ID2212 Network Programming with Java
Lecture 8

# Java Message Service (JMS) API.
# JavaEmail API.
# Java Naming and Directory Interface (JNDI).

Vladimir Vlassov and Leif Lindbäck
KTH/ICT/SCS
HT 2015

# The Java Message Service API (JMS)

**`javax.jms`**

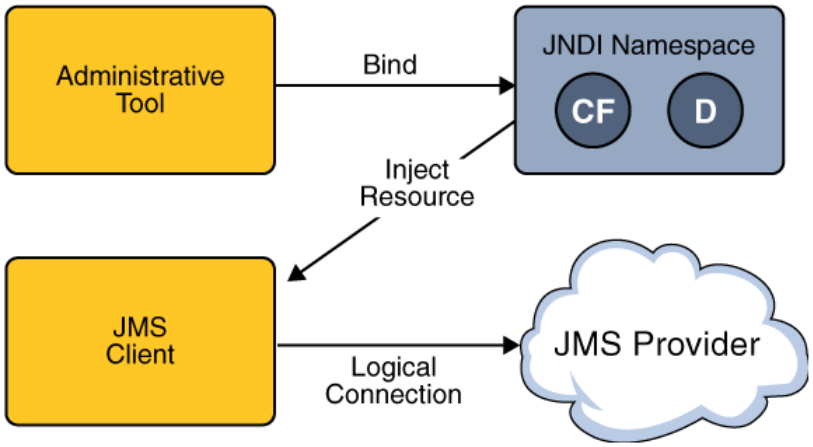
home page:
http://www.oracle.com/technetwork/java/jms/index.html

# The Java Message Service API (JMS)

- Allows Java applications and software components (EJBs, web components) to create, send, receive, and read messages.
- ***Asynchronous message production*** (send)
- ***Synchronous message consumption*** (receive)
- ***Asynchronous message consumption*** by a message listener registered a consumer.
  - Message-driven EJBs asynchronously consume messages.
- ***Reliable messaging***: Can ensure that a message is delivered once and only once.
- ***JMS provider*** is a messaging agent performing messaging

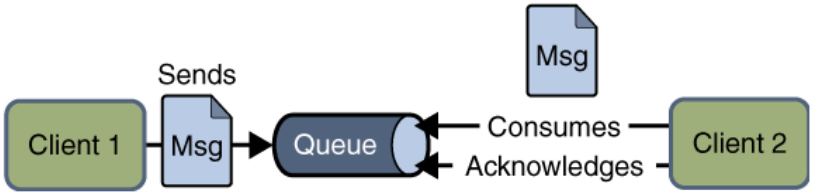
# JMS Architecture

- A JMS application is composed of
- ***A JMS provider***
  - An Application Server, e.g. GlassFish, can be used as a JMS provider.
- ***JMS clients***
  - producing and/or consuming ***messages***.
- ***Messages***
  - objects that communicate information between JMS clients.
- ***Administered objects***
  - ***Destinations (D)***;
  - ***Connection Factories (CF)*** described in Administered Objects
  - created by an administrator for the use of clients

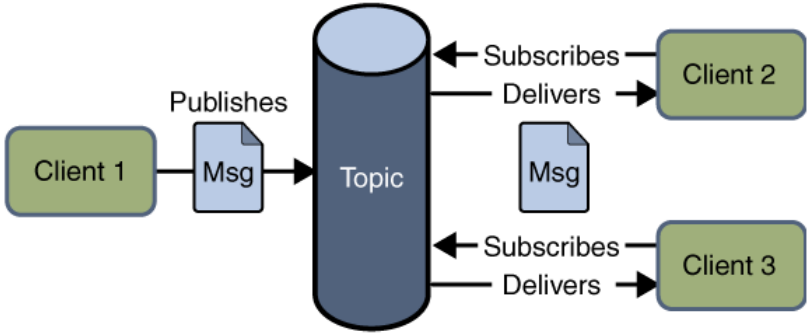


# Two Messaging Domains

- **Queues**: Point-to-Point (PTP) Messaging Domain



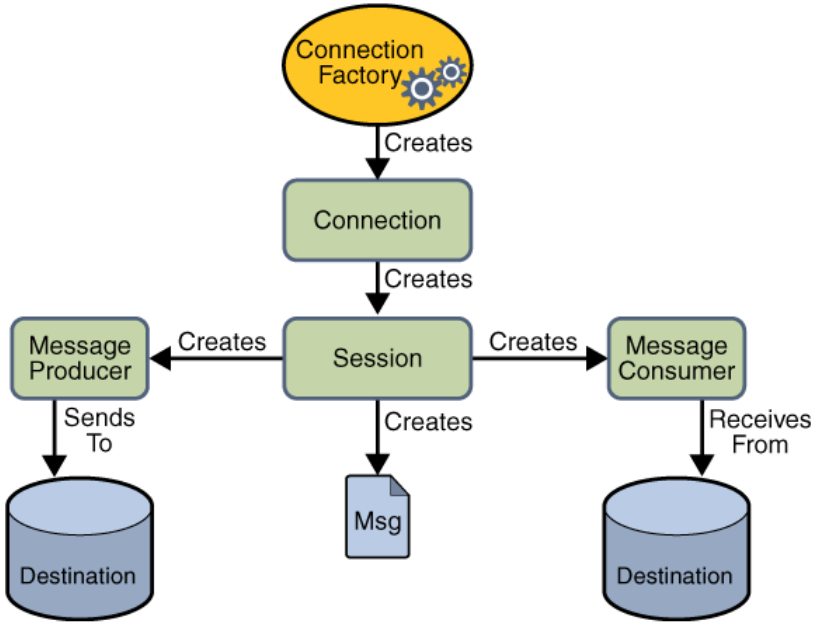

- **Topics**: Publish/Subscribe (pub/sub) Messaging Domain




- A stand-alone JMS provider can implement one or both domains.
- A Java EE provider must implement both domains.

# JMS Programming Concepts

- Administered Objects
  - Connection Factory
  - Destinations (queues, topics, both)
- Connection
- Session
- Message Producers
- Message Consumers
  - Message consumers
  - Message listeners
  - Message selectors
- Messages
  - Headers, properties, bodies
- Queue Browsers




- Steps:
  - Creating a connection and a session
  - Creating message producers and consumers
  - Sending and receiving messages

# Administered Objects

- ***Connection factory*** – the object a client uses to create a connection to a provider:

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

- ***Destination*** – the object a client uses to specify the target of messages it produces and the source of messages it consumes, e.g. Queues, topics, both

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

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

# Connections

- ***Connection*** encapsulates a virtual connection with a JMS provider

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

- Used to create a ***session***
- To consume messages, call the connection's `start` method.

# Sessions

- ***Session*** is a single-threaded context for producing and consuming messages.
- Used to create message producers and consumers, messages, queue browsers, temporary queues and topics
- A ***not transacted session*** with automatic acknowledgement of messages:

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

- A ***transacted session***:

```
Session session = connection.createSession( true, 0);
```

# Message Producer

- A ***message producer*** is an object created by a session and used for sending messages to a destination.
  - Create a producer for a Destination object, a Queue object, or a Topic object:

```
MessageProducer producer = session.createProducer(dest);
MessageProducer producer = session.createProducer(queue);
MessageProducer producer = session.createProducer(topic);
```

  - Send messages by using the send method:

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

  - Create an unidentified producer and specify a destination when sending a message:

```
MessageProducer anon_prod = session.createProducer(null);
anon_prod.send(dest, message);
```

# Message Consumer

- A ***message consumer*** is an object that is created by a session and used for receiving messages sent to a destination.
- Create a MessageConsumer for a Destination object, a Queue object, or a Topic object:

```
MessageConsumer consumer = session.createConsumer(dest);
MessageConsumer consumer = session.createConsumer(queue);
MessageConsumer consumer = session.createConsumer(topic);
```

- Start the connection and use the receive method to consume a message synchronously.

```
connection.start();
Message m = consumer.receive();
connection.start();
Message m = consumer.receive(1000); // time out after a
  second
```

- To consume messages asynchronously, you use ***a message listener***

# Message Listeners

- A ***message listener*** is an object that acts as an asynchronous event handler for messages.
  - Implements the `MessageListener` interface with one method, `onMessage`.

```
public void onMessage(Message message) {
// define the actions to take when a message
   arrives
```

- Register the message listener with a specific MessageConsumer

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

# Messages

- A JMS message has three parts:
  1. (required) a header,
  2. (optional) properties,
  3. (optional) a body.
- A ***header*** contains predefined fields with values that both clients and providers use to identify and to route messages.

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

# Message Body Types

- Five message body formats (a.k.a. message types)
- Examples:

```
TextMessage message =
  session.createTextMessage();
message.setText(msg_text);
producer.send(message);
...
Message m = consumer.receive();
if (m instanceof TextMessage) {
   TextMessage message =
    (TextMessage) m;
   System.out.println("Message: "
    + message.getText());
} else {
   // Handle error
}
```

| Message Type | Contents |
|---|---|
| `TextMessage` | A `String` object (for example, the contents of an XML file). |
| `MapMessage` | A set of name-value pairs, names as `String` and values as primitive types. Entries can be accessed sequentially by enumerator or randomly by name. |
| `BytesMessage` | A stream of bytes. |
| `StreamMessage` | A stream of primitive values. |
| `ObjectMessage` | A `Serializable` object. |
| `Message` | Nothing, but header fields and properties only. |

# Example: Message Producer and Consumer

- Message Producer

```
package jmsprodcon;
import javax.jms.*;
import javax.annotation.Resource;
public class Producer {

    @Resource(mappedName = "jms/ConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/Queue")
    private static Queue queue;
    @Resource(mappedName = "jms/Topic")
    private static Topic topic;
    final static String USAGE = "Producer <destination-type: queue or topic> [<number-
```

- To create JMS administered objects use either
  - Server Admin Console (http://host:4848/)
    - Resources → JMS Resource → Connection Factories (and Destination Resources)
  - The ant tool (see examples in the Java EE tutorial)
  - **In NetBeans add resource to Server Resources**

# Message Producer (cont)

- Destination: either ”queue” or ”topic”

```
public static void main(String[] args) {
        int numberOfMessages = 1;

        if (args.length == 0 ||
                (!args[0].equals("queue") && !args[0].equals("topic")
            System.out.println(USAGE);
            System.exit(1);
        }
        if (args.length > 1) {
            try {
                numberOfMessages = Integer.parseInt(args[1]);
            } catch (NumberFormatException e) {
                System.out.println(USAGE);
                System.exit(1);
            }
        }
        Destination destination = (args[0].equals("queue")) ? queue
        Connection connection = null;
```

# Message Producer (cont)

- Sending messages:

```
    try {
        connection = connectionFactory.createConnection();
        Session session = connection.createSession(false, Session.AUTO_A
        MessageProducer producer = session.createProducer(destination);
        TextMessage message = session.createTextMessage();
        for (int i = 1; i <= numberOfMessages; i++) {
            message.setText("Message number " + i);
            System.out.println("Sending message: " + message.getText());
            producer.send(message);
        }
    } catch (Exception ex) {
        System.out.println("Exception occurred: " + ex.toString());
    } finally {
        if (connection != null) {
            try {
                connection.close();
            } catch (Exception ex) {
            }
        }
    }
}
```

# Synchronous Message Consumer

```
package jmsprodcon;
import javax.jms.*;
import javax.annotation.Resource;

public class Consumer {

    @Resource(mappedName = "jms/ConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/Queue")
    private static Queue queue;
    @Resource(mappedName = "jms/Topic")
    private static Topic topic;
    final static String USAGE = "Consumer <destination-type: queue or topic>";

    public static void main(String[] args) {
        if (args.length == 0 || (!args[0].equals("queue") && !args[0].equals("topi
            System.out.println(USAGE);
            System.exit(1);
        }
        Destination destination = (args[0].equals("queue")) ? queue : topic;
        Connection connection = null;
```

# Synchronous Message Consumer (cont)

```
        try {
            connection = connectionFactory.createConnection();
            Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE
            MessageConsumer consumer = session.createConsumer(destination);
            connection.start();
            Message message;
            while (true) {
                message = consumer.receive(1000);
                if (message == null) {
                    continue;
                }
                if (message instanceof TextMessage) {
                    System.out.println("Received the message: " + ((TextMessage) messa
                } else {
                    break;
                }
            }
        } catch (Exception ex) {
            System.out.println("Exception occurred: " + ex.toString());
        } finally {
            if (connection != null) {
                try {
                    connection.close();
                } catch (Exception ex) { }
            }
        }
    }
}
```

# Asynchronous Message Consumer

- Uses an object with the `MessageListener` interface to be notified on message arrival
- Register a `MessageListener` object with a `MessageConsumer` object
- The `onMessage(Message message)` method on the `MessageListener` object is invoked when a message arrives at the destination

# Asynchronous Message Consumer (cont)

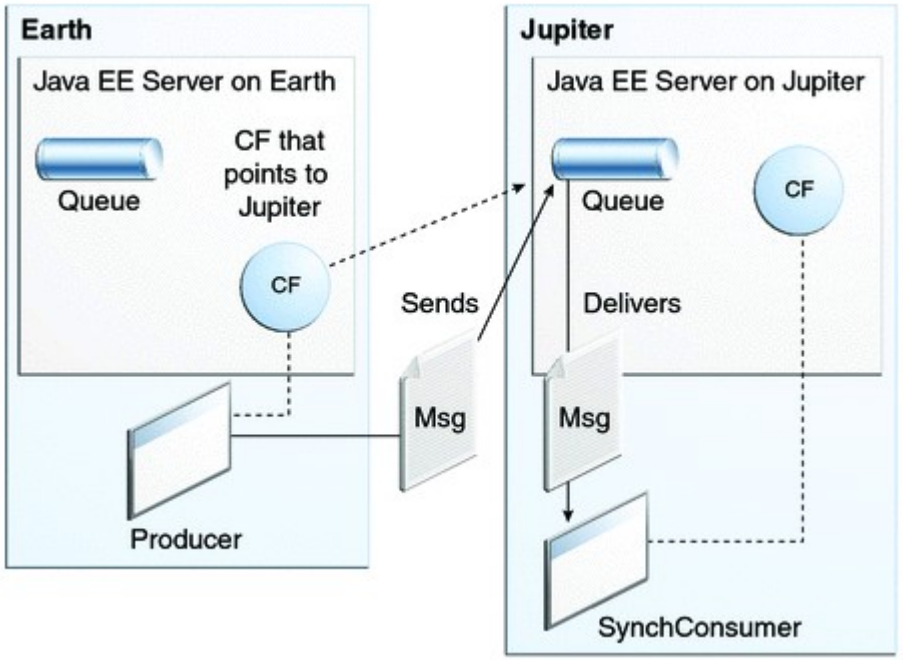
```
package jmsasynchconsumer;
import javax.jms.*;
import javax.annotation.Resource;

public class AsynchConsumer {
    @Resource(mappedName = "jms/ConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/Queue")
    private static Queue queue;
    @Resource(mappedName = "jms/Topic")
    private static Topic topic;
    final static String USAGE = "AsynchConsumer <destination-type: queu

    public static void main(String[] args) {
        if (args.length == 0 || (!args[0].equals("queue")
                    && !args[0].equals("topic"))) {
            System.out.println(USAGE);
            System.exit(1);
        }
        Destination destination = (args[0].equals("queue")) ? queue : t
        Connection connection = null;
```

# Asynchronous Message Consumer (cont)

```
        try {
            connection = connectionFactory.createConnection();
            Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE)
            MessageConsumer consumer = session.createConsumer(destination);
            consumer.setMessageListener(new MessageListener() {
                @Override
                public void onMessage(Message message) {
                    try {
                        if (message instanceof TextMessage)
                            System.out.println("Reading message: " + ((TextMessage)mess
                    } catch (Exception e) {
                    }
                }
            });
            connection.start();
            Object synch = new Object();
            synchronized (synch) {
                try {
                    synch.wait();
                } catch (InterruptedException ex) {}
            }
        } catch (JMSException ex) {
        }
        finally {
            if (connection != null)
                try {
                    connection.close();
                } catch (JMSException ex) {
                }
        }
    }
}
```

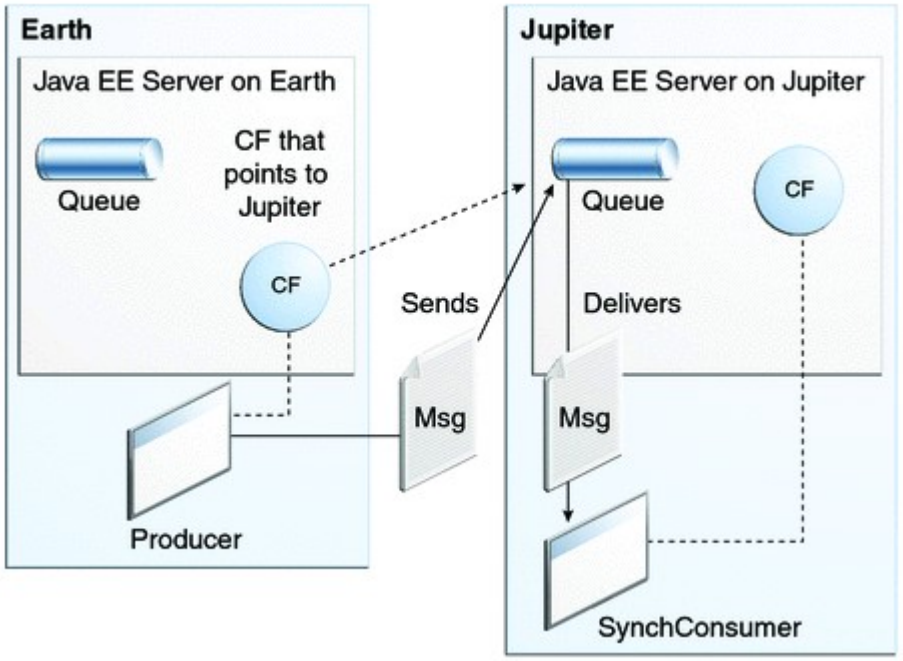
# JMS Clients on Different Systems

- JMS clients that use the GlassFish Server can communicate with each other, when running on different systems in a network.
  - The systems must be visible to each other by name (IP address) and must be running the GlassFish Server.
  - Configuration issue



# JMS Clients on Different Systems (cont'd)

- The connection factory on "Earth" must be remote
- Create and use `jms/JupiterConnectionFactory`
- Set the `AddressList` property for the factory to the name of the remote system
- Set `default_JMS_host` to "0.0.0.0"
- See examples in JavaEE 6 Tutorial



# JavaMail API

`javax.mail.*` and `com.sun.mail.*`

home page:

http://www.oracle.com/technetwork/java/javamail/index.html

# JavaMail API

- The JavaMail API is an optional package for reading, composing, and sending emails.
- The JavaMail API has two parts:
  - An application-level interface for sending/receiving e-mail in a unified way (independent of provider/protocol);
  - A service provider interface (SMTP, POP, IMAP, NNTP).
    - The J2EE platform includes JavaMail with a service provider that allows application components to send Internet mail.

# JavaMail Programming Concepts

- ***Session***
  - A basic mail session
- ***Message***
  - A message to be sent
- ***Address***
  - An email address of a recipient or a sender
- ***Transport***
  - A facility used to connect to the mail server and to send a message
- ***Store and Folder***
  - Represent folders and an email folder, respectively
- ***Authenticator***
  - An object to access to the mail server via a username and password.

# Session

- A basic mail session
- An object of the **`Session`** class
- For example:

```
Properties props = new Properties();
// fill props with any information, e.g. mail s
Session session = Session.getDefaultInstance(pr
// Session session = Session.getInstance(props,
```

# Message

- An email message to be sent
- An object of a `Message` subclass
  - such as `javax.mail.internet.MimeMessage` – a email message that understands MIME types and headers
- For example:

```
MimeMessage message = new MimeMessage(s
message.setContent("Hello", "text/plain
message.setText("Hello");
message.setSubject("First");
```

# Address

- An email address of a recipient or a sender – an object of the **`javax.mail.internet.InternetAddress`** class
- For example:

```
Address fromAddress = new
InternetAddress( “vlad@it.kth.se”,
                                “Vladimir Vlassov”);
Address toAddress = new
InternetAddress(“id2212@ict.kth.se”);
Address ccAddress = new
InternetAddress(“id2212_teachers@ict.kth.se”);
message.setFrom(fromAddress);
message.addRecipient(Message.RecipientType.TO, toAddress);
message.addRecipient(Message.RecipientType.CC, ccAddress);
message.addRecipient(Message.RecipientType.BCC,
fromAddress);
```

# Transport

- An object of the **`Transport`** class used to connect to the mail server and to send a message;
  - Uses a specific protocol for sending messages (usually SMTP).
- For example:

```
message.saveChanges(); // implicit with send()
Transport transport = session.getTransport("smtp");
transport.connect(host, username, password);
transport.sendMessage(message, message.getAllRecipients());
transport.close();
// Transport.send(message); // using the default version of
```

# Store and Folder

- Represent folders and an email folder, respectively
- An object of the **`Folder`** class used for fetching messages from an associated mail folder
- For example:

```
// Store store = session.getStore("imap");
Store store = session.getStore("pop3");
store.connect(host, username, password);
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
Message message[] = folder.getMessages();
System.out.println(((MimeMessage)message).getConte
folder.close(aBoolean);
store.close();
```

# Authenticator

- An object to access to the mail server using a username and password.
- Develop a subclass of `Authenticator` that is used to create `PasswordAuthentication` object when authentication is necessary.
- Instantiate the `Authenticator` subclass and set it in the `Session` object.

# Sending Email Messages

1. Get the system Properties.
2. Setup a mail server:
   - Add the name of an SMTP server to the properties for the `mail.smtp.host` property key.
3. Get a Session object based on the Properties.
4. Create a MimeMessage from the session.
5. Set the from field of the message.
6. Add recepient(s) to the message (to, cc, bcc).
7. Set the subject of the message.
8. Set the content of the message.
9. Use a Transport to send the message.

# Example: Send email

```
package jmail;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;
public class Sender {
    public Sender(String[] args) {
        String host = " webmail.kth.se";
        String[] from = {"vladv@kth.se", "Vladimir Vlassov"};
        String[] to = {"id2212_teachers@ict.kth.se", "ID2212_teachers"};
        String text = "Hi there! This is a test. Ignore.\nCheers!\n-Vlad";
        if (args.length > 0) host = args[0];
        Properties props = System.getProperties(); // Get system properties
        props.put("mail.smtp.host", host); // Setup a mail server
        Session session = Session.getDefaultInstance(props, null);
        MimeMessage message = new MimeMessage(session); // Define a message
        try {
            message.setFrom(new InternetAddress(from[0], from[1]));
            message.addRecipient(Message.RecipientType.TO, new InternetAddress(to[0],
   to[1]));
            message.addRecipient(Message.RecipientType.BCC, new InternetAddress(from[0],
   from[1]));
            message.setSubject("Hello From KTH");
            message.setText(text);
            Transport.send(message); // Send the message
        } catch (UnsupportedEncodingException e) { e.printStackTrace();
        } catch (MessagingException e) { e.printStackTrace(); }
    }
    public static void main(String[] args) {
        Sender sender = new Sender(args);
    }
}
```

# Fetching and Reading Email Messages

- Typical steps:
  1. Get the system Properties.
  2. Get a Session object based on the Properties.
  3. Get a Store for your email protocol, either pop3 or imap.
  4. Connect to the mail host's store with the appropriate username and password.
  5. Get the folder to read, e.g. the INBOX.
  6. Open the folder read-only.
  7. Get a directory of the messages in the folder (a list of messages).
  8. Display the messages one by one (e.g. the “from” field, the “subject” field, a message content).
  9. Close the connection to the folder and store.

# Example: Reading Emails

```
package jmail;
import java.io.*;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
public class Receiver {
   public static void main(String[] args) {
        Receiver receiver = new Receiver(args);
    }
    public Receiver(String[] args) {
        Properties props = new Properties(); // Create empty properties
        // set this session up to use SSL for IMAP connections
        props.setProperty("mail.imap.socketFactory.class",
   "javax.net.ssl.SSLSocketFactory");
        // don't fallback to normal IMAP connections on failure.
        props.setProperty("mail.imap.socketFactory.fallback", "false");
        // use the simap port for imap/ssl connections.
        props.setProperty("mail.imap.socketFactory.port", "993");
        Session session = Session.getInstance(props, null); // Get
   session
```

```
try {
            String host = (args.length > 0) ? args[0] : "webmail.kth.se";
            // Read user name and password; get the store and connect
            Console console = System.console();
            String username = console.readLine("Enter your login: ");
            String password = new String(console.readPassword("Enter password:
   "));
           Store store = session.getStore("imap");
            store.connect(host, 993, username, password);
            // Get folder and open it read-only; show ”From” and ”Subject”
            Folder folder = store.getFolder("INBOX");
            folder.open(Folder.READ_ONLY);
            Message[] message = folder.getMessages();
            for (int i = message.length - 1, n = 0; i >= n; i--) {
                System.out.println(i + ": " + message[i].getFrom()[0] +
                          "\t" + message[i].getSubject());
                String line = console.readLine("Read? [YES/NO/QUIT]");
                if ("YES".equals(line)) {
                    message[i].writeTo(System.out);
                } else if ("QUIT".equals(line)) {
                    break;
                }
            }
            // Close connection
            folder.close(false);
            store.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

# JavaMail API: Other Functionalities

- See:
  - http://www.oracle.com/technetwork/java/index-jsp-139225.html
  - http://java.sun.com/developer/onlineTraining/JavaMail/contents.html
- Deleting Messages and Flags
- Authenticating Yourself
- Replying to Messages
- Forwarding Messages
- Sending and Getting Attachments
- Sending HTML Messages
  - Including Images

# JNDI: Java Naming and Directory Interface

`javax.naming.*`

home page:

http://www.oracle.com/technetwork/java/index-jsp-137536.html

# JNDI Programming Concepts

- ***Name***
  - an entity associated with (bound to) an object or an object reference.
  - A naming system determines the syntax that the name must follow.
- ***Binding***
  - the association of a name with an object.
- ***Context***
  - a set of name-to-object bindings.
- ***Initial context***
  - the starting context for performing naming operations.
- ***Naming service***
  - the means of binding names to objects and looking up objects by names.
- ***Directory and Directory service***
  - a hierarchical collection of named objects with attributes

# JNDI Programming Concepts (cont)

- ***Naming service***
  - the means by which names are bound to objects and objects are found by their names.
  - A client of the service can bind an object to a name, and look up an object by its name.
  - Provides a lookup (resolution) operation that returns the object with a given name
  - May provide operations for binding names, unbinding names, and listing bound names.
  - The operations are performed within the context.
- ***Context (Naming Service)***
  - A set of name-to-object bindings.
  - Has an associated naming convention.
  - Provides naming service operations performed in the context

# Storing Object in Naming Services

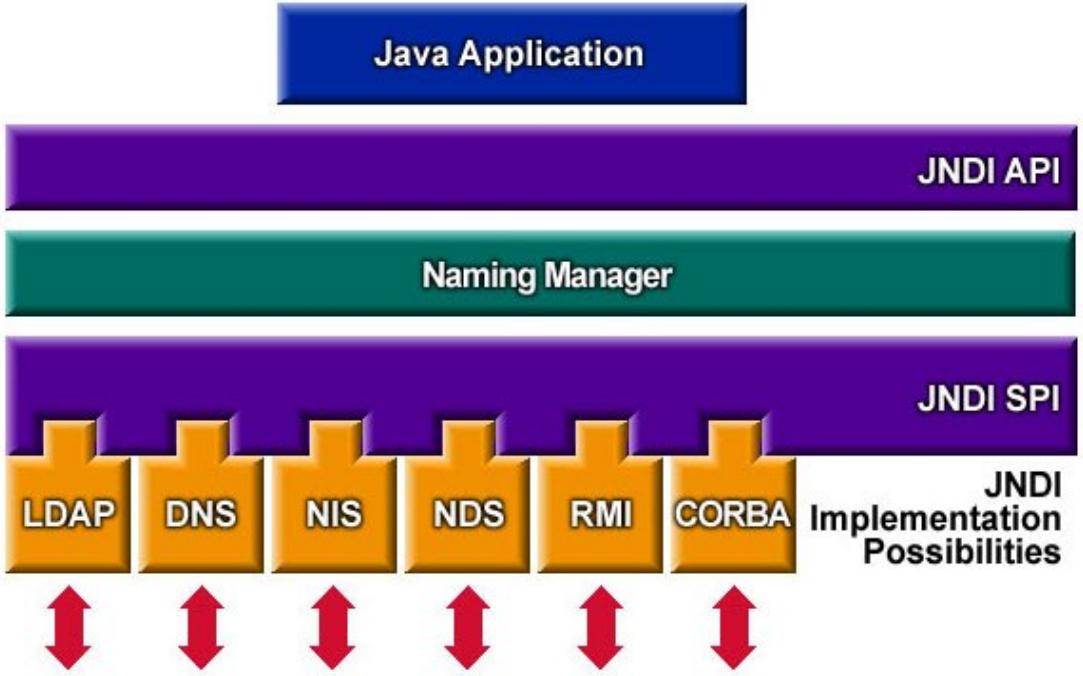
- Two general ways:
  - Serialized version of the Java object.
  - (Remote) object reference with information how to construct an instance of the object
    - The class name
    - A vector of `RefAddr` objects representing the addresses
    - The class name and location of a factory object used to reproduce the object
- A naming service provider can maintain an object store

# Directories and Directory Service

- ***Directory***
  - A connected set of directory objects.
  - A directory object is a named object with attributes, e.g. id/value pairs.
  - A directory can be searched for an object not only by its name but also by its attributes.
- ***Directory service***
  - Many naming services are extended with a directory service. A directory service associates names with objects and also allows such objects to have attributes.
  - A client of the service can bind an object to a name, set /change object's attributes, create subdirectories, search a directory for objects by names and/or by attributes.

# JNDI Architecture

- The JNDI architecture consists of an API and a service provider interface (SPI).



# JNDI Packages

- `javax.naming`
  - classes and interfaces for accessing naming services, such as `Context`, `Name`, `InitialContext`, `Binding`, `Reference`.
  - For binding and looking up objects and object references by names.
- Extensions:
  - `javax.naming.directory` – For accessing directory services in addition to naming services. Search in directories
  - `javax.naming.event` – For supporting event notification in naming and directory services.
  - `javax.naming.ldap` – Features specific for LDAP
  - `javax.naming.spi` – For developers of different naming/directory service providers

# JNDI Service Providers

- To use the JNDI, one must have one or more service providers.
- The Java SDK includes SPI to service providers of the following naming/directory services:
  - LDAP: Lightweight Directory Access Protocol,
  - CORBA Common Object Services (COS) naming service
    - The Naming service provider in JavaIDL: `tnameserv`
    - The client class `org.omg.CORBA.CosNaming`
    - Default port 1050
  - Java Remote Method Invocation (RMI) Registry
    - The Naming service provider in Java RMI: `rmiregistry`
    - The client class `java.rmi.Naming`
    - Default port 1099
  - The Domain Name System (DNS)

# Named Components

- The naming service for an enterprise application deployed on the Sun Java application server is based on the underlying COS naming service
- Names (and aliases) are given to components on deployment
- URI for a named component (e.g. EJB or a data source) of an application starts with `"java:comp/env/"`
- A component can lookup other components by names
- An application component must declare all the environments entries accessed from the application component code

# URLs as JNDI Names

- In the general form, JNDI names look like URIs (URLs)
  - An URI can be used as an address and as a name (in the initial context)
- For example,
  - `java:comp/env/jdbc/Banks`
    - Where `java:comp/env` is a namespace of Java environments
    - `java:comp/env/jdbc` is a namespace of JDBC data sources
  - `java:comp/env/ejb/Adder`
    - `java:comp/env/ejb/` is a namespace of EJBs, i.e. (remote) home interfaces
- Other URI schemes
  - `rmi://host:port/nameOfObject`
  - `ldap://home:port/attr1=value,attr2=value,...`
  - `file:path`

# A Servlet Looks up for an EJB 2

```
Adder adder; // reference to the bean's remote interface
// The web server initializes the servlet
public void init() throws ServletException {
    try {
        InitialContext ic = new InitialContext();
        // The servlet looks up the Adder bean
        Object objref = ic.lookup("java:comp/env/ejb/Adder");
        // Get (connect to) the home interface of the bean
        AdderHome home =
          (AdderHome)PortableRemoteObject.narrow( objref,
          AdderHome.class);
        // Create (initialize) the bean instance
        adder = home.create(0);
    } catch (Exception e) { e.printStackTrace(); }
}
```

# A Server Binds a Name to a Remote Object Reference

- A CORBA application uses COS naming service **`tnameserv`** via the **`NamingContextHelper`** class

```
ORB orb = ORB.init(args, null);
bank.BankImpl bankRef = new bank.BankImpl(args[2]);
orb.connect(bankRef);
org.omg.CORBA.Object objRef =
  orb.resolve_initial_references("NameService");
NamingContext context = NamingContextHelper.narrow(objRef);
NameComponent component = new NameComponent(args[2], "");
NameComponent path[] = {component};
context.rebind(path, bankRef);
```

- A Java RMI application uses **`rmiregistry`** via the **`Naming`** class

```
Bank bankobj = new BankImpl(bankname);
// Register the newly created object at rmiregistry.
Naming.rebind(bankname, bankobj);
```

# A Client Looks Up a Remote Object by Name

- JavaIDL uses **`tnameserv`** via the **`NamingContextHelper`** client

```
ORB orb = ORB.init(args, null);
org.omg.CORBA.Object objRef =
  orb.resolve_initial_references("NameService");
NamingContext context = NamingContextHelper.narrow(objRef);
NameComponent nc = new NameComponent(bankname, "");
NameComponent path[] = {nc};
bankobj =
  bank.BankHelper.narrow(context.resolve(path));
```

- Java RMI uses **`rmiregistry`**

```
Bank bankobj = (Bank)Naming.lookup(bankname);
```