.
- XML is designed to carry data, not to display data.
- XML tags are not predefined. You must define your own tags.
- XML is platform independent and language independent.

XML Example

XML documents create a hierarchical structure looks like a tree so it is known as XML Tree that starts at "the root" and branches to "the leaves".

Example of XML Document

XML documents uses a self-describing and simple syntax:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

The first line is the XML declaration. It defines the XML version (1.0) and the encoding used (ISO-8859-1 = Latin-1/West European character set).

The next line describes the root element of the document (like saying: "this document is a note"):

```
<note>
```

The next 4 lines describe 4 child elements of the root (to, from, heading, and body).

```
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
```

And finally the last line defines the end of the root element.

```
</note>
```

XML documents must contain a **root element.** This element is "the parent" of all other elements.

The elements in an XML document form a document tree. The tree starts at the root and branches to the lowest level of the tree.

All elements can have sub elements (child elements).

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

The terms parent, child, and sibling are used to describe the relationships between elements. Parent elements have children. Children on the same level are called siblings (brothers or sisters).

All elements can have text content and attributes (just like in HTML).

RMI

Remote Method Invocation (RMI) is an API that allows an object to invoke a method on an object that exists in another address space, which could be on the same machine or on a remote machine. Through RMI, an object running in a JVM present on a computer (Client-side) can invoke methods on an object present in another JVM (Server-side). RMI creates a public remote server object that enables client and server-side communications through simple method calls on the server object

Understanding stub and skeleton

RMI uses stub and skeleton object for communication with the remote object.

A **remote object** is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:

stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

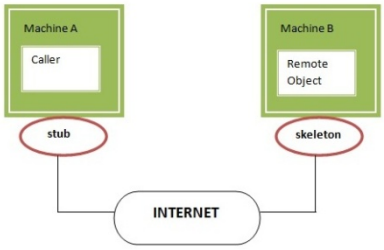
1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

In the Java 2 SDK, an stub protocol was introduced that eliminates the need for



skeletons.

These are the steps to be followed sequentially to implement Interface as defined below as follows:

1. Defining a remote interface
2. Implementing the remote interface
3. Creating Stub and Skeleton objects from the implementation class using rmic (RMI compiler)
4. Start the rmiregistry
5. Create and execute the server application program
6. Create and execute the client application program.

create the remote interface

For creating the remote interface, extend the Remote interface and declare the RemoteException with all the methods of the remote interface. Here, we are creating a remote interface that extends the Remote interface. There is only one method named add() and it declares RemoteException.

```
1. import java.rmi.*;
2. public interface Adder extends Remote{
```

```
3. public int add(int x,int y)throws RemoteException;
4. }
```

Provide the implementation of the remote interface

Now provide the implementation of the remote interface. For providing the implementation of the Remote interface, we need to

- Either extend the UnicastRemoteObject class,
- or use the exportObject() method of the UnicastRemoteObject class

In case, you extend the UnicastRemoteObject class, you must define a constructor that declares RemoteException.

```
1. import java.rmi.*;
2. import java.rmi.server.*;
3. public class AdderRemote extends UnicastRemoteObject implements Adder{
4. AdderRemote()throws RemoteException{
5. super();
6. }
7. public int add(int x,int y){return x+y;}
8. }
```

Create the stub and skeleton objects using the rmic tool.

Next step is to create stub and skeleton objects using the rmi compiler. The rmic tool invokes the RMI compiler and creates stub and skeleton objects.

rmic AdderRemote

Start the registry service by the rmiregistry tool

Now start the registry service by using the rmiregistry tool. If you don't specify the port number, it uses a default port number. In this example, we are using the port number 5000.

```
1. rmiregistry 5000
```

Create and run the client application

At the client we are getting the stub object by the lookup() method of the Naming class and invoking the method on this object. In this example, we are running the server and client applications, in the same machine so we are using localhost. If you want to access the remote object from another machine, change the localhost to the host name (or IP address) where the remote object is located.

```
1. import java.rmi.*;
2. public class MyClient{
```

```
3. public static void main(String args[]){
4. try{
5. Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/sonoo");
6. System.out.println(stub.add(34,4));
7. }catch(Exception e){}
8. }
9. }
```

Exceptions During Remote Object Export

When a remote object class is created that extends UnicastRemoteObject, the object is exported, meaning it can receive calls from external Java virtual machines and can be passed in an RMI call as either a parameter or return value. An object can either be exported on an anonymous port or on a specified port. For objects not extended from UnicastRemoteObject,
the java.rmi.server.UnicastRemoteObject.exportObject method is used to explicitly export the object.

java.rmi.StubNotFoundException

Class of stub not found.

Name collision with class of same name as stub causes one of these errors:

- Stub can't be instantiated
- Stub not of correct class

Bad URL due to wrong codebase.

Stub not of correct class.

java.rmi.server.SkeletonNotFoundException

Note: this exception is deprecated as of Java 2 SDK, Standard Edition, v1.2

Class of skeleton not found.

Name collision with class of same name as skeleton causes one of these errors:

- Skeleton can't be instantiated
- Skeleton not of correct class

Bad URL due to wrong codebase.

Skeleton not of correct class.

java.rmi.server.ExportException

The port is in use by another VM.

A.2 Exceptions During RMI Call

java.rmi.UnknownHostException

Unknown host.

java.rmi.ConnectException

Connection refused to host.

java.rmi.ConnectIOException

I/O error creating connection.

java.rmi.MarshalException

I/O error marshaling transport header, marshaling call header, or marshaling arguments.

java.rmi.NoSuchObjectException

Attempt to invoke a method on an object that is no longer available.

java.rmi.StubNotFoundException

Remote object not exported.

java.rmi.activation.ActivateFailedException

Thrown by RMI runtime when activation fails during a remote call to an activatable object

A.3 Exceptions or Errors During Return

java.rmi.UnmarshalException

Corrupted stream leads to either an I/O or protocol error when:

- Marshaling return header
- Checking return type
- Checking return code
- Unmarshaling return

Return value class not found.

java.rmi.UnexpectedException

An exception not mentioned in the method signature occurred (excluding runtime exceptions). The UnexpectedException exception object contains the underlying exception that was thrown by the server.

java.rmi.ServerError

Any error that occurs while the server is executing a remote method.
The ServerError exception object contains the underlying error that was thrown by the server.

java.rmi.ServerException

This exception is thrown as a result of a remote method invocation when a RemoteException is thrown while processing the invocation on the server, either while unmarshalling the arguments or executing the remote method itself. For examples, see Section A.3.1, "Possible Causes of java.rmi.ServerException".

java.rmi.ServerRuntimeException

Possible Causes of java.rmi.ServerException

These are some of the underlying exceptions which can occur on the server when the server is itself executing a remote method invocation. These exceptions are wrapped in a java.rmi.ServerException; that is the java.rmi.ServerException contains the original exception for the client to extract. These exceptions are wrapped by ServerException so that the client will know that its own remote method invocation on the server did not fail, but that a secondary remote method invocation made by the server failed.

java.rmi.server.SkeletonMismatchException

Note: this exception is deprecated as of the Java 2 SDK, Standard Edition, v1.2

Hash mismatch of stub and skeleton.

java.rmi.UnmarshalException

- I/O error unmarshaling call header.
- I/O error unmarshaling arguments.
- Invalid method number or method hash.

java.rmi.MarshalException

Protocol error marshaling return.