Ansätze:

- Remote Procedure Call (RPC)
  Verteilte Objekte (RPC + Objektorientierung)
- Message Passing

Technologien:

CORBA, RMI, MPI, Jakarta Messaging (JMS), …

- ohne Details zu Übertragung
- ohne Kenntnis zu Protokollen

Gemeinsame Aspekte der Kommunikation

Namensauflösung (und Adressierung) – Suche nach einem entfernten Objekt (Prozedur) über Naming Services

+Initialer Kontext – Konfigurationseinstellungen (Parameter) für Zugriff auf Namensdienst

Binding-Vorgang – Aufbau eines Verbindungskontextes zwischen Client und Server

- statisch
- dynamisch

Java Namensdienst: Java Naming and Directory Interface (JNDI) – eine Programmierschnittstelle für Namens- und Verzeichnisdienste

- liefert zum Namen einen Objekt (Referenz),
- unabhängig von Implementierung

- Bestandteile:
  - eigentliche API - die Sammlung von Interfaces, die vom Service-Provider implementiert werden (**Service Provider Interface)**
  - Service-Implementierung – **Service-Provider** (z.B. RMI-Registry).

- Konfigurationsparameter, werden für den Zugriff auf Namensdienst benötigt:
  - Service Provider, der eine Implementierung zur Verfügung stellt
  - Ort, um Dienst zu lokalisieren
  - optional Security-Einstellungen
  - optional zu verwendete Sprache
  - ...

Übergabe der Konfigurationseinstellungen (z.B. für RMI):

- über InitialContext Konstruktor

```
Hashtable<String, String> env = new Hashtable<String, String>();
env.put (Context.INITIAL_CONTEXT_FACTORY,
                 "com.sun.jndi.rmi.registry.RegistryContextFactory");
env.put (Context.PROVIDER_URL, "rmi://localhost:1099");

InitialContext ictx = new InitialContext(env);
```

Übergabe der Konfigurationseinstellungen (z.B. für RMI):

- Setzen der Systemeigenschaft im Programmcode:

```
System.setProperty("java.naming.factory.initial",
                   "com.sun.jndi.rmi.registry.RegistryContextFactory");
System.setProperty("java.naming.provider.url",
                   "rmi://localhost:1099");
```

- über jndi.properties-Datei:

```
java.naming.factory.initial =
                    com.sun.jndi.rmi.registry.RegistryContextFactory
java.naming.provider.url = rmi://localhost:1099
```

- Angabe beim Programmstart in der Kommandozeile:

```
java –Djava.naming.factory.initial =
                    com.sun.jndi.rmi.registry.RegistryContextFactory
–Djava.naming.provider.url = rmi://localhost:1099
```

Binding-Vorgang: Registrieren bzw. Suchen mit Methoden:

bind(), rebind()

rename()

unbind()

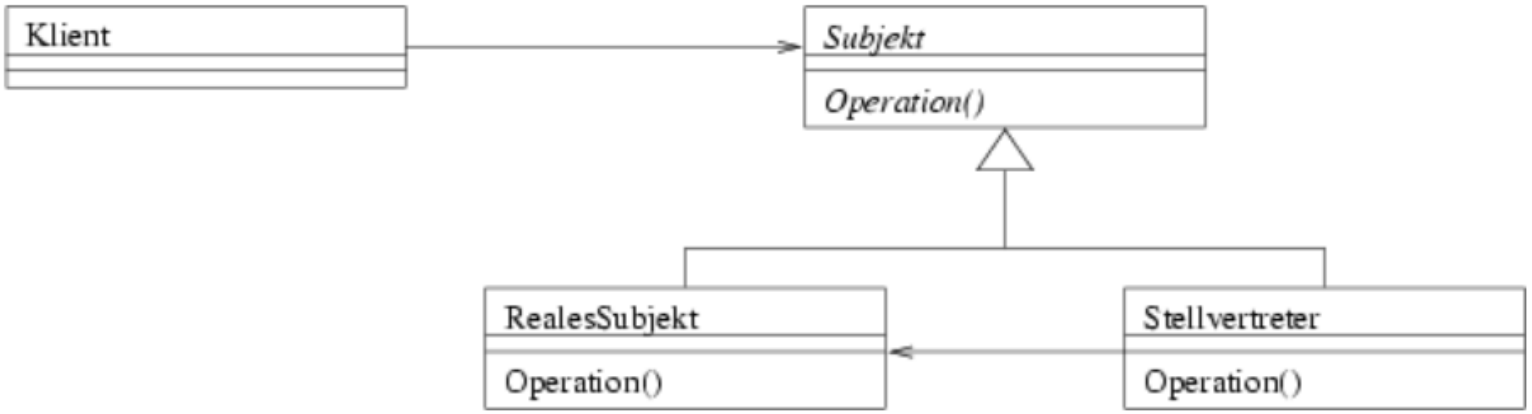
lookup()

```
Object object=context.lookup("name");
```

list()

```
NamingEnumeration enum = context.list("name");
```

Entwurfsmuster - Proxy-Pattern (Stellvertreter)

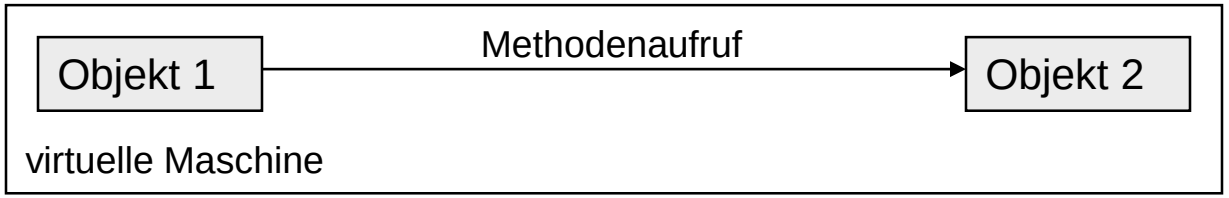


- Klient greift nicht direkt auf das reales Subjekt zu, sondern durch Stellvertreter
- Stellvertreter bietet nach außen eine zum realen Subjekt identische Schnittstelle

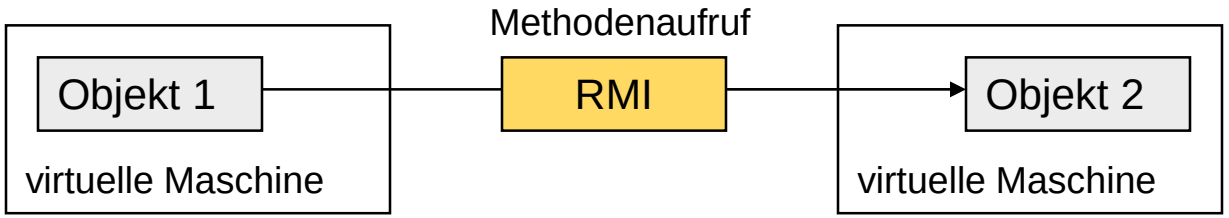
Remote Method Invocation

Ortstransparenz von Objekten:
Methodenaufruf über Rechengrenzen (in anderen virtuellen Maschinen)

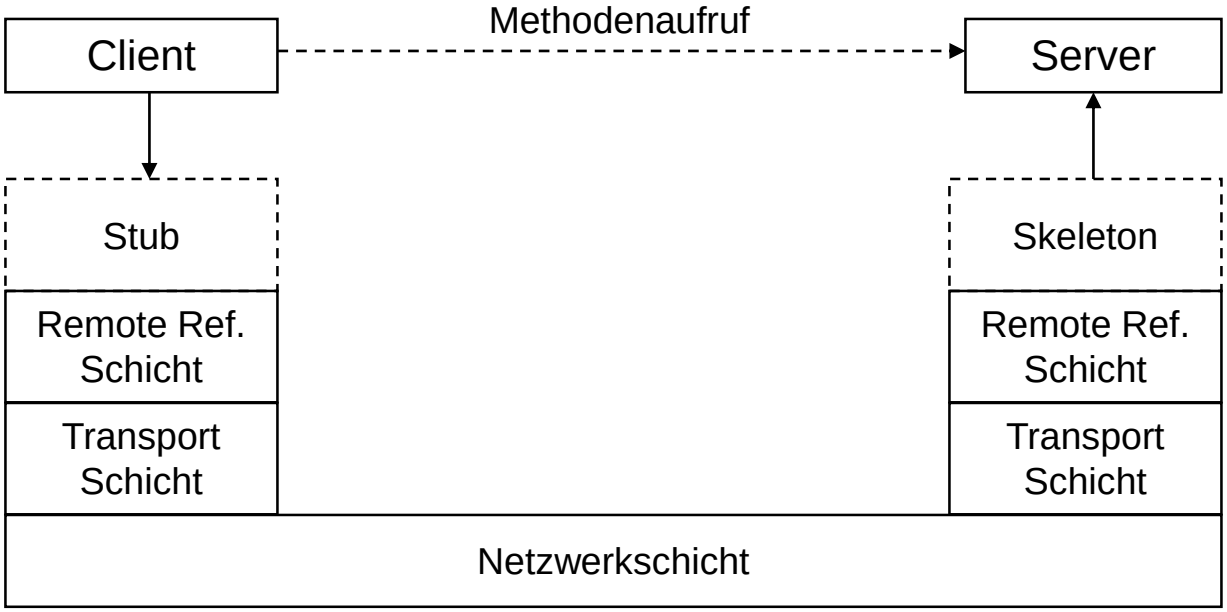
lokaler Methodenaufruf



entfernter Methodenaufruf



RMI Architektur



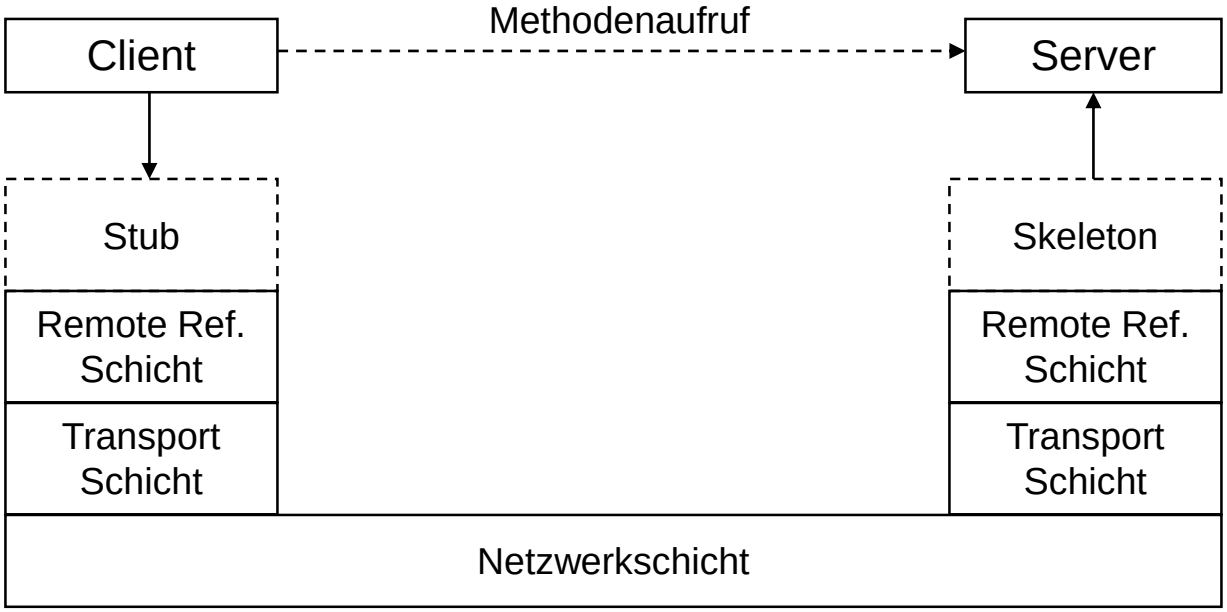
Stub: Stellvertreter des Servers (Proxy), wird generiert ab JDK 5.0

- verfügt über die gleiche Schnittstelle wie Remote-Objekt
- nimmt Parameter entgegen, verpackt in eine Serveranfrage und verschickt diese

Skeleton: ab JDK 1.2 generische Klasse

- nimmt Anfragen entgegen, entnimmt die Daten und führt die Methode auf dem Remote-Objekt aus

RMI Architektur



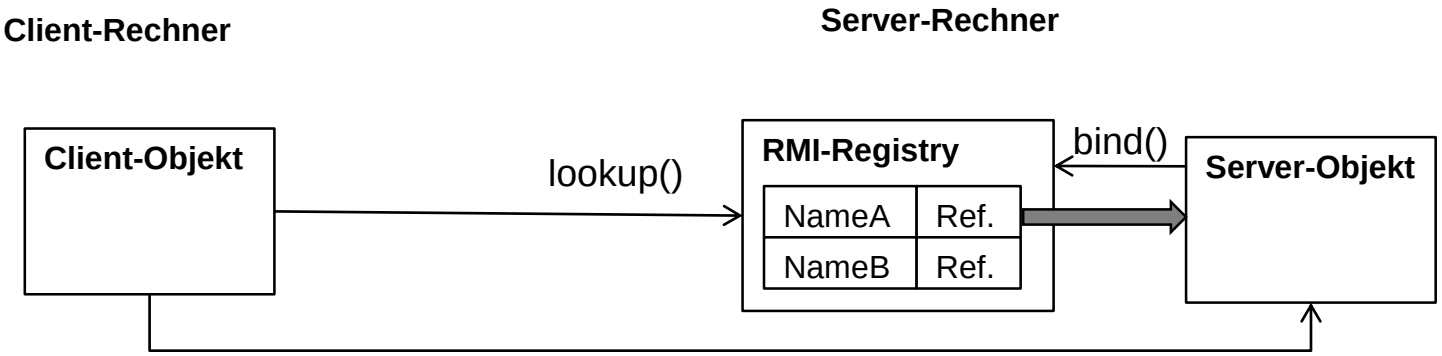
Remote Referenz Schicht:

- Adressierung (IP-Adresse, Port), für RMI-Registry 1099 reserviert

Transportschicht: Datentransportdienste

- Synchrone Kommunikation
- Protokoll: RMI Wire Protocol
- Kommunikation über output und input stream

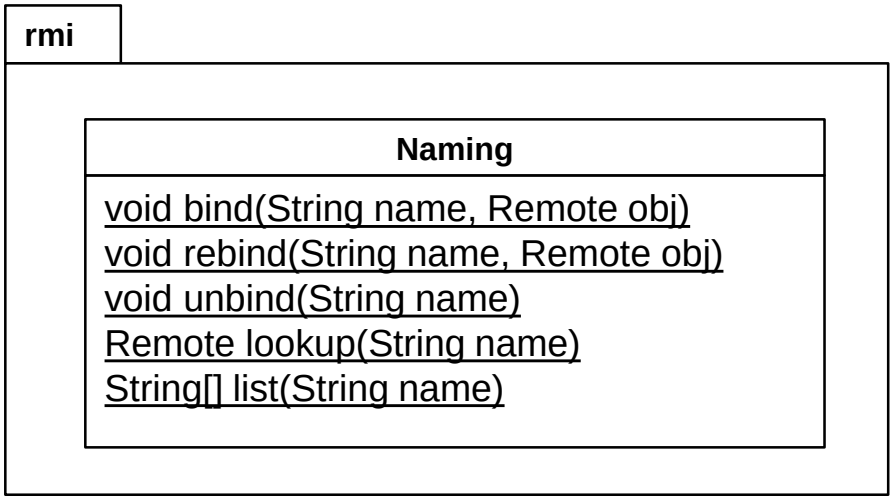
Ablauf



1. Server stellt das entfernte Objekt mit der Methodenimplementierung bereit
2. Server registriert das Objekt bei der RMI-Registry unter einem eindeutigen Namen (bind, rebind)
3. Client sucht in der RMI-Registry unter diesem Namen nach dem entfernten Objekt, erhält Objektreferenz (lookup)
4. Client ruft die Methode an der Objektreferenz mit Parametern auf
5. Server führt die Methode auf dem entfernten Objekt aus

RMI-Registry

- verbindet ein Objekt mit einem eindeutigen Namen
- ein eigenständiges Programm, muss vor RMI-Nutzung aktiv sein
- kann von der Kommandozeile (start registry, Beenden mit ^C) oder aus den Programm gestartet werden
- kommuniziert auf einem vereinbarten Port



Name:

`//HostName/ServiceName`

`///ServiceName`

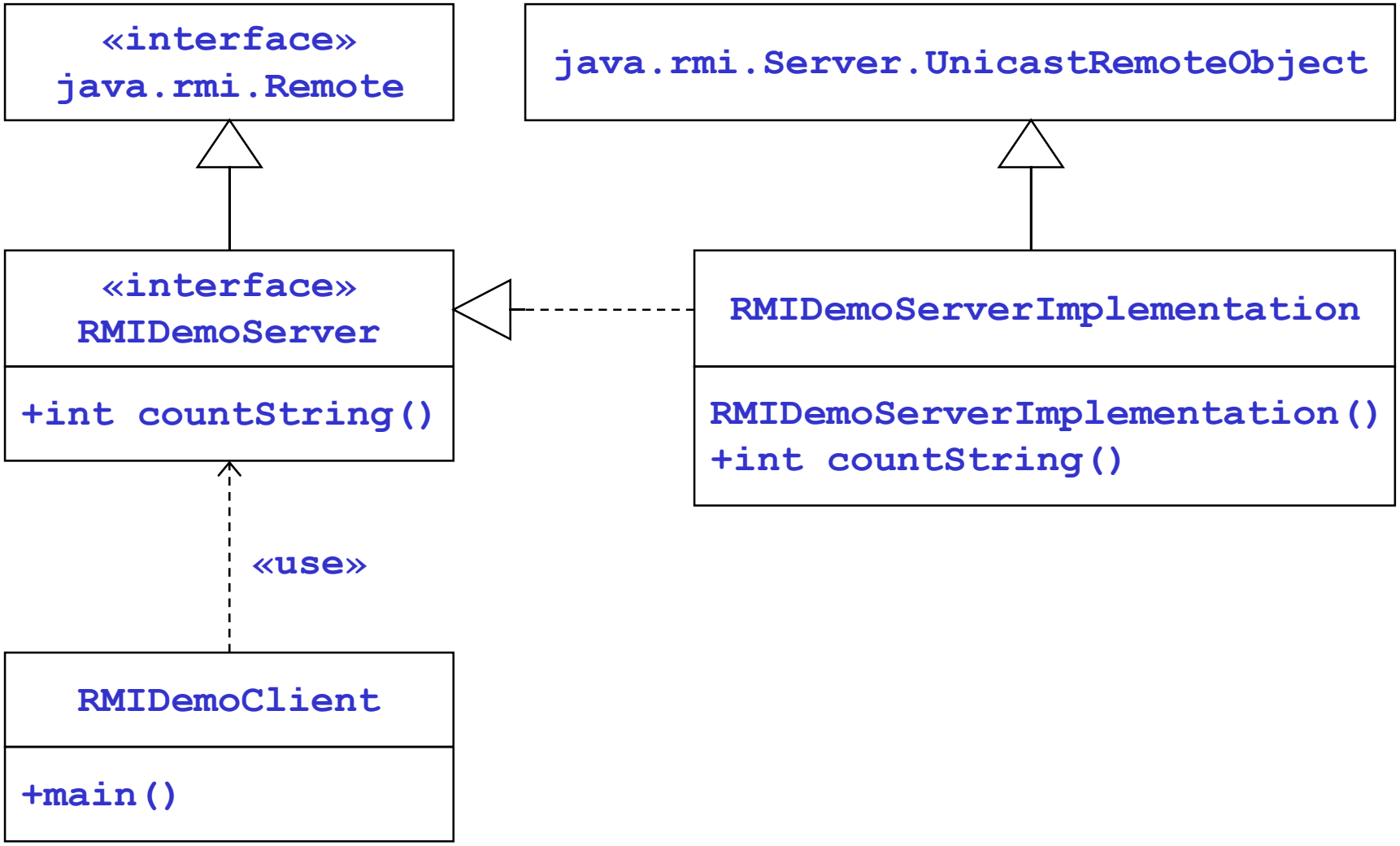
`ServiceName`

Exception

```
MalformedURLException,
RemoteException,    //Ausführung fehlgeschlagen
AccessException,    //Zugriff über Naming-Methode f.
NotBoundException,
AlreadyBoundException
```

Weitere Klassen und Schnittstellen (java.rmi)

```
Remote
RemoteObject //analog Object
server.UnicastRemoteObject
server.RMIClassLoader
RMISecurityManager  //vom Loader benötigt
```



Quelle: Prof. B. Steinbach, Vorlesung Verteilte Software

Remote Interface

gleich beim Server und Client, da die Methodenaufrufe auf dem Stub und Server übereinstimmen müssen

Remote Object

das entfernte Objekt liegt auf dem Server und implementiert das Remote-Interface

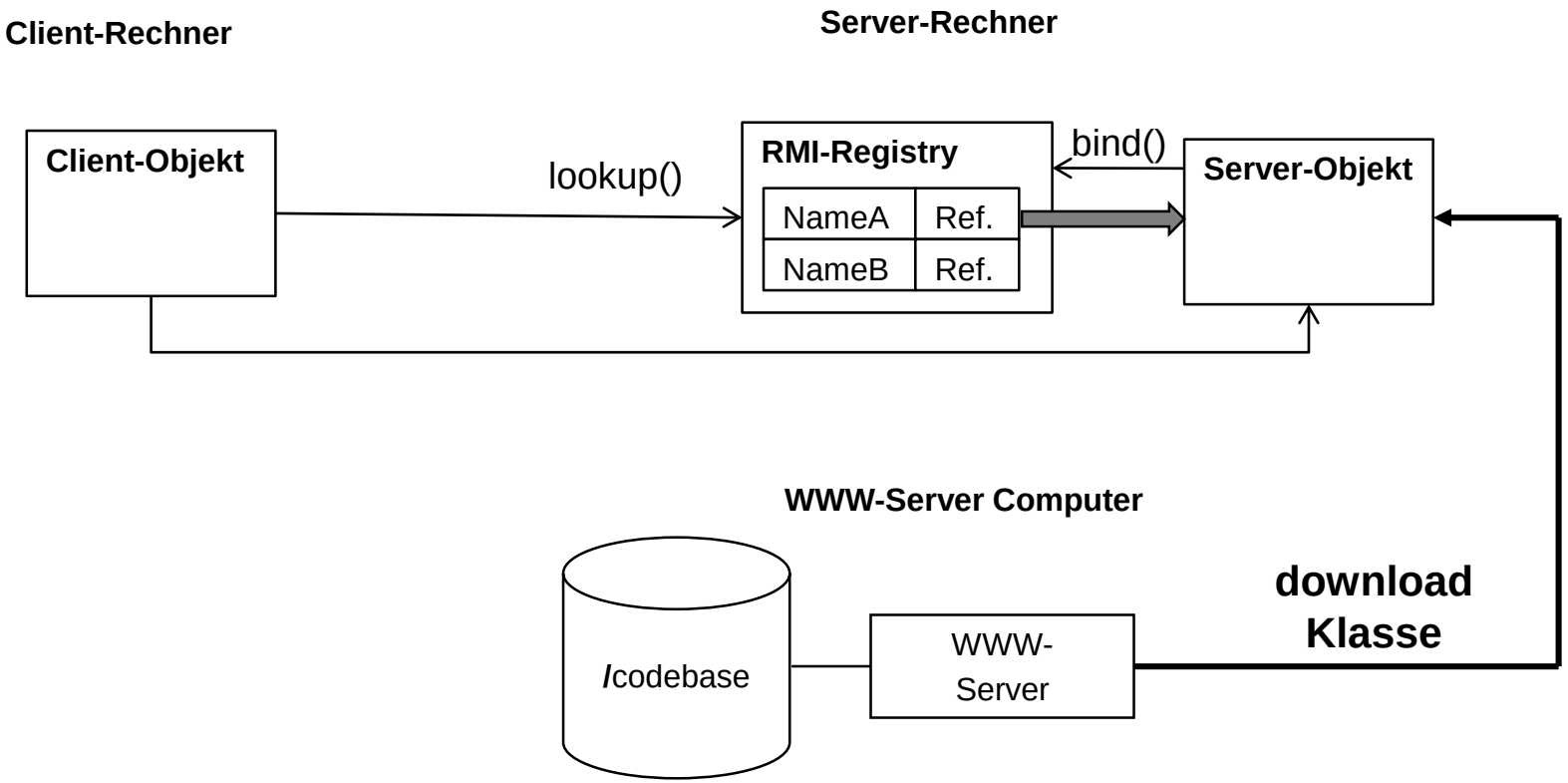
Remote Reference

eine Referenz auf Remote Objects auf der Client-Seite

```
import java.rmi.*;

public interface RMIDemoServer extends Remote
{
  // remote ausführbare Methode des DemoServers
  int countString(String str) throws RemoteException;
}
```

```
import java.rmi.*;
import java.rmi.server.*;
public class RMIDemoServerImplementation extends UnicastRemoteObject
implements RMIDemoServer
{
    public RMIDemoServerImplementation() throws RemoteException
    { try
      { LocateRegistry.createRegistry(Registry.REGISTRY_PORT);//1099
        Naming.rebind("//localhost/DemoServerCS", this);
        System.out.println("Server gebunden");
      }
      catch (Exception e){ // URL or remote exception
        System.out.println("Exception: " + e.getMessage() );
      }
    }
    public int countString(String s) throws RemoteException
    { System.out.println(s);
      return s.length();
    }
    public static void main(String[] args)
    { try { new RMIDemoServerImplementation(); }
      catch (RemoteException e)
      { System.out.println("Fehler beim Instanziieren: " + e.getMessage() ); }
    }
}
```

```
import java.rmi.*;
public class RMIDemoClient
{
     public static void main(String[] args) {
         int zeichenanzahl;
         String str= " standard string ";

         if (args.length != 1)
             str = "Aufruf mit java RMIDemo.RMIDemoClient \"Zeichenkette\"";
         else str = args[0];

         try
         {
          // search in registry for server
           RMIDemoServer server =
                   (RMIDemoServer)Naming.lookup("//localhost/DemoServerCS");
           System.out.println(str);
           // execute server method
           zeichenanzahl = server.countString(str);
           System.out.println("Anzahl der Zeichen: " + zeichenanzahl);
         }
         catch (Exception e)
         { System.out.println("Exception: " + e.getMessage() ); }
     }
}
```


Client-Rechner
Server-Rechner
Client-Objekt
lookup()
RMI-Registry
NameA
Ref.
NameB
Ref.
bind()
Server-Objekt
WWW-Server Computer
/codebase
WWW-
Server
download
Klasse

Parameter:

1. Klassen, die auf beiden Seiten vorliegen
2. Klassen, die dem Server oder dem Client nicht bekannt sind

   z.B.

   ```
   methode (X x); //übergeben wird aber von X abgeleitetes Y
   int max(List<?> v);
   ```

   dynamisches Nachladen notwendig!

   Klassen werden gesucht:
   - im Verzeichnis (CLASSPATH oder aktuellem)
   - evtl. auf dem Webserver

3. Klassen, die selbst Remote implementieren

   Reference-Parameter!

Dynamisches Laden (hier vom Server)

Client:
Umgebungsvariable:

- System.setProperty("**java.rmi.server.codebase**" , " … ");
- java –Djava.rmi.codebase=http://… <java-Klasse><Argumente>

Server:
Sicherheitsmanager:

- System.setSecurityManager(new RMISecurityManager());
- java –Djava.security.manager <java-Klasse><Argumente>

java.policy-Datei:

- java –Djava.security.policy=<Dateiname> <java-Klasse><Argumente>

```
grant {
  permission java.security.AllPermission;
};
```

Quellen:
https://docs.oracle.com/javase/tutorial/rmi/overview.html
https://docs.oracle.com/javase/8/docs/technotes/guides/security/PolicyFiles.html

Übertragung der Parameter

call by reference

- nur wenn Objekt ein Remote-Objekt ist
  (java.rmi.Remote implementiert)
  und als Parameter der Methode übergeben wird
- alle Attribute von Remote-Objekten sind transient

call by value

- eine Kopien des Objektes wird verschickt
- Objekte müssen serialisierbar sein:
  - java.lang.Seriaizable implementieren
  - SerialisierungsID besitzen

```
import java.io.Serializable;
import java.util.ArrayList;

public class Ergebnis implements Serializable {
  private static final long serialVersionUID = 1L;
  private ArrayList<String> saetze;

  public Ergebnis() {
    super();
    saetze= new ArrayList<String>();
  }

  public ArrayList<String> getSaetze() {
    return saetze;
  }
}
```

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RMIInterface extends Remote {
  public Ergebnis getOffers(String s) throws RemoteException;
}
```

```
import java.net.MalformedURLException;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.UnicastRemoteObject;

public class RMIServer extends UnicastRemoteObject implements RMIInterface{

protected RMIServer() throws RemoteException {super();}

@Override
public Ergebnis getOffers(String s) {
    Ergebnis erg=new Ergebnis();
    erg.getSaetze().add("Hey "+ s); //…
    return erg;
}

public static void main(String[] args) {
    try {
        LocateRegistry.createRegistry( Registry.REGISTRY_PORT );
        Naming.rebind("offer-server", new RMIServer());
        System.out.println("offer-server gebunden");
    }
    catch (RemoteException | MalformedURLException e) { e.printStackTrace();}
  }
}
```

```
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;

public class RMIClient {

public static void main( String[] args ) throws RemoteException, NotBoundException
{
    try {
      RMIInterface server;
      server = (RMIInterface) Naming.lookup("//localhost/offer-server");
      Ergebnis erg=server.getOffers("Klaus");
      for (int i=0;i<erg.getSaetze().size();i++)
            System.out.println( erg.getSaetze().get(i) );
    }
    catch (MalformedURLException e) {
      e.printStackTrace();
    }
}
}
```

Achtung: RMI ist synchron

was ist wenn

- Kommunikationspartner gleichberechtigt sind, wie z.B. bei Producer-Consumer-Problemstellungen
- Empfänger nicht immer verfügbar ist. Wie kann man die enge Kopplung aufbrechen?
    - Nachrichten müssen ggf. zwischengespeichert werden,
    - der Sender muss weiterarbeiten können und
    - Empfänger muss die Nachrichten später holen können

Jakarta Messaging (Java Message Service)

Wikipedia: eine Programmierschnittstelle (API) für die Ansteuerung einer Message Oriented Middleware (MOM) zum Senden und Empfangen von Nachrichten aus einem *Client* heraus

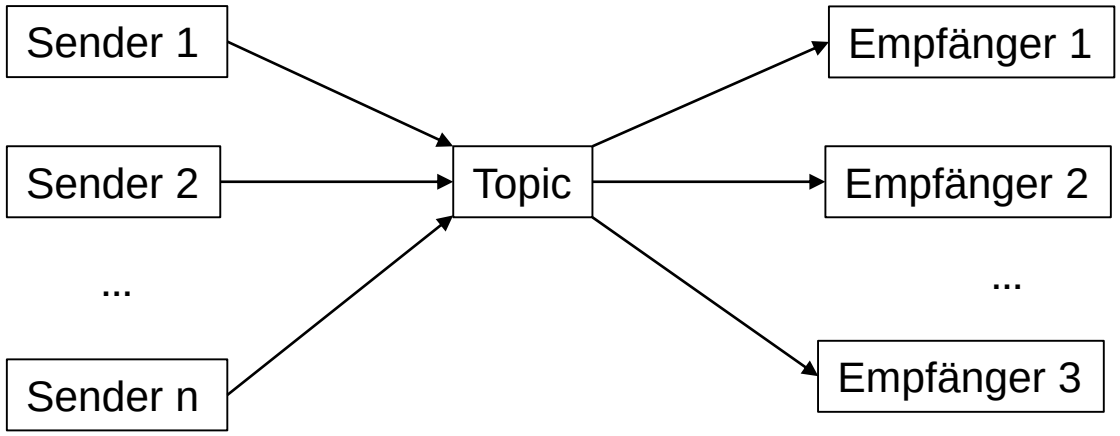
- benötigt eine Zwischenkomponente - einen Provider, der den Dienst bereitstellt
- wird meist von einem Web-Server zur Verfügung gestellt (Glassfish…)
- Java-Clients
  - verbinden sich mit JMS-System (Verbindungsfabriken),
  - verschicken und
  - abholen Nachrichten über Nachrichtenkanäle
- Nachrichtenkanäle und Connection-Factories werden vom Provider verwaltet
- ermöglicht asynchrone Kommunikation
- Paket jakarta.jms

JMS-Provider: verwaltet Nachrichten (Topics, Queues) und Verbindungen (Connection Factories)

Name	Firma	Lizenz	Betriebsmodi	URL
ActiveMQ	Apache	Open Source (Apache 2)	eigenständig, eingebettet	apache.org
FuseMQ	Red Hat	Open Source (Apache 2), kommerzieller Support möglich	eigenständig, eingebettet	fusesource.com
Apollo	Apache	Open Source (Apache 2)		apache.org
FioranoMQ	Fiorano	kommerziell		
iBus//MessageServer	Softwired	kommerziell		
HornetQ (ehemals bekannt unter "JBoss Messaging")	Red Hat	Open Source (Apache 2)	eigenständig, eingebettet	jboss.org
JORAM	OW2	Open Source (LGPL)	eigenständig, eingebettet	ow2.org
MantaRay	Coridan	Open Source (Mozilla Public License)	eigenständig, eingebettet	
Mom4j	Mom4j development team	Open Source (LGPL)	eingebettet	
MRG Messaging	Red Hat	kommerziell		redhat.com
MuleMQ	MuleSoft Inc.	kommerziell		mulesoft.com
OpenJMS		Open Source	eigenständig, eingebettet	sourceforge.net
Open Message Queue	Sun Microsystems	Open Source	eigenständig	mq.java.net
Oracle Advanced Queuing (OAQ)	Oracle	kommerziell	eingebettet	oracle.com
Qpid	Apache	Open Source (Apache 2)		apache.org
SAP JMS	SAP	kommerziell	eingebettet	sap.com
SonicMQ	Progress Software	kommerziell		
SwiftMQ	IIT Software	Open Source (Apache 2)	eigenständig	swiftmq.com
TIBCO Enterprise Message Service	TIBCO	kommerziell		
webMethods Broker	Software AG	kommerziell	eigenständig, eingebettet	softwareag.com
webMethods Universal Messaging	Software AG	kommerziell	eigenständig, eingebettet	softwareag.com
Websphere MQ	IBM	kommerziell	eigenständig, eingebettet	ibm.com
WSO2 Message Broker	WSO2	Open Source (Apache 2)	eigenständig	wso2.com

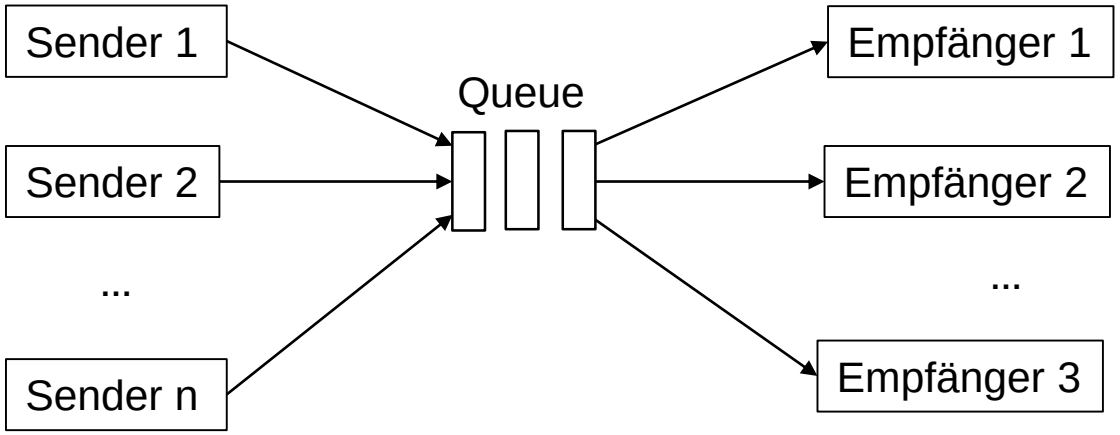
https://de.wikipedia.org/wiki/Jakarta_Messaging

Publish-Subscribe-Modell mit Topic



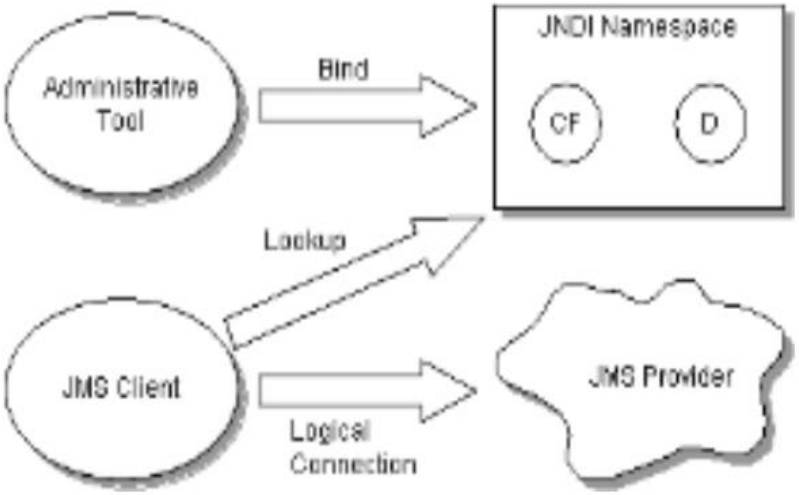
- Nachrichten werden von mehreren Empfängern empfangen und von mehreren Sendern gesendet
- Subscriber registriert sich zur Laufzeit für ein oder mehrere Topics
- Nachrichten werden automatisch an Subscriber geleitet, Subscriber muss aktiv sein

Point-to-Point-Modell mit Queue



- Nachricht wird an eine bestimmte Queue adressiert
- Queue behält die Nachricht so lange, bis sie abgerufen oder abgelaufen ist
- der erste Empfänger, der eine Nachricht liest, entfernt diese aus der Queue
- Empfänger können zur Laufzeit hinzukommen

Konfigurieren des Servers



Quelle:
https://jakarta.ee/specifications/messaging/3.0/jakarta-messaging-spec-3.0.html

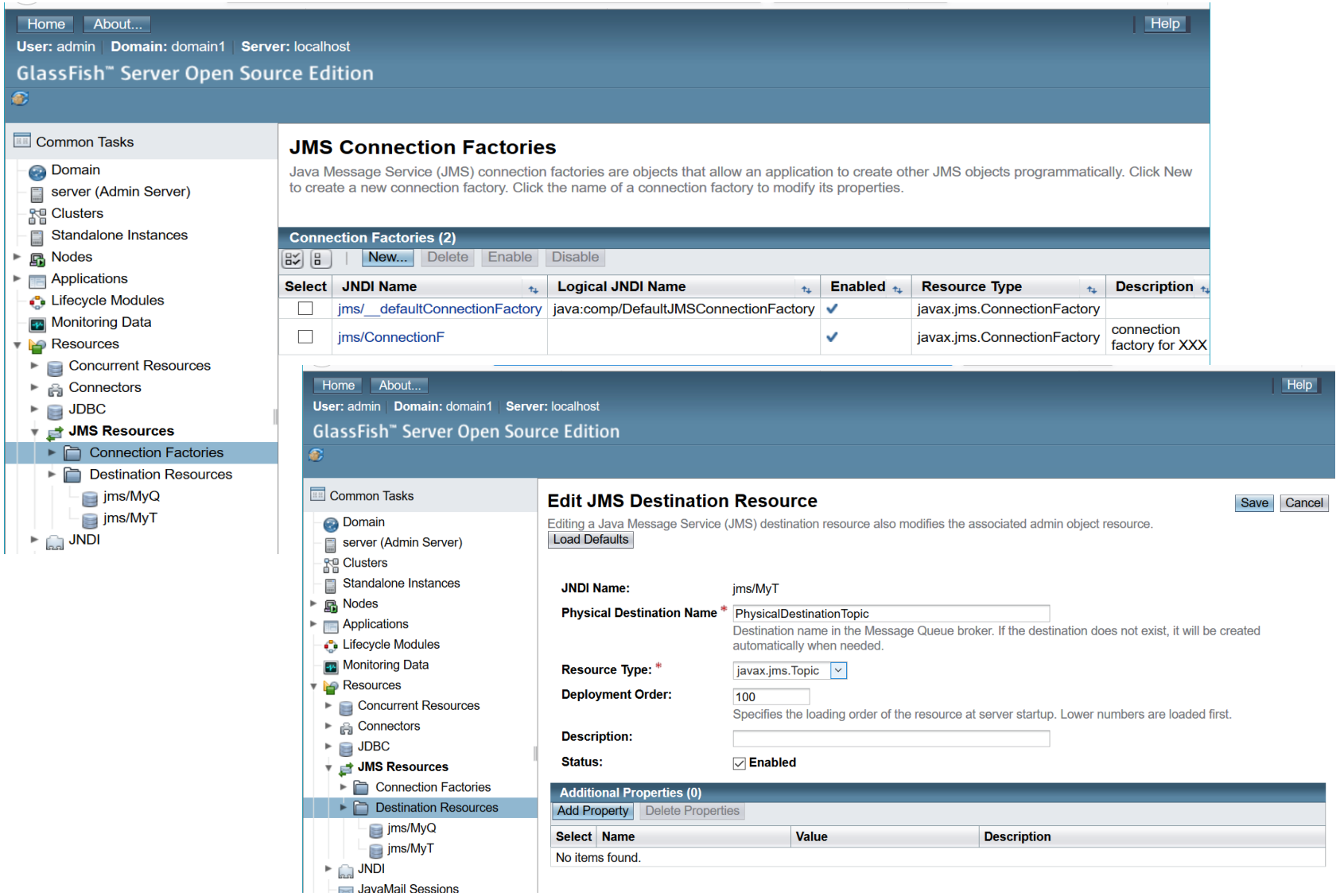
Einrichten von
- Verbindungsfabriken (Connection Factory) und
- Nachrichtenkanälen (Destination: Topic, Queue)

Beide werden unter einem eindeutigen Namen (frei wählbar, aber standardisiert) registriert und vom Client angesprochen. Die Verwaltung der Namen übernimmt das JNDI (java naming and directory service).

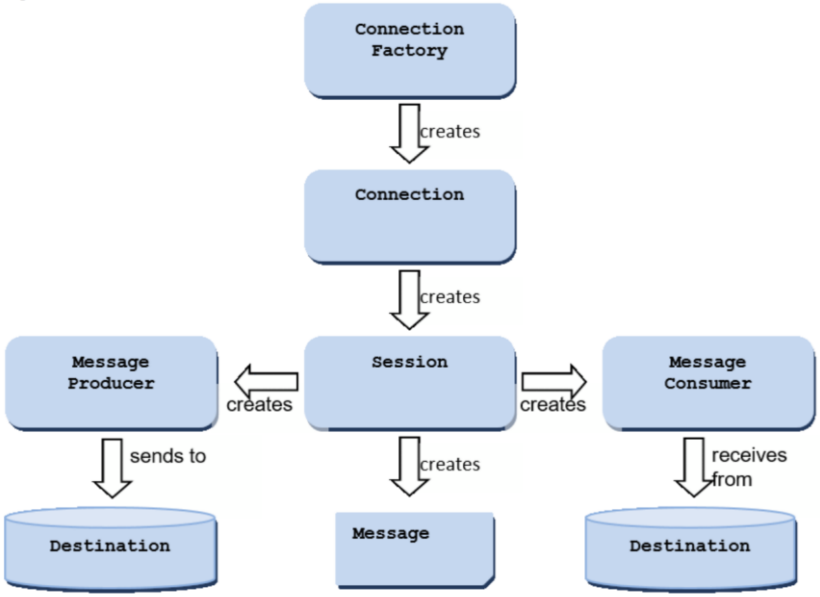
Konfigurieren erfolgt über Admin-Console,
des GlassFish-Servers z.B.

http//localhost:4848/

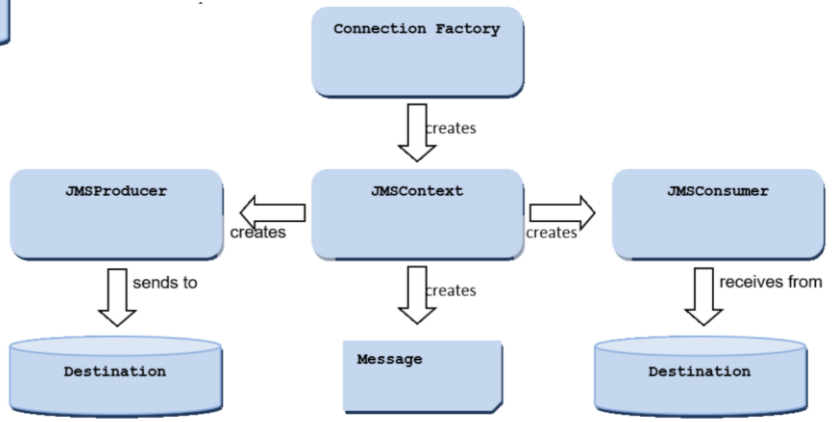
…\glassfish-4.1.1\bin\asadmin.bat (beenden mit exit)


Home
About...
Help
User: admin | Domain: domain1 | Server: localhost
GlassFish™ Server Open Source Edition
Common Tasks
Domain
server (Admin Server)
Clusters
Standalone Instances
Nodes
Applications
Lifecycle Modules
Monitoring Data
Resources
Concurrent Resources
Connectors
JDBC
JMS Resources
Connection Factories
Destination Resources
jms/MyQ
jms/MyT
JNDI
JMS Connection Factories
Java Message Service (JMS) connection factories are objects that allow an application to create other JMS objects programmatically. Click New to create a new connection factory. Click the name of a connection factory to modify its properties.
Connection Factories (2)
New...
Delete
Enable
Disable
Select
JNDI Name
Logical JNDI Name
Enabled
Resource Type
Description
jms/__defaultConnectionFactory
java:comp/DefaultJMSConnectionFactory
javax.jms.ConnectionFactory
jms/ConnectionF
javax.jms.ConnectionFactory
connection factory for XXX
Edit JMS Destination Resource
Save
Cancel
Editing a Java Message Service (JMS) destination resource also modifies the associated admin object resource.
Load Defaults
JNDI Name:
jms/MyT
Physical Destination Name *
PhysicalDestinationTopic
Destination name in the Message Queue broker. If the destination does not exist, it will be created automatically when needed.
Resource Type: *
javax.jms.Topic
Deployment Order:
100
Specifies the loading order of the resource at server startup. Lower numbers are loaded first.
Description:
Status:
Enabled
Additional Properties (0)
Add Property
Delete Properties
Select
Name
Value
Description
No items found.
JavaMail Sessions

Ablauf



Ab JMS 2.0 API: JMSContext = Connection und Session



Quelle:
https://jakarta.ee/specifications/messaging/3.0/jakarta-messaging-spec-3.0.html

Interfaces

Connection Factory:

ConnectionFactory, QueueConnectionFactory, TopicConnectionFactory

Connection:

Connection, QueueConnection, TopicConnection

Session:

Session, QueueSession, TopicSession

Context:

JMSContext

Message:

Message, BytesMessage, MapMessage, ObjectMessage, StreamMessage, TextMessage

Resource:

Topic, Queue

Message Producer:

MessageProducer, QueueSender, TopicPublisher, **JMSProducer**

Message Consumer:

MessageConsumer, QueueReceiver, TopicSubscriber, **JMSConsumer**

ConnectionFactory (QueueConnectionFactory, TopicConnectionFactory)

- erstellt eine Verbindung zum Provider
- benötigt Konfigurationsparameter (InitialContext)
- stellt Methoden zum Erstellen von JMSContext (Connection etc.) zur Verfügung

```
Properties p = new Properties();
//hier können die eigenschaften definiert werden
Context context = new InitialContext(p);
ConnectionFactory connectionFactory =
          (ConnectionFactory)context.lookup("jms/RemoteConnectionFactory");
```

Methoden:

```
JMSContext createContext()
Connection createConnection()
//und weitere
```

JMSContext (Connection+Session)

- kombiniert Funktionalität von Connection und Session
- produziert und konsumiert Messages
- kann mehrere Sender und Empfänger erstellen und betreuen

Methoden:

```
Message createMessage()
TextMessage createTextMessage()
JMSConsumer createConsumer(Destination destination)
JMSProducer createProducer()
```

Attribute:

```
static final int AUTO_ACKNOWLEDGE //Annehmen von Nachrichten
static final int SESSION_TRANSACTED //für mehrere Ressourcen
```

Beispiele:

```
TextMessage msg = context.createTextMessage("Mesaje");
JMSProducer sender=context.createProducer();
JMSConsumer receiver=context.createConsumer(queue);
```

Message:

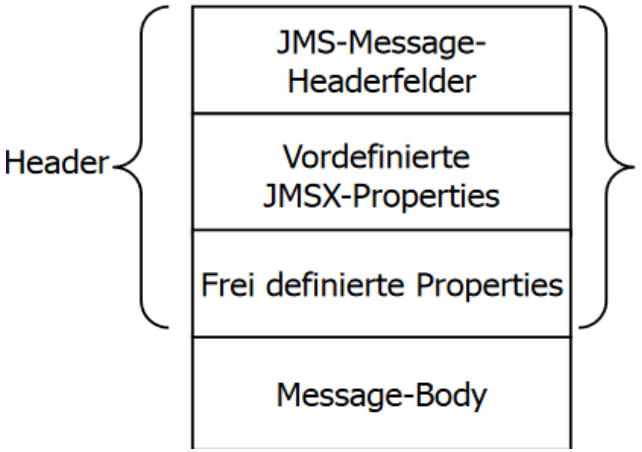
Message, TextMessage, MapMessage, BytesMessage, StreamMessage, ObjectMessage

wird erstell JMSContext (Session)

```
TextMessage message = session.createTextMessage();
message.setText(“Dies ist der Nachrichtentext“);
```

Bestandteile: JMS-Message-Header, Properties, Body (Daten)

aber nur JMS-Header ist Pflicht



- werden z. T. implizit beim Senden gesetzt
- können mit get/set-Methoden gelesen bzw. gesetzt werden
- werden in den entsprechenden Attributen festgehalten

JMS-Message-Header-Felder:

- Informationen:
  - Ziel (Topic oder Queue)
  - eindeutige ID (String)
  - vom JMS-Client vergebener Typ der Nachricht
  - Zeitpunkt der Übergabe an den JMS-Provider
  - Priorität: 0-4 normal, 5-9 hoch
  - weitere

Properties

- vordefinierte (JMSX-Properties):
  z.B. Identifikation eines JMS-Clients
- individuelle: mit Name und Value spezifiziert

```
msg.setIntProperty("releaseYear", 2022);
int releaseYear = msg.getIntProperty("releaseYear");
```

Message Producer

JMSProducer (**MessageProducer, QueueSender, TopicPublisher)**

- wird von JMSContext (Session) erzeugt
- erhält eine Ressource (Queue oder Topic)
- versendet Nachrichten über verschiedene send-Methoden
- für asynchrones Senden mit Antwort benötigt ein CompletionListener (`jakarta.jms.CompletionListener`)
  mit `onCompletion` und `onException`-Methoden

Beispiele:

```
JMSProducer sender=context.createProducer();
sender.send(queue, message);

JMSProducer sender=context.createProducer();
MyCompletionListener complistener=new MyCompletionListener();
sender.setAsync(complistener);
sender.send(…);

public class MyCompletionListener implements
CompletionListener{
    @Override
    public void onCompletion(Message msg) {
        System.out.println("Message successfully sent");
    }

    @Override
    public void onException(Message msg, Exception e) {
        System.out.println("Message not sent");
    }
}
```

Message Consumer

JMSConsumer (MessageConsumer, QueueReceiver, TopicSubscriber)

- wird von Session (JMSContext) erzeugt
- erhält eine Ressource (Queue oder Topic)
- empfängt Nachrichten
- realisiert synchroner Empfang mit receive-Methoden
- benötigt für den asynchronen Empfang einen `MessageListener` (`onMessage`-Methode)

Beispiel:

```
//synchroner Empfang

JMSConsumer consumer=context.createConsumer(topic);
TextMessage msg = (TextMessage) consumer.receive(10000);
// empfängt 10 sekunden lang
msg.acknowledge();//Empfangsbestätigung
```

Beispiel:

```
//asynchroner Empfang

public class MyTopicListener implements MessageListener{
     public static void main( String[] args ) throws Exception  {
         consumer=context.createConsumer(topic);
         //statt receive:
         consumer.setMessageListener (new MyTopicListener() );
         Thread.sleep( 20000 );
     }

    public void onMessage( Message message )  {
                try {
                      //hier message auswerten
                      message.acknowledge();
                } catch( JMSException ex ) {
                      System.out.println( ex.getMessage() );
     }
}
```

Interface Destination

Basis-Interface für Ressourcen
kapselt provider-spezifische Adresse
kann provider-spezifische Konfigurationsinformationen enthalten

Interface Queue (extends Destination)

kapselt provider-spezifischen Queue-Namen
Methoden:

```
String getQueueName() throws JMSException
String toString()
```

Interface Topic (extends Destination)

kapselt provider-spezifischen Topic-Namen
Methoden:

```
String getTopicName() throws JMSException
String toString()
```

Queue (Point-to-Point, PTP):

Queues (Warteschlangen): Nachrichtenkanäle, normalerweise zwischen genau einem Sender und einem Empfänger. Nach dem Empfang wird die Nachricht meist aus dem Nachrichtenkanal entfernt

Topic (Point-to-Multipoint, Publish-and-Subscribe):

Topics (Themen): themenorientierte Nachrichtenkanäle, haben normalerweise viele Konsumenten. Nach dem Erhalt durch einen Empfänger, bleibt die Nachricht meist weiter im Nachrichtenkanal für weitere Konsumenten bis zum evtl. Verfallsdatum

Clients

Queue-Producer (Sender)

Queue-Consumer (Receiver)

Topic-Publisher

Topic-Subscriber

Queue-Producer (Sender)

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import javax.naming.*;
import javax.jms.*;  //jakarta statt javax!

public class MySenderBsp {
    public static void main(String[] args) {
       try
        {
            InitialContext ctx=new InitialContext();
            QueueConnectionFactory f=(QueueConnectionFactory)ctx.lookup("jms/ConnectionF");
            JMSContext context = f.createContext();
            Queue queue=(Queue)ctx.lookup("jms/MyQ");

            JMSProducer sender=context.createProducer();
            TextMessage msg=context.createTextMessage();
            BufferedReader b=new BufferedReader(new InputStreamReader(System.in));
            while(!s.equals("end"))
            {
                System.out.println("Gib Message ein, end zum Beenden:");
                String s=b.readLine();
                if (!s.equals("end")) {
                  msg.setText(s);
                  sender.send(queue,msg);  //sender.send(queue,s);

            }
            context.close();
         }catch(Exception e){System.out.println(e);}
    }
}
```

Queue-Consumer (Receiver)

```
public class MyReceiverBsp {
public static void main(String[] args) {
    try{
        InitialContext ctx=new InitialContext();
        QueueConnectionFactory f=(QueueConnectionFactory)ctx.lookup("jms/ConnectionF");
        JMSContext context = f.createContext();

        Queue t=(Queue)ctx.lookup("jms/MyQ");
        JMSConsumer receiver=context.createConsumer(t);
        MyListener listener=new MyListener();
        receiver.setMessageListener(listener);

        System.out.println("Fertig zum Empfangen");
        System.out.println("zum Beenden ctrl+c");
        while(true){
              Thread.sleep(60000);
        }
       }catch(Exception e){System.out.println(e);}
}}
```

```
public class MyListener implements MessageListener {
    public void onMessage(Message m) {
        try{
           TextMessage msg=(TextMessage)m;
           System.out.println("Folgendes empfangen:"+msg.getText());
        }catch(JMSException e){
           System.out.println(e);
        }
    }
}
```

Queue-Producer (Sender)

```
public class MySender {
        public static void main(String[] args) {
           try
            {
                InitialContext ctx=new InitialContext();
                QueueConnectionFactory f=(QueueConnectionFactory)ctx.lookup("jms/ConnectionF");
                JMSContext context = f.createContext();
                Queue queue=(Queue)ctx.lookup("jms/MyQ");
                JMSProducer sender=context.createProducer();
                TextMessage msg=context.createTextMessage();

                MyCompletionListener comlistener=new MyCompletionListener();
                sender.setAsync(comlistener);

                BufferedReader b=new BufferedReader(new InputStreamReader(System.in));
                while(true)
                {
                    System.out.println ("Gib Message ein, end zum Beenden:");
                    String s=b.readLine();
                    if (s.equals("end")) break;
                    msg.setText(s);
                    sender.send(queue,msg);
                }
                context.close();
            }catch(Exception e){System.out.println(e);}
        }
}
```

```
import javax.jms.CompletionListener;
import javax.jms.Message;

public class MyCompletionListener implements CompletionListener{
    @Override
    public void onCompletion(Message msg) {
        System.out.println("Message erfolgreich gesendet");
    }

    @Override
    public void onException(Message msg, Exception e) {
        System.out.println("Message nicht gesendet");
    }
}
```

Literatur:

Christian Ullenboom: Java ist auch eine Insel

https://jakarta.ee/specifications/messaging/3.0/jakarta-messaging-spec-3.0.html