In diesem Kursteil

Java in Verteilte Systeme

1.1. Kursübersicht

Der Kurs ***Java in Verteilten Systemen*** ist eine Einführung in die Technoligien

- JDBC : Datenbankzugriff
- RMI : verteilte Objektsysteme
- Objektserialisierung

mit dem Ziel, Ihnen Wissen zu vermitteln, welche Sie befähigen wird, verteilte Anwendungen zu entwickeln. Dieser Kurs beschreibt Technologien, mit deren Hilfe Sie verteilte Anwendungen entwickeln können, basierend auf Java™ Application Programming Interfaces (API).

Voraussetzung für diesen Kurs sind Kenntnisse im Bereich *Java Programmierung*, *Objekt-Orientiertes Design und Analyse* und mindestens teilweise folgende praktischen Erfahrungen:

- Entwicklung von Java Applikationen
- Grundkenntnisse in Datenbanken
- Grundkenntnisse von SQL

Sie sollten idealerweise auch bereits Erfahrungen in der objektorientierten Programmierung haben.

1.1.1. Lernziele

Nach dem Durcharbeiten dieser Kursunterlagen sollten Sie in der Lage sein

- unterschiedliche Technologien für die Programmierung verteilter Systeme in Java zu kennen und zu vergleichen
- einfache Datenbank-Anwendungen zu schreiben
- Remote Methode Invocation Applikationen zu schreiben und Daten mittels Objektserialisierung langfristig zu speichern.
- einfache Java IDL Applikatione zu schreiben

1.2. Modul 1 : Java Database Connectivity JDBC

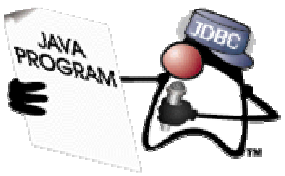
In diesem Modul

- Modul 1 : Java Database Connectivity (JDBC)
- Modul Einleitung
- JDBC Driver
- Das java.sql Package
- JDBC Abläufe
- DB Verbindungsaufbau mittels JDBC
- JDBC Treiber
- JDBC Connection
- JDBC Anweisungen
- Abbildung von SQL Datentypen auf Java Datentypen
- Einsatz des JDBC APIs
- Datenbank Design
- Applets
- Praktische Übung
- Quiz
- Zusammenfassung

1.2.1. Einleitung

Das JDBC API ist ein Set von Interfaces, mit deren Hilfe in Datenbankanwendungen die Details des Datenbankzugriffs isoliert werden können. Die JDBC Interfaces gestatten es dem Entwickler, sich auf die eigentlich Applikation zu konzentrieren, also sicherzustellen, dass die Datenbankabfragen korrekt formuliert sind und die Datenbank korrekt designed ist.

Mit JDBC kann der Entwickler gegen ein Interface entwickeln, mit den Methoden und Datenfeldern, die das Interface zur Verfügung stellt. Der Entwickler braucht sich dabei nicht gross darum zu kümmern, dass es Interfaces vor sich hat. Die Trieber Anbieter stellen Klassen zur Verfügung, welche die Interfaces implementieren. Daher wird der Programmierer eigentlich gegen einen Treiber programmieren.



Das JDBC API gestattet es den Entwicklern, beliebige Zeichenketten direkt an den Treiber zu übergeben, also low level Programmierung. Daher kann der Entwickler auch SQL Spracheigenschaften einer bestimmten Implementation verwenden, nicht nur SQL Konstrukte des ANSI SQL Standards.

1.2.1.1. Lernziele

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- zu erklären, was JDBC ist.
- die fünf wichtigsten Aufgaben bei der Programmierung eines JDBC Programms aufzählen können.
- zu erklären, wie der JDBC Driver mit dem JDBC Driver Manager zusammenhängt.
- zu erklären, wie die Datenbank-Datentypen in Java Datentypen umgewandelt werden.
- unterschiedliche Architekturen für verteilte Systeme, beispielsweise die zwei-Tier und drei-Tier Architektur gegeneinander abzugrenzen, speziell im Zusammenhang mit JDBC.
- einfache JDBC Datenbank Applikationen zu schreiben und JDBC Probleme zu lösen.

1.2.1.2. Referenzen

Teile dieses Moduls stammen aus der JDBC Spezifikation

- " The JDBC ™ 1.2 Specification" bei Sun http://java.sun.com/products/jdbc/
- SQL können Sie beispielsweise in ***The Practical SQL Handbook*** von Emerson, Darnovsky und Bowman (Addison-Wesley, 1989) nachlesen.

1.2.2. JDBC Driver

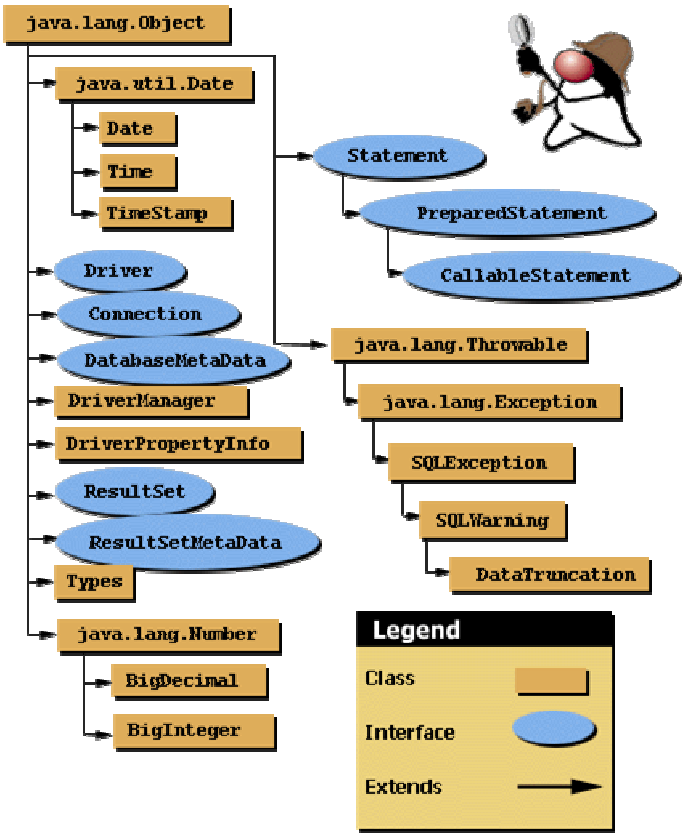
Ein JDBC Driver ist eine Sammlung von Klassen, welche die JDBC Interfaces implementieren, also die Interfaces, welche es Java Programmen erlauben, auf Datenbanken zuzugreifen. Jeder Datenbank Treiber muss mindestens eine Klasse zur Verfügungstellen, welche das `java.sql.Driver` Interface implementiert. Diese Klasse wird von der generischen `java.sql.DriverManager` Klasse benutzt, falls diese einen Treiber benötigt, um auf eine bestimmte Datenbank mittels eines Uniform Resource Locators (URL) zuzugreifen. JDBC sieht ähnlich aus wie ODBC. Daher kann man JDBC auch zusammen mit ODBC effizient implementieren. Der Overhead ist gering, die Implementation also effizient.

Eine einzige Java Applikation (oder ein Applet) kann gleichzeitig auf mehrere Datenbanken mittels eines oder mehreren Treibern zugreifen.

1.2.3. Das *java.sql* Package

Zur Zeit gibt es acht Interfaces im Zusammenhang mit JDBC:

1. Driver
2. Connection
3. Statement
4. PreparedStatement
5. CallableStatement
6. ResultSet
7. ResultSetMetaData
8. DatabaseMetaData

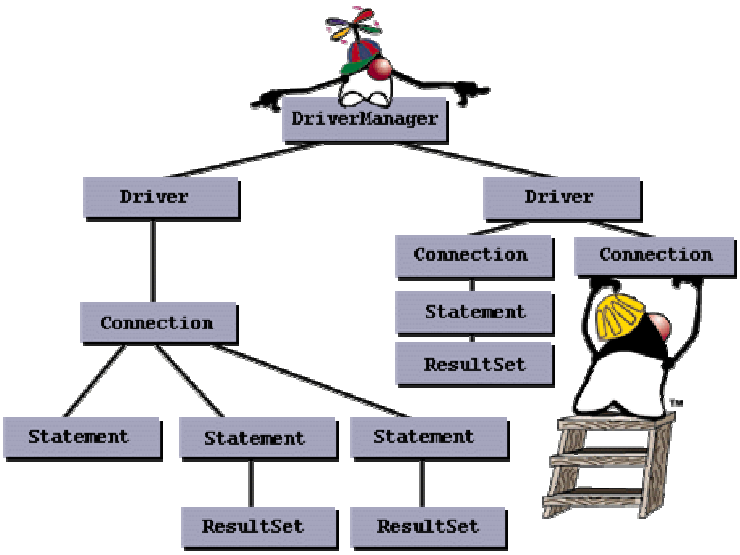


Die obigen Interfaces müssen alle implementiert werden. Aber es liegt an Ihnen zu entscheiden, welche der definierten Methoden Sie für Ihre Applikation konkret benötigen.

1.2.4. JDBC Abläufe

Jedes der Interfaces stellt Ihnen bestimmte Verbindungsoptionen für eine Verbindung zu einer Datenbank, zum Ausführen von SQL Anweisungen und zum Bearbeiten der Ergebnisse zur Verfügung.

Eine URL Zeichenkette wird an die getConnection() Methode der DriverManagers übergeben. Der DriverManager sucht einen Driver und damit kann eine Verbindung eine Connection hergestellt werden.



Mit der Verbindung können Sie eine Datenbank Anweisung, ein Statement, absetzen. Falls das Statement ausgeführt wird, mit der executeQuery() Methode, liefert die Datenbank einen Ergebnisset, den ResultSet, da beispielsweise eine Datenbankabfrage in der Regel mehr als einen Datensatz zurückliefert. Im anderen Extremfall ist der ResultSet leer, weil kein Datensatz die Abfragekriterien erfüllt, eine Mutation durchgeführt wurde oder ein neuer Datensatz in die Datenbank eingefügt wurde.

1.2.5. Verbindungsaufbau mittels JDBC Interface

Nun betrachten wir einige typischen Aufgaben, die Sie mit JDBC erledigen, falls Sie als Programmierer damit arbeiten. Wir werden mit MS-Access, Textdateien, Excel und ähnlichen ODBC Datenbanken arbeiten. Aber eigentlich spielt dies keine zentrale Rolle, da wir mit ODBC arbeiten können und damit die Datenbank selber keine Rolle spielt. Die im Text verwendeten Driver und SQL DB ist : My-SQL mit derm JDBC Driver von imaginary.

```
package connecttodbms;

import java.sql.*;

public class ConnectToCoffeeBreakDB {

      public static void main(String args[]) {
            String url = "jdbc:odbc:coffeebreak";
            Connection con;
            System.out.println("Try : Class.forName - Laden des Drivers");
            try {
                  Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");

            } catch(java.lang.ClassNotFoundException e) {
                  System.err.print("ClassNotFoundException: ");
                  System.err.println(e.getMessage());
            }
            System.out.println("Try : DriverManager.getConnection -
                                          Verbindung herstellen");
            try {
                  con = DriverManager.getConnection(url, "", "");
                  System.out.println("con.close() -
                                       Verbindung schliessen");
                  con.close();
            } catch(SQLException ex) {
                  System.err.println("SQLException: Verbindungsaufbau " +
                                    ex.getMessage());
            }
      }
}
```

Im folgenden Beispiel sehen Sie, wie Sie beispielsweise in einer JDBC Applikation eine Driver Instanz bilden können, anschliessend ein Verbindungsobjekt kreieren, ein Statementobjekt bilden und eine Abfrage ausführen und anschliessend die Ergebnisse der Abfrage im ResultSet Objekt bearbeiten:

```
import java.sql.*;
import com.imaginary.sql.msql.MsqlDriver;// miniSQL Treiber,
// falls Sie nicht mit ODBC arbeiten, wie oben

public class JDBCBeispiel {

   public static void main (String args[]) {

      if (args.length < 1) {
      System.out.println ("Usage:");
      System.out.println ("java JDBCBeispiel <db server hostname>");
      System.exit (1);
    }

    try {
      // Instanz des iMsqlDrivers bilden (mySQL z.B.)
      new com.imaginary.sql.msql.MsqlDriver ();
```

```
      // Definition der "url"
      String url = "jdbc:msql://" + args[0] + ":1112/FlugreservationsDB";

      // Verbindungsaufbau mit Hilfe des DriverManager
      Connection conn = DriverManager.getConnection (url);

      // kreieren eines Statement Objekts
      Statement stmt = conn.createStatement ();

      // Query ausführen (mit dem Statement Objekt)
      // und Definition des ResultSet Objekts
      ResultSet rs = stmt.executeQuery ("SELECT * from FlugzeugTyp");
      // Ausgabe der Ergebnisse - Zeile für Zeile
      while (rs.next()) {
        System.out.println ("");
        System.out.println ("Flugzeugtyp:           "
        + rs.getString (1));
        System.out.println ("First Class Plätze:    "
        + rs.getInt (2));
        System.out.println ("Business Class Plätze: "
        + rs.getInt (3));
        System.out.println ("Economy Class Plätze:    "
        + rs.getInt (4));
      }

  } catch (SQLException e) {
      e.printStackTrace();
  }
 }
}
```

Falls Sie das Programm starten, werden die Inhalte der Tabelle FlugzeugTyp angezeigt:

```
Flugzeugtyp:           B747
First Class Plätze:    30
Business Class Plätze: 56
Economy Class Plätze:  350

Flugzeugtyp:           B727
First Class Plätze:    24
Business Class Plätze: 0
Economy Class Plätze:  226

Flugzeugtyp:           B757
First Class Plätze:    12
Business Class Plätze: 0
Economy Class Plätze:  112

Flugzeugtyp:           MD-DC10
First Class Plätze:    34
Business Class Plätze: 28
Economy Class Plätze:  304
```

wobei ich vorgängig die Access Datenbank und die Tabelle angelegt und Daten eingegeben habe.

1.2.6. Kreieren eines JDBC Driver Objekts

Das Treiberobjekt können Sie entweder direkt im Programm oder aber als Programm Property beim Starten angeben:

```
java –D jdbc.drivers= com.imaginary.sql.msql.MsqlDriver Abfrage
```

Wie dies genau geschieht werden wir noch anschauen. Der obige JDBC Treiber stammt von der Firma Imaginary und ist für die mSQL (mini-SQL) Datenbank entwickelt worden.
Um mit einer Datenbank kommunizieren zu können, muss auch jeden Fall eine Instanz des JDBC Treibers vorhanden sein. Der Treiber agiert im Hintergrund. Er behandelt alle Anfragen an die Datenbank.

Sie brauchen keine Refernz des Treibers treibers, es genügt also, wenn Sie einfach ein Objekt kreieren:

```
new com.imaginary.sql.msql.MsqlDriver();
```

Der Treiber existiert nach dieser Anweisung, sofern er erfolgreich geladen werden kann. Daher können wir auch einfach mit

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

die Klasse direkt laden. Im ersten Fall wird der Konstruktor ein statisches Objekt kreieren, welches im System erhalten bleibt. Eine Referenz selbst wird nicht benötigt. Alle Methoden der Klasse DriverManager sind `static`, können also direkt mit dem Klassenpräfix aufgerufen werden. Der Driver ist auch für das Registrieren des Drivers zuständig. Dies können Sie auch explizit durchführen:

```
Driver drv = DriverManager.getDriver(url);
DriverManager.registerDriver(drv);
```

Wenn Sie dies nochmals explizit machen, wird der selbe Driver zweimal registriert, wie Sie mit `DriverManager.getDrivers()` abfragen können:

```
// fakultatives Registrieren (wird implizit gemacht)
Driver drv = DriverManager.getDriver(url);
DriverManager.registerDriver(drv);
for (Enumeration drvs=DriverManager.getDrivers();drvs.hasMoreElements(); )
      System.out.println(drvs.nextElement().getClass());
```

Dies liefert folgende Ausgabe:

```
class sun.jdbc.odbc.JdbcOdbcDriver
class sun.jdbc.odbc.JdbcOdbcDriver
```

Wie Sie sehen, wurde der selbe Driver zweimal geladen. Gleichzeitig sehen Sie auch, dass mehr als ein Driver geladen werden kann, sogar mehrere Driver, die den Zugriff auf die selbe Datenbank gestatten, beispielsweise einmal über JDBC, einmal über ODBC.

Als Property, auf der Kommandozeile, sieht die Spezifikation mehrerer Driver so aus:

```
jdbc.drivers = com.imaginary.sql.msql.MsqlDriver:Acme.wonder.driver
```

Properties setzt man mit der -D Option des Java Interpreters oder mittels der -J Option beim Appletviewer.

Beispiel:

```
java –D jdbc.drivers= com.imaginary.sql.msql.MsqlDriver:Acme.wonder.driver
```

Falls Ihr Programm versucht, eine Verbindung mit einer Datenbank über JDBC aufzunehmen, wird der erste Driver der gefunden wird eingesetzt. Die Reihenfolge ist folgendermassen festgelegt:

1. die Driver der Property Angabe werden von links nach rechts geprüft und der erste mögliche davon eingesetzt.
2. dann werden die bereits geladenen Driver geprüft, in der Reihenfolge, in der sie geladen wurden.

Falls ein Driver mit untrusted Code geladen wurde, also nicht sicher sein muss, wird der Driver nicht weiterverwendet, ausser das Programm hat den Driver neu von der selben Quelle geladen.

Wie auch immer, sobald der Driver geladen ist, ist er dafür verantwortlich, sich selbst beim Driver Manager zu registrieren. Schauen wir uns den Mini-SQL Driver von Imaginary an:

```
/* Copyright (c) 1997 George Reese */
package com.imaginary.sql.msql;
import java.sql.Connection;
import java.sql.Driver;
import java.sql.DriverManager;
import java.sql.DriverPropertyInfo;
import java.sql.SQLException;
import java.util.Properties;
/**
 * The MsqlDriver class implements the JDBC Driver interface from the
 * JDBC specification.  A Driver is specifically concerned with making
 * database connections via new JDBC Connection instances by responding
 * to URL requests.<BR>
 * Last modified 97/10/28
 * @version @(#) MsqlDriver.java 1.4@(#)
 * @author George Reese (borg@imaginary.com)
 */
public class MsqlDriver implements Driver {
    /****************** Static methods and attributes *****************/
    // The static constructor does according to the JDBC specification.
    // Specifically, it creates a new Driver instance and registers it.
    static {
      try {
          new MsqlDriver();
      }
      catch( SQLException e ) {
          e.printStackTrace();
      }
    }
    /****************** Instance methods and attributes *****************/
    /**
     * Constructs an MsqlDriver instance.  The JDBC specification requires
     * the driver then to register itself with the DriverManager.
     * @exception java.sql.SQLException an error occurred in registering
     */
    public MsqlDriver() throws SQLException {
      super();
```

```
  DriverManager.registerDriver(this);
}
/**
 * Gives the major version for this driver as required by the JDBC
 * specification.
 * @see java.sql.Driver#getMajorVersion
 * @return the major version
 */
public int getMajorVersion() {
  return 1;
}
/**
 * Gives the minor version for this driver as required by the JDBC
 * specification.
 * @see java.sql.Driver#getMinorVersion
 * @return the minor version
 */
public int getMinorVersion() {
  return 0;
}

/**
 * The getPropertyInfo method is intended to allow a generic GUI tool
 * to discover what properties it should prompt a human for in order to
 * get enough information to connect to a database.Note that depending
 * on the values the human has supplied so far, additional values
 * may become necessary, so it may be necessary to iterate though
 * several calls to getPropertyInfo.
 * @param url The URL of the database to connect to.
 * @param info A proposed list of tag/value pairs that will be sent on
 *          connect open.
 * @return An array of DriverPropertyInfo objects describing possible
 *        properties.This array may be an empty array if no properties
 *          are required.
 * @exception java.sql.SQLException never actually thrown
 */
public DriverPropertyInfo[] getPropertyInfo(String url,Properties info)
throws SQLException {
  return new DriverPropertyInfo[0];
}

/**
 * Returns true if the driver thinks that it can open a connection
 * to the given URL.  In this case, true is returned if and only if
 * the subprotocol is 'msql'.
 * @param url The URL of the database.
 * @return True if this driver can connect to the given URL.
 * @exception java.sql.SQLException never actually is thrown
 */
public boolean acceptsURL(String url) throws SQLException {
  if( url.length() < 10 ) {
      return false;
  }
  else {
      return url.substring(5,9).equals("msql");
  }
}

/**
 * Takes a look at the given URL to see if it is meant for this
 * driver.  If not, simply return null.  If it is, then go ahead and
 * connect to the database.  For the mSQL implementation of JDBC, it
```

```
     * looks for URL's in the form of <P>
     * <PRE>
     *     jdbc:msql://[host_addr]:[port]/[db_name]
     * </PRE>
     * @see java.sql.Driver#connect
     * @param url the URL for the database in question
     * @param p the properties object
     * @return null if the URL should be ignored, a new Connection
     * implementation if the URL is a valid mSQL URL
     * @exception java.sql.SQLException an error occurred during connection
     * such as a network error or bad URL
     */
    public Connection connect(String url,Properties p)throws SQLException {
      String host, database, orig = url;
      int i, port;

      if( url.startsWith("jdbc:") ) {
          if( url.length() < 6 ) {
            return null;
          }
          url = url.substring(5);
      }
      if( !url.startsWith("msql://") ) {
          return null;
      }
      if( url.length() < 8 ) {
          return null;
      }
      url = url.substring(7);
      i = url.indexOf(':');
      if( i == -1 ) {
          port = 1114;
          i = url.indexOf('/');
          if( i == -1 ) {
            throw new SQLException("Invalid mSQL URL: " + orig);
          }
          if( url.length() < i+1 ) {
            throw new SQLException("Invalid mSQL URL: " + orig);
          }
          host = url.substring(0, i);
          database = url.substring(i+1);
      }
      else {
          host = url.substring(0, i);
          if( url.length() < i+1 ) {
            throw new SQLException("Invalid mSQL URL: " + orig);
          }
          url = url.substring(i+1);
          i = url.indexOf('/');
          if( i == -1 ) {
            throw new SQLException("Invalid mSQL URL: " + orig);
          }
          if( url.length() < i+1 ) {
            throw new SQLException("Invalid mSQL URL: " + orig);
          }
          try {
            port = Integer.parseInt(url.substring(0, i));
          }
          catch( NumberFormatException e ) {
            throw new SQLException("Invalid port number: " +
                                   url.substring(0, i));
          }
```

```
            database = url.substring(i+1);
          }
          return new MsqlConnection(orig, host, port, database, p);
        }

        /**
         * Returns information noting the fact that the mSQL database is not
         * SQL-92 and thus cannot support a JDBC compliant implementation.
         */
        public boolean jdbcCompliant() {
          return false;
        }
}
```

Der JDBC Treiber der Firma Imaginary kreiert eine Instanz von sich selbst, oben im statischen Block und registriert sich entweder automatisch oder explizit.

Nach dem JDBC Treiber müssen wir nun die Datenbank angeben, auf die wir zugreifen möchten. Dies geschieht mit Hilfe einer URL, welche den Datenbanktyp angibt. Dabei handelt es sich um eine URL ähnliche Notation, also keine echte URL.

```
jdbc:subprotocol:parameters
```

Beispiel:
`jdbc:odbc:FlugreservationsDB`

dabei steht `subprotocol` einen bestimmten Datenbank Verbindungsmechanismus, im Beispiel also ODBC. Die Parameter hängen vom DBMS und dem Protokoll ab:

```
// konstruiere die URL für den JDBC Zugriff
String url = new String ("jdbc:msql://" + args[0]+":1112/TicketingDB");
```

In diesem Fall haben wir auch noch den Port und die Datenbank angegeben, da wir nicht über ODBC die entsprechenden Informationen beschaffen können. In diesem Beispiel ist das Subprotokoll

```
msql
```

Falls wir an Stelle des Mini-SQL Treibers über ODBC auf die Datenbank zugreifen möchten, hätten wir

```
jdbc:odbc:Object.TicketingDB
```

angeben können. Aber auch ein Netzwerk-Protokoll könnte verwendet werden:

```
jdbc:nisnaming:Ticketing-Info
```

Der JDBC URL Mechanismus liefert ein Framework, so dass unterschiedliche Driver eingesetzt werden können. Jeder Treiber muss einfach die URL Syntax verstehen, da er irgend etwas daraus machen muss!

1.2.7. Die JDBC Verbindung - Connection

Nachdem wir eine "URL" zur Datenbank besitzen und damit den Datenbanktypus festgelegt haben, müssen wir die Verbindung zur Datenbank aufbauen. Dies geschieht mit einem

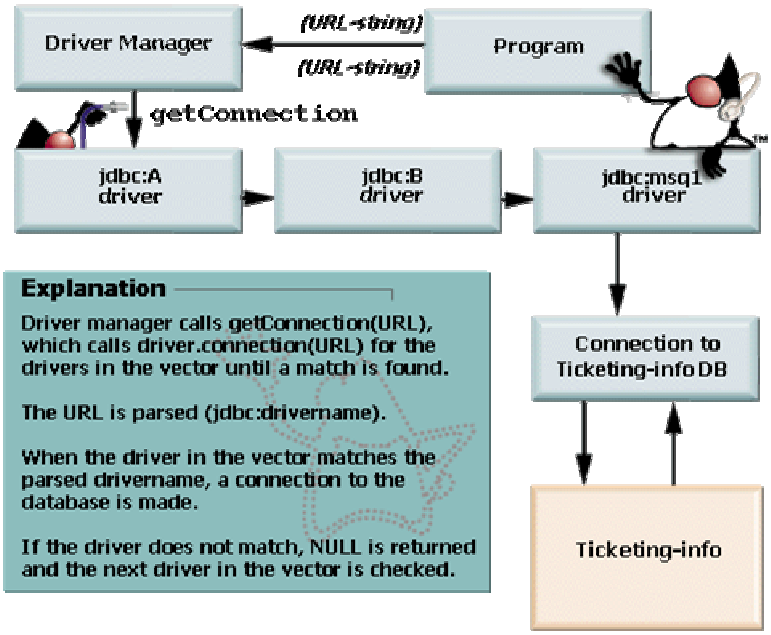
`java.sql.Connection`

Objekt, indem die

`java.sql.DriverManager.getConnection()`

Methode des JDBC Driver Manager aufgerufen wird.

Schematisch, gemäss original Sun Dokumentation:




```
// Verbindungsaufbau zur Datenbank mittels des
// DriverManager
Connection conn = DriverManager.getConnection (url);
```

Der Ablauf ist :

1. Der `DriverManager` ruft die Methode `Driver.getConnection()` auf, für jeden registrierten Driver, mit der URL, also einer Netzwerkadresse, als Parameter.

2. Falls der Driver eine Verbindung herstellen kann, dann liefert er ein Connection Objekt zurück, sonst null.

Ein `DriverManager` löst die URL Referenz in der `getConnection()` Methode auf. Falls der `DriverManager` `null` zurückliefert, versucht er gleich mit dem nächsten Treiber eine Verbindung aufzubauen, bis schliesslich die Liste erschöpft ist, oder eine Verbindung hergestellt werden konnte. Die Verbindung selbst besteht nicht zwischen dem Treiber und der Datenbank - die Verbindung ist die Aufgabe der Implementation des `Connection` Interfaces.

1.2.8. JDBC Anweisungen

Um eine Abfrage an eine Datenbank zu senden, muss man zuerst ein `Statement` Objekt von der `Connection.createStatement()` Methode erhalten.

```
// kreieren eines Statement Objekts
      try {
            Statement stmt = conn.createStatement ();
      catch(SQLException se) {
            System.out.println(e.getMessage() );
      }
```

Bemerkung SQLException

SQL Ausnahmen treten bei Datenbankfehlern auf. Dies kann beispielsweise eine unterbrochene Verbindung oder ein heruntergefahreren Datenbankserver sein. SQL Exceptions liefern zusätzliche Debugging Informationen:

- eine Zeichenkettenbeschreibung des Fehlers
- eine SQL Zustandsbeschreibung gemäss dem Xopen Standard
- ein anbieterspezifischer Fehlercode

1.2.8.1. Direkte Ausführung - `Statement`

Mit der Methode `Statement.executeQuery(...)` kann man die SQL Aweisung an die Datenbank senden. JDBC verändert die SQL Anweisung nicht, sondern reicht sie einfach weiter. JDBC interpretiert also SQL nicht.

```
      // kreieren eines Statement Objekts
      Statement stmt = conn.createStatement();
      // Query ausführen (mit dem Statement Objekt)
      // und Definition des ResultSet Objekts
      ResultSet rs = stmt.executeQuery ("SELECT * from FlugzeugTyp");
```

1.2.8.2. Vorbereitete Ausführung - `PrepareStatement`

Die `Statement.executeQuery()` Methode liefert ein ResultSet, welches anschliessend bearbeitet werden muss. Falls die selbe Anweisung öfters verwendet werden soll, ist es vorteilhaft zuerst die `PrepareStatement` Anweisung auszuführen.

```
public static void main(String args[]) {
    System.out.println("[PrepareStatement]Start");
    String url = "jdbc:odbc:coffeebreak";
    Connection con;
    String query = "SELECT * FROM SUPPLIERS";
    Statement stmt;
    System.out.println("[PrepareStatement]Treiber laden");
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    } catch(java.lang.ClassNotFoundException e) {
      System.err.print("ClassNotFoundException: ");
      System.err.println(e.getMessage());
    }
    try {
      System.out.println("[PrepareStatement]getConnection()");
      con = DriverManager.getConnection(url, "myLogin", "myPassword");
      System.out.println("[PrepareStatement]executeQuery");
```

```
      PreparedStatement prepStmnt = con.prepareStatement(query);
      ResultSet rs = prepStmnt.executeQuery();
      ResultSetMetaData rsmd = rs.getMetaData();
      int numberOfColumns = rsmd.getColumnCount();
      int rowCount = 1;
      while (rs.next()) {
        System.out.println("Row " + rowCount + ":  ");
        for (int i = 1; i <= numberOfColumns; i++) {
          System.out.print("   Column " + i + ":  ");
          System.out.println(rs.getString(i));
        }
        System.out.println("");
        rowCount++;
      }
      con.close();
    } catch(SQLException ex) {
      System.err.println("SQLException: Procedures werden nicht unterstützt
");
      System.err.println(ex.getMessage());
    }
  }
```

mit der Ausgabe

```
[PrepareStatement]Start
[PrepareStatement]Treiber laden
[PrepareStatement]getConnection()
[PrepareStatement]executeQuery
Row 1:
   Column 1:  49
   Column 2:  Superior Coffee
   Column 3:  1 Party Place
   Column 4:  Mendocino
   Column 5:  CA
   Column 6:  95460

Row 2:
   Column 1:  101
   Column 2:  Acme, Inc.
   Column 3:  99 Market Street
   Column 4:  Groundsville
   Column 5:  CA
   Column 6:  95199

Row 3:
   Column 1:  150
   Column 2:  The High Ground
   Column 3:  100 Coffee Lane
   Column 4:  Meadows
   Column 5:  CA
   Column 6:  93966
```

Sie müssen klar auseinanderhalten

1) auf der einen Seite "prepared":
   in diesem Fall bereiten Sie die Ausführung einer SQL Anweisung vor und führen Sie nachher mehrere Male (oder auch nur einmal) aus.
2) auf der andern Seite "stored"
   in diesemFall bereiten Sie die Ausführung auch vor. Aber die Anweisung selber wird in der Datenbank abgespeichert, 'stored'.

Zusätzlich können Sie auch SQL Anweisungen parametrisiert vorbereiten:

```
public boolean prepStatement(Reservation obj){
      PreparedStatement prepStmnt = msqlConn.prepareStatement( "UPDATE
                           Fluege SET anzahlVerfuegbareFCSitze = ? WHERE
                           flugNummer = ?" );
prepStmnt.setInt(1,(Integer.parseInt(obj.anzahlVerfuegbareFCSitze)-1));
      prepStmnt.setLong(2, obj.FlugNr);
      int rowsUpdated = prepStmnt.executeUpdate();
      if (rowsUpdated > 0){
            return true;
      } else {
            return false;
      }
}
```

Platzhalter, Parameter des `PreparedStatement` werden einfach durchnummeriert.
Die Parameter könnten `in`, `out` oder `inout` Parameter sein. JDBC unterstützt nur `in` und `out`. `inout` werden im Rahmen von CORBA IDL berücksichtigt. `in` Variablen werden 'by value' übergeben, also wertmässig. `out` Parameter werden 'by reference', an eine Referemz übergeben.

Die übergebenen Datentypen müssen mit den von SQL unterstützten Datentypen übereinstimmen:

Methode	SQL Typ(en)
setASCIIStream	verwendet einen ASCII Stream um ein LONGVARCHAR zu produzieren
setBigDecimal	NUMERIC
setBinaryStream	LONGVARBINARY
setBoolean	BIT
setByte	TINYINT
setBytes	VARBINARY oder LONGVARBINARY (abhängig von der Grösse und der möglichen Grösse von VARBINARY)
setDate	DATE
setDouble	DOUBLE
setFloat	FLOAT
setInt	INTEGER
setLong	BIGINT
setNull	NULL
setObject	das Java Objekt wird in einen SQL Datentyp konvertiert bevor es an die DB gesandt wird
setShort	SMALLINT
setString	VARCHAR oder LONGVARCHAR (abhängig von der Grösse, die der Driver VARCHAR zuordnet)
setTime	TIME
setTimestamp	TIMESTAMP
setUnicodeStream	UNICODE

1.2.8.3. Gespeicherten Anweisung - `CallableStatement`

Das dritte Interface, welches eine Erweiterung des PreparedStatement darstellt, geht davon aus, dass eine SQL Anweisung vorbereitet und in der Datenbankabgespeichert werden kann. Solche Anweisungen nennt man deswegen auch 'Stored Procedures', eben weil sie in der Datenbank abgespeichert werden.

In JDBC definiert man `CallableStamenet` sogar etwas allgemeiner: ein `CallableStatement` gestattet die Ausführung von nicht-SQL Anweisungen gegenüber einer Datenbank. Typischerweise handelt es sich dabei um Anweisungen, welche nur von einzelnen Datenbanken unterstützt werden.

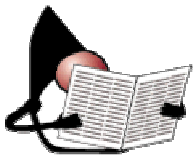
Da das Interface `CallableStatement` eine Erweiterung des `PreparedStatement` ist, können auch `in` Parameter gesetzt werden. Da `PreparedStatement` das Interface Statement erweitert, können auch dessen Methoden problemlos eingesetzt werden, beispielsweise `Statement.getMoreResults()`.

Hier ein Beispiel für eine Abfrage einer Datenbank mit einer precompiled SQL Anweisung, die Abfrage der Anzahl Plätze eines Fluges:

```
1  String planeID = "727";
2  CallableStatement querySeats = msqlConn.prepareCall("{call
      return_seats[?, ?, ?, ?]}");
   //              1, 2, 3, 4
3  try {
4      querySeats.setString(1, planeID);
5      querySeats.registerOutParameter(2, java.sql.Type.INTEGER);
6      querySeats.registerOutParameter(3, java.sql.Type.INTEGER);
7      querySeats.registerOutParameter(4, java.sql.Type.INTEGER);
8      querySeats.execute();
9      int FCSeats = querySeats.getInt(2);
10     int BCSeats = querySeats.getInt(3);
11     int CCSeats = querySeats.getInt(4);
12 } catch (SQLException SQLEx){
13         System.out.println("Abfrage schlug fehl!");
14         SQLEx.printStackTrace();
15 }
```

Bevor man eine Stored Procedure aufrufen kann, muss man explizit den Ausgabeparameter registrieren:

```
registerOutParameter(parameter, java.sqlType.<SQLDatentyp>)
```



Das Ergebnis eines solchesn Aufrufes kann eine Tabelle sein, welche mittels eines `java.sql.ResultSet` Objekts abgefragt werden kann. Die Tabelle besteht aus Zeilen und Spalten. Die Zeilen werden je nach Abfrage sortiert oder einfach sequentiell in das Ergebnisset geschrieben. Ein `ResultSet` enthält einen Zeiger auf die aktuelle Datenzeile und zeigt am Anfang vor die erste Zeile. Mit dem ersten Aufruf der next() Methode zeigt der Zeiger auf die erste Zeile, der zweite Aufruf verschiebt den Zeiger auf die zweite Zeile usw.

Das ResultSet Objekt stellt einige `set...( )` Methoden zur Verfügung, mit deren Hilfe auf die einzelnen Datenelemente in der aktuellen Zeile zugegriffen werden kann, verändernd. Auf die Spaltenwerte kann man entweder mittels Spaltennamen oder mittels Index zugreifen. In der Regel ist es besser einen Index zu verwenden. Die Indices starten, wie Sie oben bereits gesehen haben, mit 1 und werden einfach durchnummeriert. Falls man Spaltennamen

verwendet könnte es im Extremfall mehr als eine Spalte mit demselben Namen haben. Das würde also sofort zu einem Konflikt führen.

Mit Hilfe verschiedener `get...( )` Methoden hat man Zugriff auf die Daten, zum Lesen. Die Spalten können in beliebiger Reihenfolge gelesen werden, innerhalb einer definierten Zeile.

Damit man Daten aus einem `ResultSet` Objekt herauslesen kann, müssen Sie die unterschiedlichen unterstützten Datentypen und den Aufbau des `ResultSet` kennen.

```
while (rs.next()) {
      System.out.println ("");
      System.out.println ("Flugzeugtyp:" + rs.getString (1));
      System.out.println ("Erste Klasse:" + rs.getInt (2));
      System.out.println ("Business Klasse:" + rs.getInt (3));
      System.out.println ("Economy Klasse:" + rs.getInt (4));
}
```

Bemerkung

Nachdem Sie das ResultSet Objekt gelesen haben, sind die Ergebnisse gelöscht. Sie können den `ResultSet` nur einmal lesen. Das Konzept wurde und wird aber im Moment erweitert, beispielsweise durch scrollable ResultSets. Aber eine Methode `ResultSet.numberOfRows()` gibt es auch weiterhin nicht.

Methode	Java Typ
getASCIIStream	java.io.InputStream
getBigDecimal	java.math.BigDecimal
getBinaryStream	java.io.InputStream
getBoolean	Boolean
getByte	Byte
getBytes	byte[]
getDate	java.sql.Date
getDouble	Double
getFloat	Float
getInt	Int
getLong	Long
getObject	Object
getShort	Short
getString	java.lang.String
getTime	java.sql.Time
getTimestamp	java.sql.Timestamp
getUnicodeStream	java.io.InputStream oder Unicode Zeichen

1.2.9. Abbildung von SQL Datentypen auf Java Datentypen

In der folgenden Tabelle sind die entsprechenden Datentypen zu den SQL Datentypen ersichtlich. Die SQL Datentypen müssen allgemeiner als die Java Datentypen definiert sein, da diese für allgemeine Datenbanken gültig sein müssen.

SQL Typ	Java Typ
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

1.2.10. Generelles zum Einsatz des JDBC APIs

Die Grundidee des JDBC war es, ein Set von Interfaces zu schaffen, welche möglichst universell, also unabhängig von der speziellen Datenbank einsetzbar sind. Zusätzlich sollte das JDBC auch nicht zu sehr an ein bestimmtes DB Design oder eine Design Methodik angelehnt sein. JDBC soll also unterschiedliche Designalternativen gestatten.

Allerdings hängt das Datenbankdesign zum Teil vom JDBC Treiber ab:

- bei einem **two-tier** (zweistufigen) Design verwenden Sie Java, um direkt auf die Datenbankmethoden oder Bibliotheken zuzugreifen; Sie verwenden also datenbankspezifischen Code und Protokolle.
- bei einem **three-tier** (dreistufigen) Design verwenden Sie Java und JDBC. Die JDBC Aufrufe werden anschliessend in DBMS Methodenaufrufe umgesetzt. Der JDBC ist somit DBMS unabhängig.
- beide Architekturen können zusätzlich verallgemeinert werden, indem Microsoft's Open Database Connectivity (ODBC) eingesetzt wird. Die JDBC Aufrufe werden in ODBC Aufrufe umgesetzt und ein in der Regel plattformunabhängiger ODBC Treiber sorgt für die Datenbankmethodenaufrufe.

Zurzeit gibt es über 100 Treiber für JDBC. Sie finden die aktuelle Liste unter

http://industry.java.sun.com/products/jdbc/drivers

In der Regel werden die Treiber auch von unabhängigen Firmen angeboten. Damit hat man die Gewähr, dass diese möglichst universell einsetzbar sind. Der JDBC zu ODBC Treiber wird auch direkt mit Java mitgeliefert. Natürlich gibt es viele Firmen, die behaupten bessere Implementationen zu besitzen, beispielsweise schnellere.

1.2.11. Datenbank Designs

Ein zweistufiges Design ist eines, bei dem der Benutzer mit einer Frontend Applikation (Client) direkt mit einer Datenbank (Server) verbunden ist.

Eine Datenbankanwendung mit einer *zweistufigen* Architektur kann JDBC auf zwei Arten einsetzen:

- ein **API Treiber** benutzt eine Bibliothek, in der Regel in C oder C++, welche für eine bestimmte Hardware- Plattform und ein bestimmtes Betriebsystem entwickelt wurde, jene für die eine Anwendung entwickelt wird. Der Client kann in irgend einer Programmiersprache geschrieben sein.
- ein **Protokoll Treiber** benutzt ein spezifisches und in der Regel Datenbank- abhängige Protokolle, welche vom Datenbankhersteller zur Verfügung gestellt werden. Der Datenbankanbieter kann beispielsweise einen TCP / IP (Transport protocol / Internet Protocol) Listener zur Verfügung stellen, der die Anfragen mehrerer Clients verwalten kann. Der Client muss dabei oberhalb TCP/IP ein Datenbank- spezifisches Protokoll einsetzen, um Abfragen an die Datenbank zu senden und die Ergebnisse zu empfangen und auszuwerten.

Ein *dreistufiges* Datenbankdesign umfasst zusätzlich einen Prozess zwischen der Endbenutzer- Applikation und dem Datenbank- Server. Das sieht auf Anhieb nach Overhead aus, also Leistungsverlust. Aber ein solches Design besitzt auch klare Vorteile:

- die Datenvalidierung wird vom mitteleren Prozess, vom Middle-Tier, übernommen.
- der Client kann mit einer einzigen Socketverbindung auf unterschiedliche Datenbanken zugreifen. Der Client kann dabei auch ein Applet in einem Browser sein.
- das Client- zum - Middle Protokoll ist unabhängig von der Datenbank. Die eigentliche Anbindung an die Datenbankist Aufgabe des middle tiers.
- im middle tier können Sie zusätzliche Funktionalitäten anbieten, beispielsweise Record Locking oder Änderungsmeldungen für einzelne Datensätze.

Datenbankanwendungen mit einem dreistufigen Design können einen Treiber einsetzen, der ausschliesslich in Java geschrieben ist. Der JDBC Treiber übersetzt die Datenbank-Methodenaufrufe in das Protokoll der mittleren Ebene. Die mittlere Ebene ist für die Auswahl der ausgewählten Datenbank zuständig.

1.2.12. Applets

Applets waren einmal der Startpunkt für die Verbreitung von Java im Internet, speziell im Web Bereich. JDBC kann auch in Applets integriert werden. Das Hauptproblem ist in diesem Falle die Security Policy.

Applets kann man aber auch im Intranet einsetzen, um beispielsweise Firmendaten abzufragen. Aber die Applets unterscheiden sich in einigen wesentlichen Aspekten von Standardapplikationen:

Wegen der Sicherheitseinschränkungen kann ein Applet nicht auf lokale Daten eines fremden Servers zugreifen. Auch ein ODBC Zugriff über lokale Registry (ODBC) Einträge ist nicht möglich.

Auch der Overhead wegen komplexen Netzwerkverbindungen, eventuell rund um die Welt, kann zu Problemen führen.

Einige der Sicherheitsprobleme kann man mit Hilfe digitaler Signaturen und kryptografischen Methoden umgehen bzw. lösen. Aber auch die Verzeichnisdienste in heterogenen Systemen können zu grossen Problemen führen.

Falls man eine dreistufige Architektur einsetzt, ist die mittlere Ebene vorzugsweise objektorientiert, selbst wenn die Aufrufe mittels RPC in C/C++ erfolgen.

1.2.13. Praktische Übung - Java Datenbank Connectivity

1.2.13.1. Lernziele

In dieser Übung sollten Sie lernen, wie aus einem Java Programm mittels JDBC auf eine Datenbank zugegriffen und diese abgefragt werden kann. Als Beispiel verwenden wir die Flugreservation, die wir als Interface definieren und in späteren Beispielen weiterverwenden.

Dieses Beispiel zeigt Ihnen, wie man

- auf eine SQL Datenbank zugreifen kann.
- wie man diese Datenbank abfragen kann und alle Abflughäfen bestimmen kann. Jeder Abflughafen wird dabei genau einmal ausgegeben.
- wie man die Datenbank abfragen kann, um alle Zielflughäfen zu bestimmen. Auch hier soll jeder Zielflughafen jeweils nur einmal ausgegeben werden.

- Falls der Abflughafen, der Zielflughafen und das Flugdatum bekannt ist, werden alle Flüge zwischen den zwei Orten an jenem Datum ausgegeben.

1.2.13.2. Das Flug-Interface

Schauen wir uns als erstes das Flug Interface an. Dieses Interface spezifiziert die Methoden, welche benötigt werden, um mit einer SQL Datenbank zu kommunizieren und die oben erwähnten Daten zu lesen:

- alle Abflughäfen, jeweils nur einmal (keine Mehrfachnennungen)
- alle Bestimmungs-Flughäfen (keine Mehrfachnennungen)
- bei gegebenem Abflughafen, Destination und Abflugdatun: alle Flüge.

```
package flugreservation;

/**
 * Title:
 * Description: Das Interface beschreibt die benötigten Methoden für unser
 *              Flugreservationsbeispiel
*/

public interface Fluege {

  /**
   * Die Methode bestimmeAlleAbflughaefen liefert alle Abflughäfen als Array
   */
  String [] bestimmeAlleAbflughaefen();

  /**
   * Die Methode bestimmeAlleZielflughaefen() liefert alle Zielflughäfen als Array
   */
  String [] bestimmeAlleZielflughaefen();

 /**
  * Die Methode bestimmeAlleFluege liefert alle Flüge
  * am Datum date
  * vom Flughafen origin
  * zum Flughafen dest
  */
  FlightInfo[] bestimmeAlleFluege(String origin, String dest, String date);

 /**
  * Die Methode produceFlightString generiert eine eindeutige Zeichenkette
  * für alle Flüge
  */
  String produziereFlugString (FlugInfo flugObjekt);
}
```

1.2.13.3. Die Verbindung zur Datenbank

Nun schauen wir uns an, wie der Verbindungsaufbau zur Datenbank aussieht.

Unser Szenario sieht folgendermassen aus: wir wollen
- den Kunden identifizieren.
- das Kundenprofil abspeichern und Informationen über den Kunden abfragen können.
- alle Flüge von...nach an einem bestimmten Tag bestimmen.
- die Sitze des Flugzeugs verwalten können.
- einen bestimmten Flug reservieren können.
- alle Informationen über einen bestimmten Flug abfragen können.

Schauen Sie sich folgendes Programmfragment an.
Frage: welche der folgenden Antworten ist korrekt?[1]
a) der Driver wird kreiert
b) die URL wird gebildet
c) eine Verbindung wird hergestellt
d) b) und c)

```
   ...
public FluegeImpl{String ServerName) {
    String url = " jdbc:odbc://"+ServerName + ":1112/FlugreservationsDB";

    try {
      mSQLcon = DriverManager.getConnection(url);
    } catch(java.sql.SQLException SQLEx) {
      System.out.println(SQLEx.getMessage() );
      SQLEx.printStackTrace();
      System.out.println("Die Datenbank antwortet nicht");
    }
  }

   ...
```

Frage: wie könnte eine SQL Abfrage aussehen, welche die Zielflughäfen bestimmt?

Lösungsskizze:

```
/**
   * Die Methode bestimmeAlleZielflughaefen() liefert alle Zielflughäfen
als Array
   */
  public String [] bestimmeAlleZielflughaefen(){
    String [] DestStaedte = null;
    Vector destStaedteV = new Vector(10, 10);
    Statement Destinationen = null;
    ResultSet rs = null;
    int j = 0;

    /* festlegen eines SQL Statements
       Abfragen mit DISTINCT Keyword garantiert, dass jedes Record,
       jeder Datensatz
       nur einmal angezeigt wird.
    */

    try {
      Destinationen = mSQLcon.createStatement();
      rs = Destinationen.executeQuery
            ("SELECT DISTINCT destStadt FROM Fluege");
    } catch (java.sql.SQLException SQLEx){
```

[1] d) ist korrekt

```
      System.out.println(SQLEx.getMessage());
    }

    try {
      DestStaedte = new String[rs.getMetaData().getColumnCount()];
      while (rs.next()){
        destStaedteV.addElement(rs.getString(1));
      }
      rs.close();       // Result Set schliessen
    } catch (java.sql.SQLException SQLEx){
      System.out.println("Spalte gibts nicht in der Datenbank-Tabelle");
    }

    DestStaedte = new String[destStaedteV.size()];
    Enumeration OE = destStaedteV.elements();
    while(OE.hasMoreElements()){
      DestStaedte[j] = (String)OE.nextElement();
      j++;
    }
    System.out.println(DestStaedte[0]+"   "+DestStaedte[1]);
    return DestStaedte;
  }
```

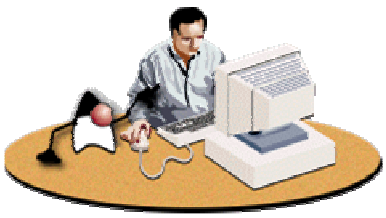
Frage: gibt es wesentliche Unterschiede zwischen den zwei Methoden (bestimmen aller Abflughäfen, bestimmen aller Zielflughäfen)?

```
/**
   * Die Methode bestimmeAlleAbflughaefen liefert alle Abflughäfen als
Array
   */
  public String [] bestimmeAlleAbflughaefen(){
    // Statement Objekt ist der Gateway zur Datenbank
    Statement Origins = null;
    Vector OrigStaedteV = new Vector(10, 10);
    String [] OrigStaedte = null;
    ResultSet rs = null;
    int j = 0;

    /* mit DISTINCT werden alle Datensätze nur einmal angezeigt
     */
    try {
      Origins = mSQLcon.createStatement();
      System.out.println (Origins);
      Rs = Origins.executeQuery("SELECT DISTINCT originCity FROM Fluege");
    } catch (java.sql.SQLException SQLEx){
      System.out.println(SQLEx.getMessage());
    }

    try {
      System.out.println(rs);
      while (rs.next()){
        OrigStaedteV.addElement(rs.getString(1));
      }
      rs.close();       // Resultset
    } catch (java.sql.SQLException SQLEx){
      System.out.println("Diese Spalte gibtes nicht");
    }

    OrigStaedte = new String[OrigStaedteV.size()];
    Enumeration OE = OrigStaedteV.elements();
    while(OE.hasMoreElements()){
      OrigStaedte[j] = (String)OE.nextElement();
      j++;
    }
    return OrigStaedte;
  }

  /**
```

```
   * Die Methode bestimmeAlleZielflughaefen() liefert alle Zielflughäfen
als Array
   */
  public String [] bestimmeAlleZielflughaefen(){
    String [] DestStaedte = null;
    Vector destStaedteV = new Vector(10, 10);
    Statement Destinationen = null;
    ResultSet rs = null;
    int j = 0;

    /* festlegen eines SQL Statements
       Abfragen mit DISTINCT Keyword garantiert, dass jedes Record / jeder
Datensatz
       nur einmal angezeigt wird.
    */

    try {
      Destinationen = mSQLcon.createStatement();
      rs = Destinationen.executeQuery("SELECT DISTINCT destStadt FROM
Fluege");
    } catch (java.sql.SQLException SQLEx){
      System.out.println(SQLEx.getMessage());
    }

    try {
      DestStaedte = new String[rs.getMetaData().getColumnCount()];
      while (rs.next()){
        destStaedteV.addElement(rs.getString(1));
      }
      rs.close();       // Result Set schliessen
    } catch (java.sql.SQLException SQLEx){
      System.out.println("Spalte mit diesem Namen gibts nicht in der
Datenbank-Tabelle");
    }

    DestStaedte = new String[destStaedteV.size()];
    Enumeration OE = destStaedteV.elements();
    while(OE.hasMoreElements()){
      DestStaedte[j] = (String)OE.nextElement();
      j++;
    }
    System.out.println(DestStaedte[0]+"   "+DestStaedte[1]);
    return DestStaedte;
  }
```

Lösung:
beide Methoden verhalten sich ähnlich!

Damit haben Sie, bis auf den FlugString, alle Methoden beieinander, um eine einfache Flugreservation zu implementieren.



Schauen Sie sich den Programmcode genau an.

Sie finden auf dem Server / der CD zusätzlich ein GUI als Prototyp, mit dem Sie eine Applikation vervollständigen können.

Ein Problem ist im obigen Prototypen vorhanden: Sie sollten sich entweder auf deutsche oder englische Datenfeldnamen festlegen. Oben finden Sie ein schlechtes Beispiel: die Namen sind gemischt und führen bei der Implementation und der Zusammenführung des GUI Prototypen und der Methoden zu Problemen.

1.2.14. Quiz

Zur Abrundung des Stoffes hier noch einige Fragen, die Sie ohne grössere Probleme beantworten können, falls Sie den Stoff zu diesem Thema eingehend durchgearbeitet haben.

Frage: welche der folgenden Aussagen trifft zu?

a) JDBC gestattet es einem Entwickler eine Applikation mittels Interfaces im JDBC API zu entwickeln, unabhängig davon, wie der Treiber implementiert wird. [2]
b) JDBC gestattet es einem Entwickler auf entfernte Daten genauso zuzugreifen, wie wenn alle Daten lokal wären.[3]
c) JDBC besteht aus einem Interface Set, welches es gestattet, die Applikation von der speziellen Datenbank zu trennen.[4]
d) a) und c) sind korrekt.[5]

Frage: welche der acht Interfaces aus JDBC (`Driver, Connection, Statement, PreparedStatement, CallableStatement, ResultSet, ResultSetMetatData und DatabaseMetaData`) müssen implementiert werden?

a) `Driver, Connection, Statement` und `ResultSet`[6]
b) `Driver, Connection, Statement, ResultSet, PrepareStatement, CallableStatement, ResultSetMetaData` und `DatabaseMetaData` [7]
c) `PreparedStatement, CallableStatement` und `ResultSetMetaData` [8]

Frage: nehmen Sie Stellung zu folgender Aussage!
Falls Sie mit einer Datenbank mittels JDBC Verbindung aufnehmen wollen, verwendet JDBC den ersten verfügbaren Treiber, um mit der URL eine Verbindung herzustellen.

Antwort:

a) trifft zu[9]
b) trifft nicht zu

[2] teilweise richtig
[3] nein: dies ist der Fall bei RMI, der Remote Methode Invocation
[4] diese Antwort ist teilweise richtig.
[5] genau
[6] denken Sie nochmals nach
[7] korrekt: alles muss implementiert werden, falls Sie diese benutzen wollen.
[8] nein
[9] korrekt: zuerst werden die Treiber aus der properties Liste verwendet, dann jene die bereits geladen sind.

Frage: welches der folgenden Objekte wird eingesetzt, falls Sie mehrfach dieselbe Abfrage machen wollen?[10]

```
a) PreparedStatement
b) Statement
c) Statement.executeQuery()
```

Frage: welche der folgenden Beschreibung gilt für ein dreistufiges, *three-tiers* Modell?

a) Sie verwenden Java, um direkt auf die Datenbankmethoden oder Bibliotheken zuzugreifen; Sie verwenden also datenbankspezifischen Code und Protokolle.
b) Sie verwenden Java und JDBC. Die JDBC Aufrufe werden anschliessend in DBMS Methodenaufrufe umgesetzt. Der JDBC ist somit DBMS unabhängig.

[10] a)

1.2.15. Zusammenfassung

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- zu erklären, was man unter JDBC versteht
- welches die fünf Aufgaben sind, mit der sich der JDBC Programmierer herumschlagen muss.
- zu erklären, wie der Treiber mit dem Treiber-Manager zusammenhängt.
- wie Datenbanktypen auf Java Datentypen abgebildet werden.
- welches die Unterschiede zwischen two-tiers und three-tiers Architekturen sind.

Sie sollten in der Lage sein, JDBC in der Praxis einzusetzen und damit DBMS Abfragen oder Mutationen durchzuführen.

1.3. Modul 2: Remote Method Invocation (RMI)

In diesem Modul

- Modul 2 : Remote Method Invocation RMI
  - Modul Einleitung
  - Was ist Java RMI?
  - RMI Architektur Übersicht
  - Der Transport Layer
  - Garbage Collection
  - Remote Reference Layer
  - RMI Stubs und Skeletons
  - RMI Packages und Hierarchien
  - Kreieren einer RMI Applikation
  - RMI Security
  - Remote Methode Invocation Praktische Übung
  - Quiz
  - Zusammenfassung

1.3.1. Einleitung

Das Remote Method Invocation (RMI) API gestatte es dem Java Entwickler Programme zu schreiben, welche auf remote Objekte zugreifen, genauso wie wenn diese Objekte lokal wären.

Analog zu den sogenannten Remote Procedure Calls (RPC) abstrahiert RMI die Socket Verbindung und die Datenumwandlungen, welche für eine Kommunikation mit einem entfernten Rechner benötigt werden. Dadurch werden entfernte Methodenaufrufe genau so gemacht, wie lokale Methodenaufrufe.

1.3.1.1. Lernziele

Nach dem Durcharbeiten dieser Unterlagen sollten Sie in der Lage sein:

- die RMI Architektur zu beschreiben, inklusive seinen verschiedenen Layern und dem (Distributed) Garbage Collector.
- RMI Server und Clients in Java zu implementieren.
- Client Stuby und Skeletons für remote Services mit Hilfe des Stub Compilers zu generieren.
- die Funktionsweise der RMI Registry zu beschreiben.
- eine RMI Applikation zu entwickeln, um eine bestimmte verteilte Aufgabe zu lösen.

1.3.1.2. Referenzen

Teile dieses Moduls stammen teilweise oder ganz aus

- "The Java Remote Method Invocation Specification"
  `http://java.sun.com/products/javaspaces/index.html`
- "Java RMI Tutorial"
  `http://java.sun.com/products/javaspaces/index.html`
- "Frequently Asked Questions, RMI and Object Serialization"
  `http://java.sun.com/products/javaspaces/index.html`

1.3.2. Was ist Java RMI?

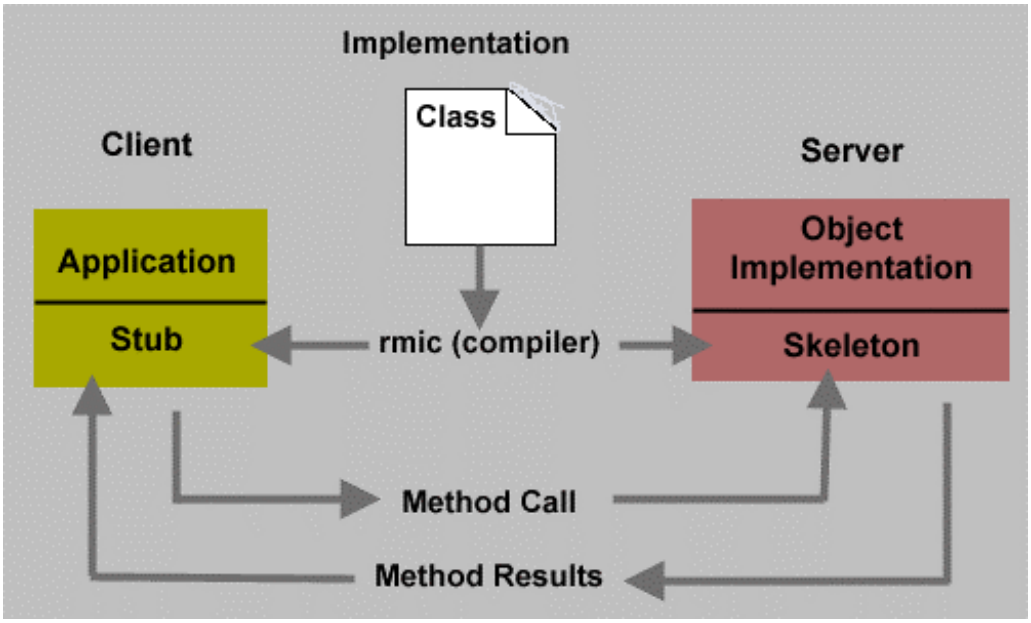
Das RMI API besteht aus einem Set von Klassen und Interfaces, mit deren Hilfe der Entwickler Methoden entfernter Objekte aufrufen kann, also Methoden von Objekten, die sich zur Laufzeit in einer anderen Java Virtuellen Maschine befinden. Diese "remote" oder "Server" JVM kann sich dabei auf derselben oder auf unterschiedlichen Maschinen befinden als der RMI Client. Java RMI ist im Gegensatz zu RPC eine reine Java Lösung.

In diesem Modul lernen Sie, wie man RMI Interfaces schreibt, die vom Clinet Code benutzt werden. und wie RMI Implementationsklassen aussehen, mit denen Objekte auf dem Server instanziert werden. Mit Hilfe der RMI Interface Beschreibungen werden Stubs (Client seitiger Code) und Skeletons (Server seitiger Code) mit Hilfe des `rmic`, des RMI Compilers generiert.

In verteilten Systemen sind ausgefeilte Ausfall- und Recovery- Mechanismen für verteilte Programme sehr wesentlich, einfach weil es viel mehr Ausfallmodi gibt. In diesem Modul lernen Sie einige dieser Java RMI definierten Exception Typen kennen, zusammen mit Hinweisen, wie Sie diese in Ihren Programmen einsetzen sollten.

1.3.3. Übersicht über die RMI Architektur

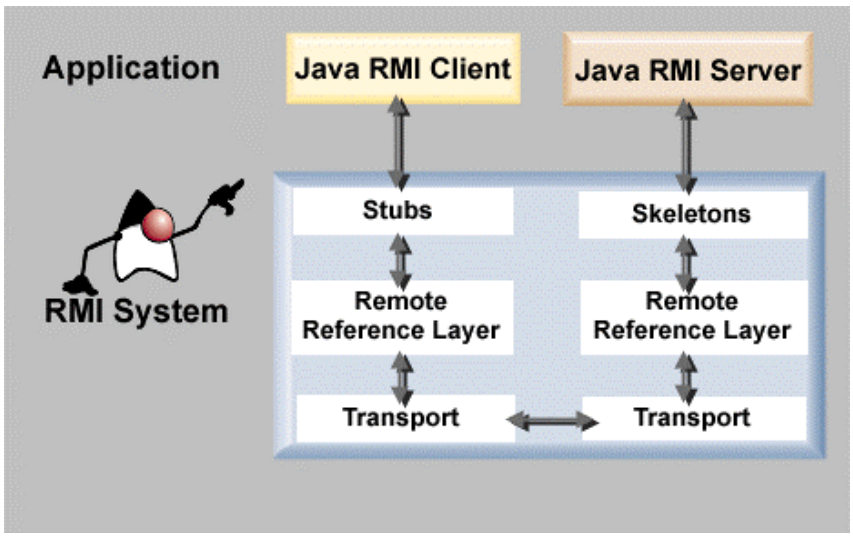
Der Aufruf einer remote Methode durch einen Client auf einem Server wird mit Hilfe verschiedener Layer des RMI Systems auf der Client Seite zum Transport Layer geleitet, dann zum Server transportiert und schliesslich mehrere Layer nach oben zum Server Objekt weitergeleitet.



Schematisch ist dies im obigen Bild (aus der RMI Dokumentation) wiedergegeben.
Das Ganze klingt trotzdem sehr komplex. Wie geschieht den dieser Transport und welche Layer gibt es?

Als Entwickler sind Sie lediglich für die Definition der Interfaces und der Implementationsklassen und das Generieren der Stub und Skeleton Klassen (mit Hilfe von `rmic`) verantwortlich.

Der Remore Reference Layer (RRL) und der Transport Layer kümmern sich um den Rest. Diese Layer könnten sich sogar von Release zu Release verändern, ohne dass Ihre Anwendungsprogramme davon betroffen wären.



Stubs und Skeleton, die Java Programme, die auf Grund unserer Interface Definitionen kreiert werden (wobei Skeletons bei neueren Releases nicht mehr benötigt werden), sind sozusagen die Schnittstelle zum Anwendungsprogramm und dem Reference Layer.

Der Reference Layer ist nötig, da Referenzen in der einen JVM auf Referenzen in einer eventuell entfernten JVM abgebildet werden müssen.

1.3.4. Der Transport Layer

Schauen wir uns als erstes die Funktionsweise des Transport Layers an.

Der Transport Layer ist für den Verbindungsaufbau, das Managementd der Verbindung und das Verfolgen und Verwalten der remote Objekte (den Zielobjekten der remote Calls) im Adressraum verantwortlich.

Der Transport Layer hat folgende Aufgaben:

- er empfängt eine Anfrage vom Remote Reference Layer auf der Clientseite.
- er lokalisiert den RMI Server, welcher das verlangte remote Objekt enthält.
- er baut eine Socket Verbindung zu diesem Server auf.
- er liefert diese Verbindung / Verbindungsinformation an den clientseitigen Remote Reference Layer.
- er trägt das remote Objekt in eine Tabelle ein, in der remote Objekte stehen, mit denen kommuniziert werden kann.
- er überwacht die "liveness" dieser Verbindung. Der Client selbst kann eine Verbindung zu einem RMI Server nicht selbst abbrechen. Dies ist eine der Aufgaben des Transport Layers.

1.3.4.1. Socket Verbindungen

Zur Zeit unterstützt Java RMI lediglich TCP Sockets. RMI benutzt typischerweise zwei Socket Verbindung: eine für die Methodenaufrufe und eine für den Distributed Garbage Collection (DGC). RMI wird versuchen, bestehende Socket Verbindungen mehrfach zu nutzen, für mehrere Objekte vom selben Server. Falls jedoch eine Socket Verbindung besetzt ist, wird automatisch eine neue Socket Verbindung aufgebaut.

Ein remote Objekt besteht aus einem Server Endpunkt und einem Objektidentifier. Dies wird als *live reference* bezeichnet.

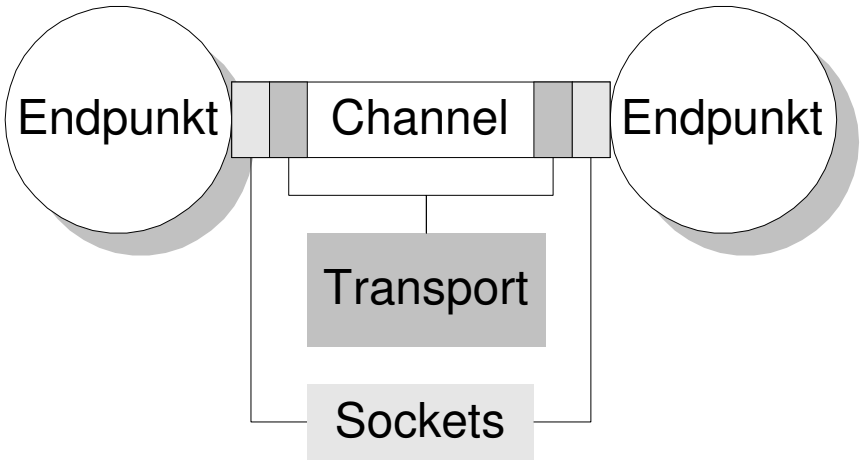
Bei gegebener live reference kann der Transport die Endpunkt Information benutzen, um eine Verbindung zum Adressraum aufzubauen, in dem das remote Objekt vorhanden ist.
Auf der Serverseite kann der Transport den Objektidentifier benutzen, um das Taregt des remote calls zu finden. Der Transport Layer für das RMI System besteht aus folgenden vier Abstraktionen:

- die **Verbindung** (*connection*)wird benutzt, um Daten zu transferieren (Eingabe/Ausgabe).
  Für jede Verbindung existiert ein Kommunikations- Kanel (*channel*), mindestens zwei Endpunkte, plus ein **Transport** (*transport*).

- ein **Endpunkt** (*endpoint*) beschreibt einen Adressraum oder Java Virtual Machine. Ein Endpunkt kann auf seinen Transport abgebildet werden. Bei gegebenem Endpunkt kann also eine spezifische Transportinstanz bestimmt werden.

- ein Channel wird als virtuelle Verbindung zwischen zwei Adressräumen eingesetzt. Der Kanal ist für das Management der Verbindungen zwischen dem lokalen Adressraum und dem remote Adressraum verantwortlich.

- ein **Transport** ist für das Management eines spezifischen Kanals zuständig.
  Der Kanal definiert, wie die konkrete Darstellung des Endpunktes aussieht. Pro Adressraumpaar oder Paar von Endpunkten besteht genau ein Transport. Bei gegebenem Endpunkt zu einem entfernten Adressraum baut ein Transport einen Kanal zwischen sich selbst und diesem Adressraum auf. Der Transport ist auch für die Annahme von ankommenden Anfragen an diesen Adressraum zuständig. Dabei wird ein Verbindungsobjekt für diese Anfrage kreiert und mit den höheren Layern kommuniziert.



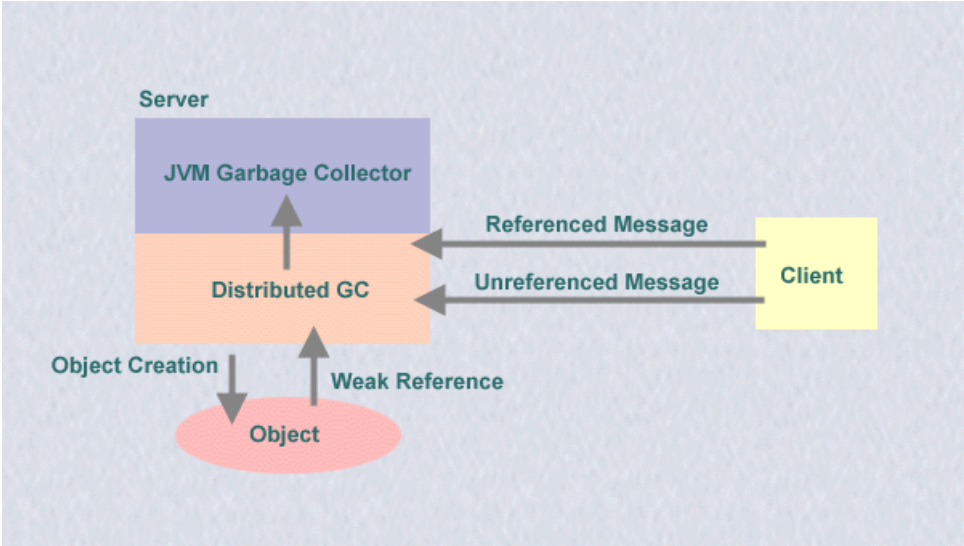
1.3.5. Garbage Collection

RMI benutzt ein Referenz Counting Garbage Collection. Alle live references innerhalb jeder JVM werden dabei berücksichtigt. Immer wenn eine live reference in eine JVM eintritt, wird deren Referenzzähler um eins erhöht. Falls ein Objekt keine Referenzen mehr besitzt, reduziert die Objekt Finalisation (*finalizer*) den Referenzzähler. Falls alle Referenzen aufgelöst wurden, wird dem Server eine "unreferenced" Meldung an den Server gesandt.

Falls ein remote Objekt von keinem Client mehr referenziert wird, wird dies als '*weak reference*' bezeichnet. Eine weak reference gestattet es dem Server Garbage Collector das Objekt zu löschen, sofern keine andere (lokale) Referenz auf dieses Objekt existiert. Solange lokale Referenzen existieren, kann das remote Objekt nicht als Abfall betrachtet werden. Es könnte nach als Parameter oder als Rückgabe eines remote calls eingesetzt werden.

1.3.5.1. Der Garbage Collection Prozess

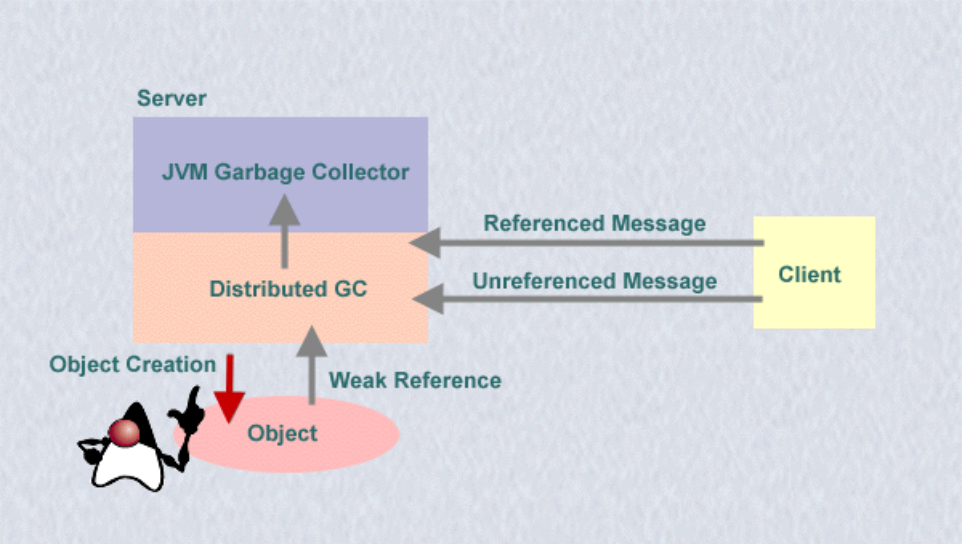
Die folgende Abbildung illustriert den fünfstufigen *Distributed Garbage Collecton* Prozess.



1) Kreieren und starten eines Objekts durch den Server oder die Implementation.
2) Die remote Referenz etabliert eine *'weak reference*' zum Objekt.
3) Falls der Client dieses Objekt verlangt, kreiert die Client JVM eine '*live reference*' und die erste Referenz auf dieses Objekt sendet eine '*Referenced*' Message zum Server.
4) Sobald das Objekt auf dem Client nicht mehr benötigt wird, sendet der Client eine '*Unreferenced*' Message an den Server
5) Falls der Referenzzähler auf dem Objekt auf Null zurück geht und es keine lokalen Referenzen gibt, kann die Objektreferenz dem lokalen GC (garbage collector) gemeldet werden.
   Der DGC kümmert sich darum; er ist Teil des RMI Laufzeitsystems, speziell des Transport Layers.

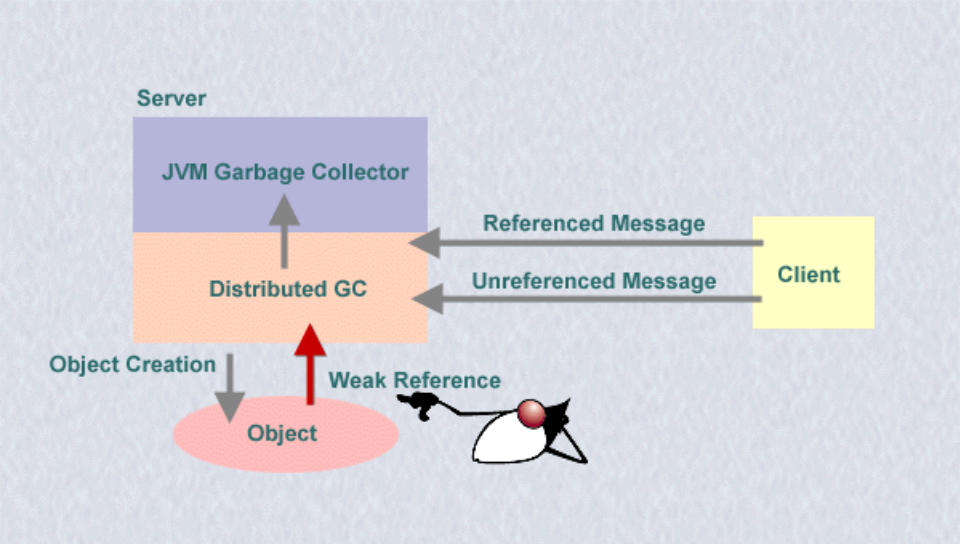
1.3.5.1.1. Distributed Garbage Collection - Schritt 1

Kreieren und starten eines Objekts durch den Server oder die Implementation.



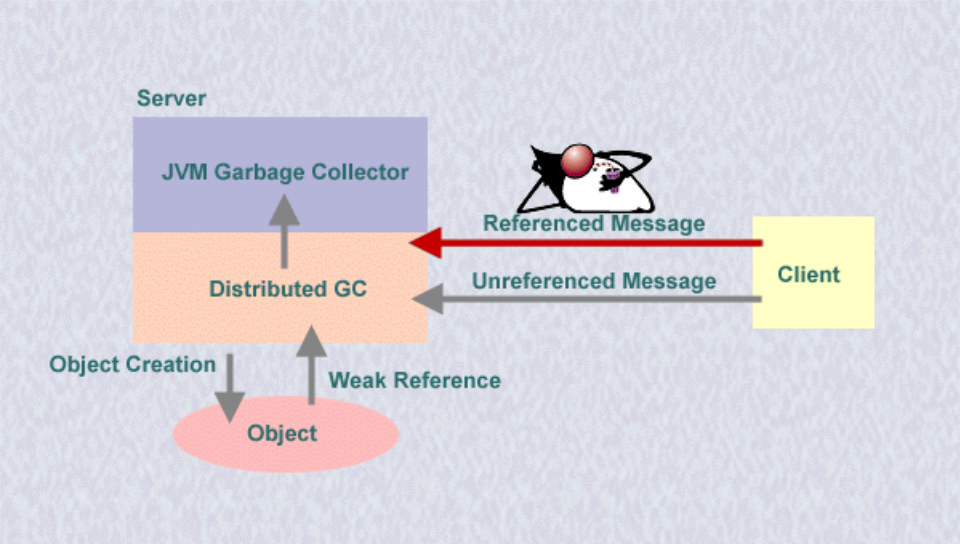
1.3.5.1.2. Distributed Garbage Collection - Schritt 2

Die remote Referenz etabliert eine *'weak reference'* zum Objekt.



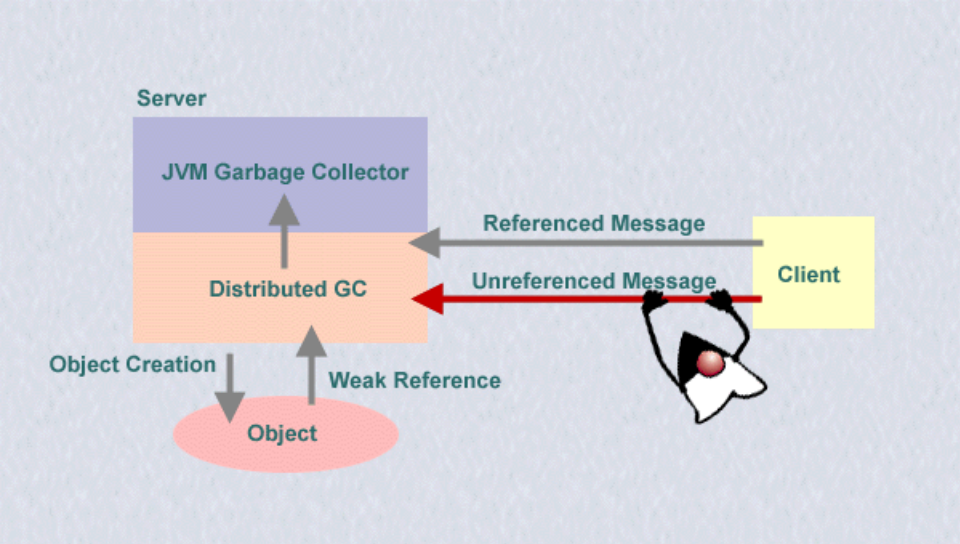
1.3.5.1.3. Distributed Garbage Collection - Schritt 3

Falls der Client dieses Objekt verlangt, kreiert die Client JVM eine '*live reference*' und die erste Referenz auf dieses Objekt sendet eine '*Referenced*' Message zum Server



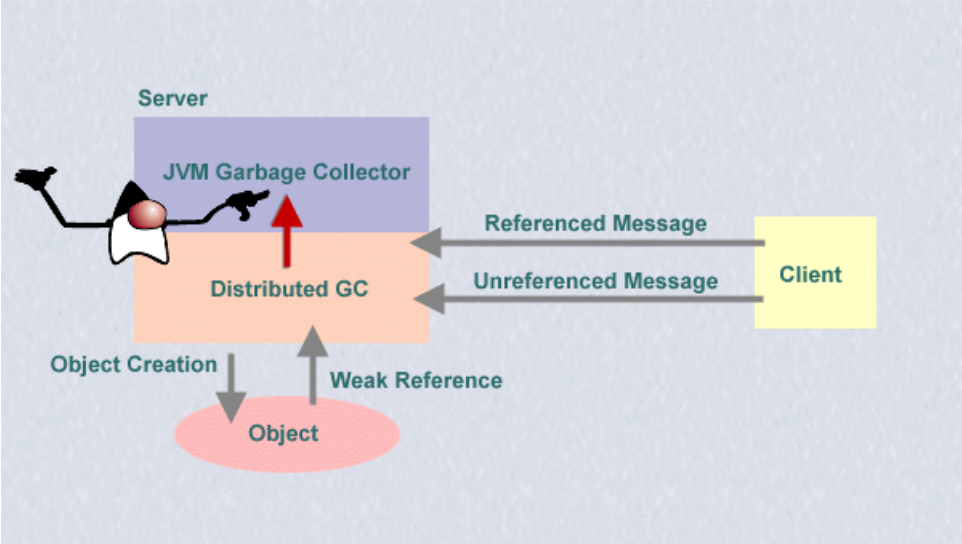
1.3.5.1.4. Distributed Garbage Collection - Schritt 4

Sobald das Objekt auf dem Client nicht mehr benötigt wird, sendet der Client eine '*Unreferenced*' Message an den Server



1.3.5.1.5. Distributed Garbage Collection - Schritt 4

Falls der Referenzzähler auf dem Objekt auf Null zurück geht und es keine lokalen Referenzen gibt, kann die Objektreferenz dem lokalen GC (garbage collector) gemeldet werden



1.3.6. Remote Reference Layer

Der Remote Reference Layer (RRL) ist für die korrekte Semantik der Methodenaufrufe zuständig. Dieser Layer kommuniziert zwischen den Stubs / Skeletons und dem tiefer liegenden Transport Interface. Dazu wird ein spezielles Remote Reference Protokoll verwendet, welches unabhängig ist von Client Stub und Server Skeletons. Zu den Aufgaben des RRL's gehört auch das Verwalten der Referenzen auf remote Objekte und Wiederverbindungsaufbau, falls ein Objekt nicht mehr verfügbar sein sollte.

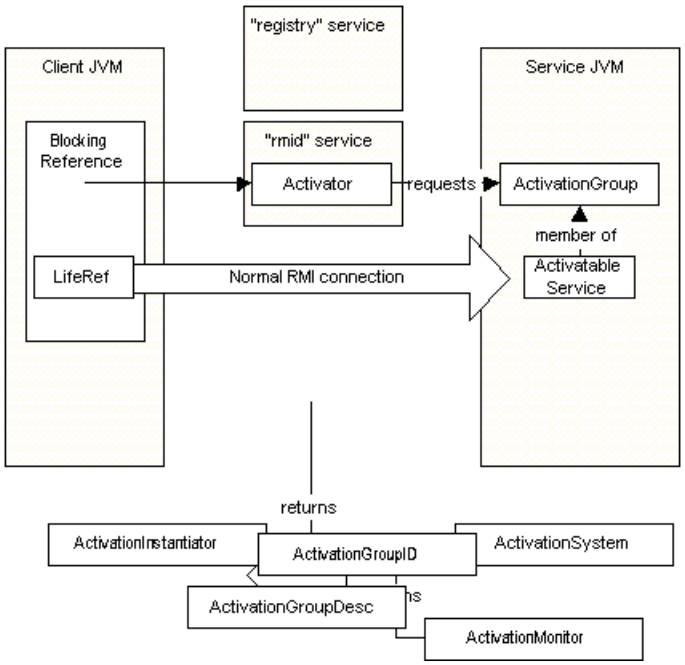
RRL besitzt zwei kooperierende Komponenten: die Client Seite und die Server Seite.

- Die clientseitige Komponente enthält spezifische Informationen über den remote Server und kommuniziert mit Hilfe des Transport Layers mit der serverseitigen Komponente.
- Die serverseitige Komponente implementiert die spezifische Referenzsemantik bevor ein remote Methodenaufruf an das Skeleton übergeben wird.

Die Referenzsemantik für den Server werden auch durch den RRL abgehandelt. RRL abstrahiert die unterschiedlichen Arten, auf die ein Objekt referenziert wird, welches

- auf Servern implementiert sind, welche dauernd auf einer Maschine laufen.
- auf Servern laufen, welche nur dann aktiviert werden, falls eine remote Methode auf ihnen aktiviert wird (mit dem **Activation Interface**).

Diese Unterschiede sind oberhalb des RRL nicht sichtbar.



1.3.7. RMI Stubs und Skeletons

Der Stub/Skeleton Layer ist das Interface zwischen den Applikationen, dem Applikationslayer, und dem Rest des RMI Systems. Dieser Layer kümmert sich also nicht um den Transport; er liefert lediglich Daten an den RRL.

Ein Client, welcher eine Methode auf einem remote Server aufruft, benutzt in Wirklichkeit einen Stub oder ein Proxy Objekt für das remote Objekt, also quasi ein Ersatz für das remote Objekt.

Ein Skeleton für ein remote Objekt ist die serverseitige Grösse, welche die Methodenaufrufe an die remote Objektimplementation weiterleitet.

Die Kommunikation der **Stubs** mit dem clientseitigen RRL geschieht auf folgende Art und Weise:

- Der Stub (clientseitig) empfängt den Aufruf einer entfernten Methode und initialisiert einen Call, einen Verbindungsaufbau zum entfernten Objekt.
- Der RRL liefert eine spezielle Art I/O Stream, einen *'marshal'* (Eingabe/ Ausgabe) Stream, mit dessen Hilfe die Kommunikation mit der Serverseite des RRL stattfindet.
- Der Stub führt den Aufruf der entfernten Methode durch und übergibt alle Argumente an diesen Stream.
- Der RRL liefert die Rückgabewerte der Methode an den Stub.
- Der Stub bestätigt dem RRL, dass der Methodenaufruf vollständig und abgeschlossen ist.

Skeletons kommunizieren mit dem serverseitigen RRL auf folgende Art und Weise:

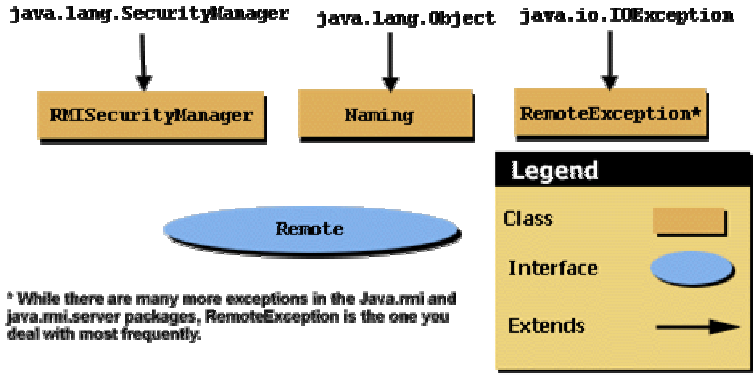
- Der Skeleton '*unmarshal*' (empfängt und interpretiert) alle Argumente aus dem I / O Stream, welcher durch den RRL aufgebaut wurde.
- Der Skeleton führt den Aufruf der aktuellen remote Objektimplementation durch.
- Der Skeleton '*marshals*' (interpretiert und sendet) die Rückgabewerte des Methodenaufrufes (oder einer Ausnahme, falls eine geworfen wurde) in den I / O Strom.

Bemerkung
Hüten Sie sich vor verteilten Deadlocks. Diese können beispielsweise auftreten, falls Sie versuchen Programmcodeböcke eines remote Objekts zu synchronisieren.
Falls Sie in einer verteilten Anwendung ein Lock, eine Sperre auf ein Objekt gesetzt haben, gibt es keinen Weg mehr zu unterscheiden, ob die Sperre von einem bestimmten Objekt oder irgendwelchen anderen Objekten stammt. Die Sperre kann somit im schlimmsten Fall auf ewig bestehen bleiben.

1.3.8. RMI Packages und Hierarchien

1.3.8.1. `java.rmi` Packages

Die grundlegenden Packages, Klassen und Interfaces, welche zur Entwicklung von RMI Clients und Servern eingesetzt werden, sind:



- `Naming`
  Diese Klasse ist `final` Klasse, mit deren Hilfe die RMI Clients und Server mit der Server Registry kommunizieren.
  Die Server Applikation benutzt die Methoden `bind` und `rebind`, um ihre Objektimplementationen bei der Registry zu registrieren (daher der Name dieses Lookup Servers 'Registry').
  Die Client Applikation benutzt die `lookup` Methode dieser Klasse, um eine Referenz auf das remote Objekt zu erhalten.

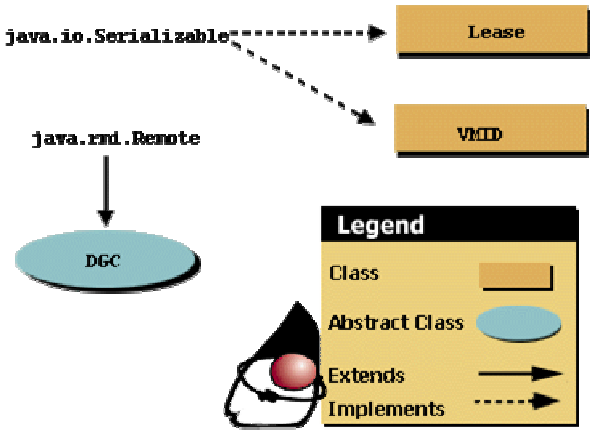
- `Remote` Interface
  Dieses Interface muss von allen Client Interfaces erweitert werden, welche auf das remote Objekt zugreifen möchten.

- `RemoteException`
  Diese Exception muss durch jede Methode geworfen werden, welche in einem remote Interface und Implementationsklassen definiert wird. Alle Clients müssen diese Exception abfangen.

- `RMISecurityManager`
  Diese kontrolliert den Zugriff auf lokale und remote Applikationen durch RMI Klassen und Interfaces.

1.3.8.2. Das `java.rmi.dgc` Package

Das `java.rmi.dgc` Package enthält Klassen, welche benötight werden, um remote Garbage Collection zu implementieren.

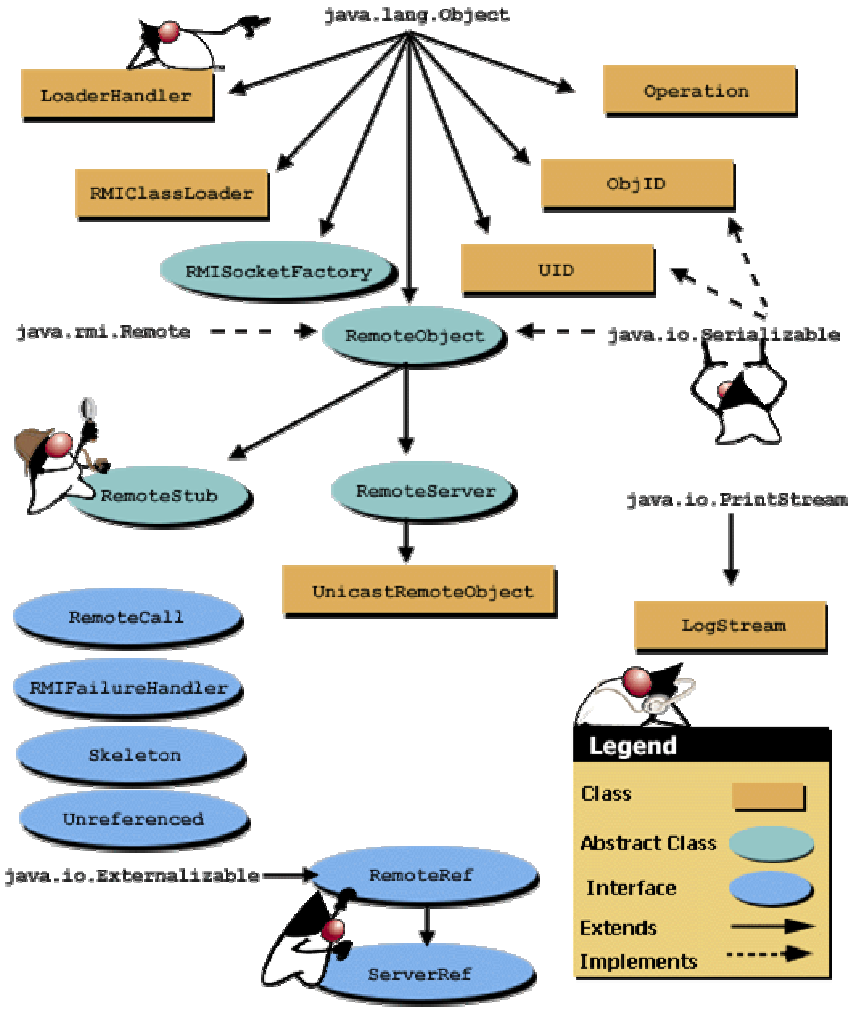


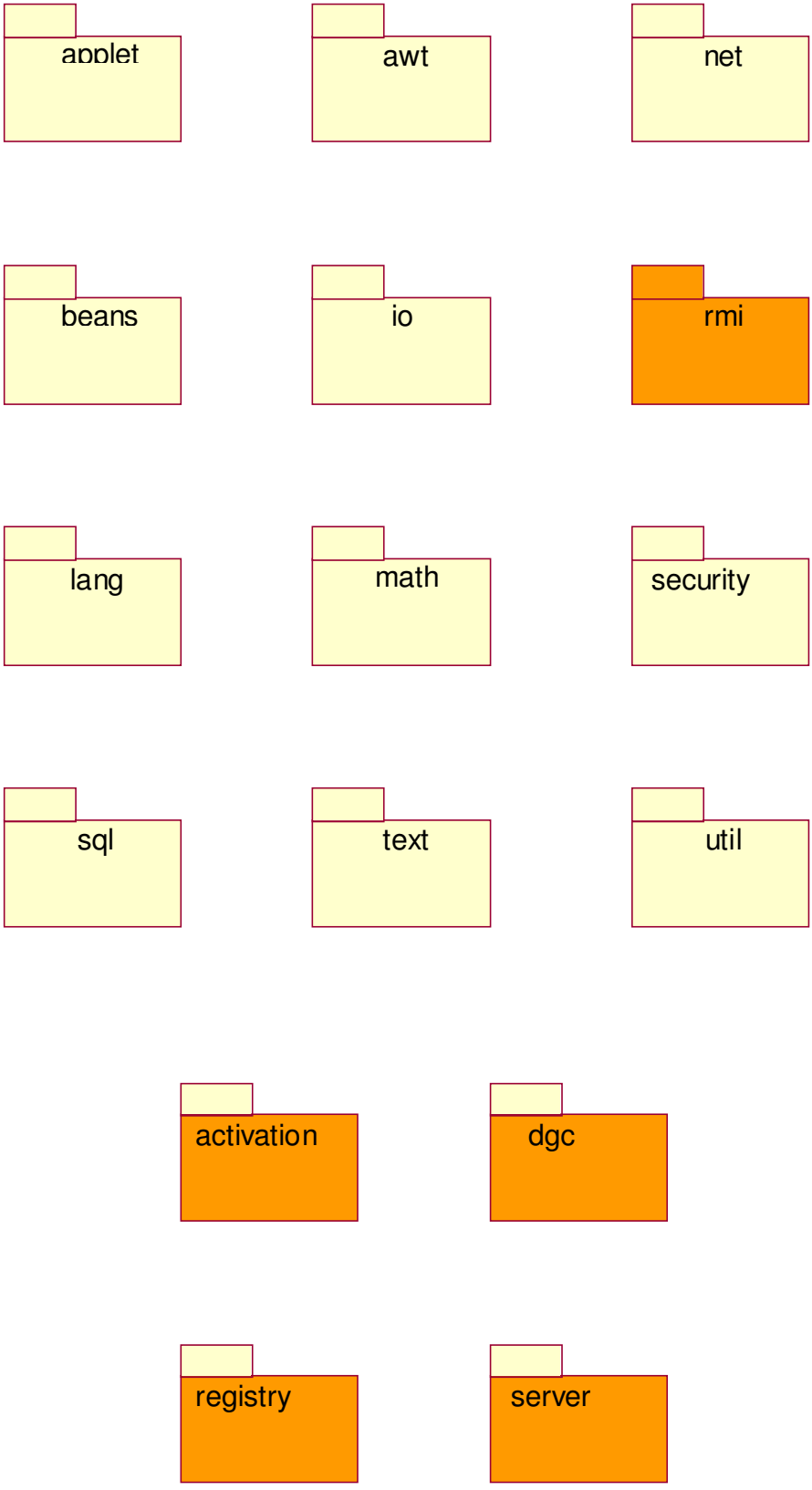
1.3.8.3. Das `java.rmi.server` Package

Aus der Vielzahl der Klassen betrachten wir zwei etwas genauer:

- `RMIClassLoader`
  Der RMI Class Loader ist der Class Loader, mit dem Stubs und Skeletons der remote Objekte (sowie Argumente und Rückgabewerte von remote Methoden) geladen werden. Falls der RMIClassLoader versucht Klassen vom Netzwerk zu laden wird eine Exception geworfen, falls kein Security Manager installiert wurde.

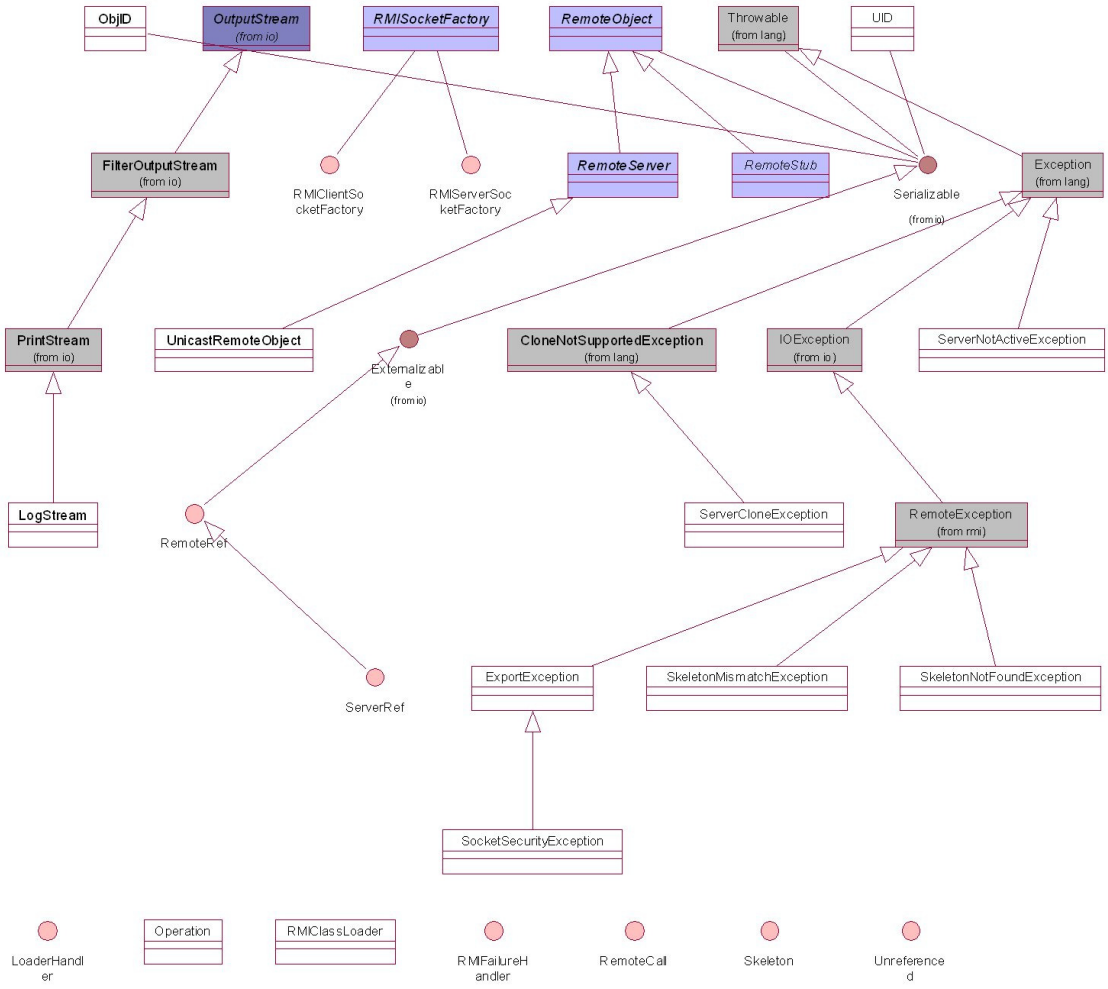
- `UnicastRemoteObject`
  Das Unicast Remote Objekt ist die Oberklasse für alle RMI Implementationsklassen.



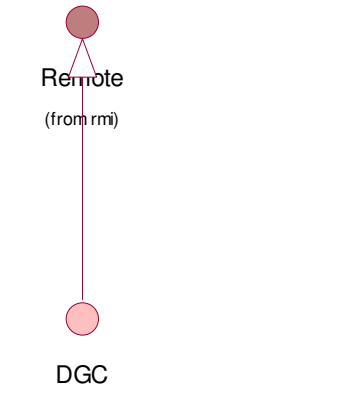


Sie finden grosse Bilder der Packages server und activation auf dem Server.

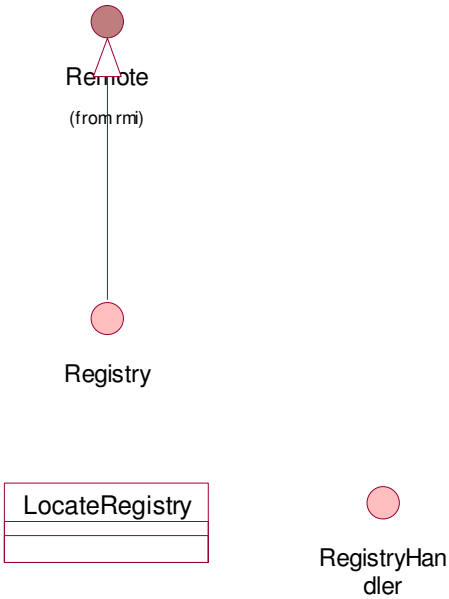
1.3.8.3.1. Das RMI Server Package



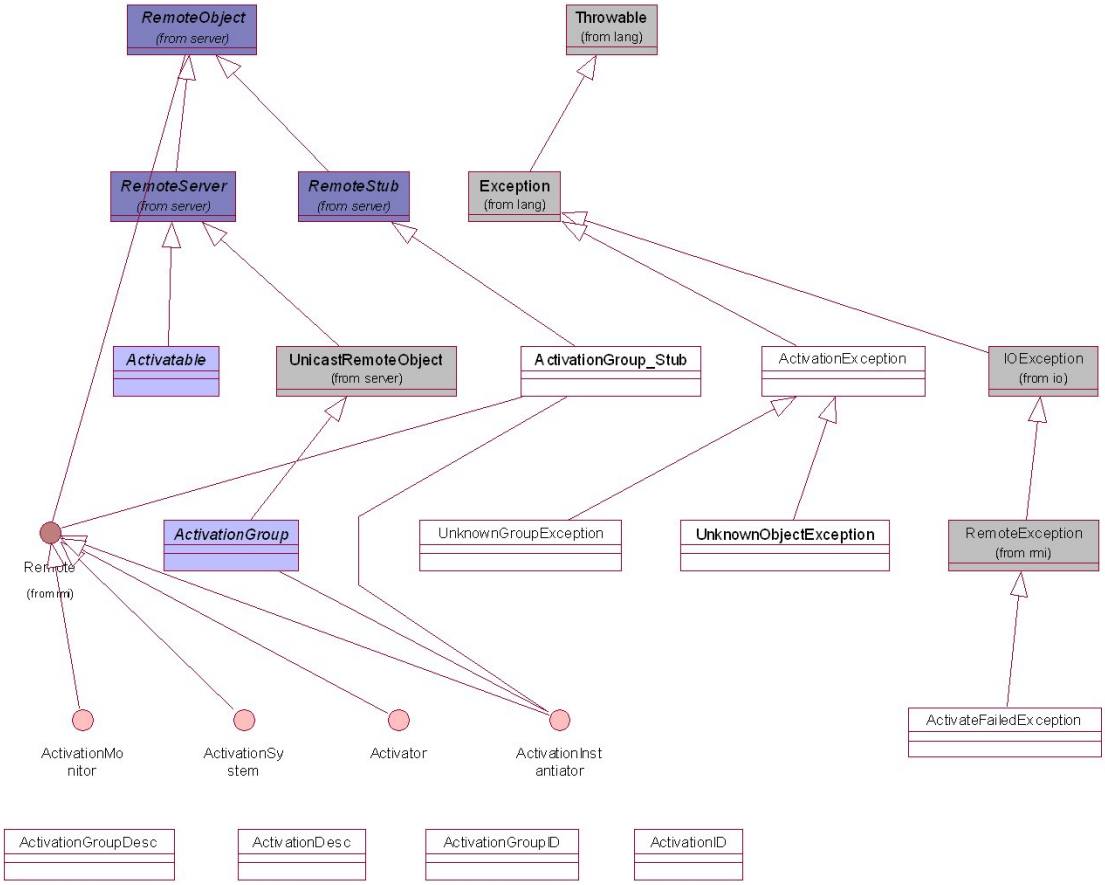
1.3.8.3.2. Das dgc Package



1.3.8.3.3. Das Registry Package



1.3.8.3.4. Das Activation Package



1.3.9. Kreieren einer RMI Applikation

Das folgende Beispiel verwendet ein Bankkonto und einen (Bank-)Kontomanager, um zu illustrieren, wie RMI eingesetzt werden kann. Beteiligt ist also eine Bank, bei der Sie ein Konto eröffnen können. Die Konten werden von einem Kontomanager, einem Angestellten der Bank, verwaltet.

Nachdem Sie ein Konto eröffnet haben, können Sie Geld einzahlen, Geld abheben und den Kontostand abfragen.

Falls Sie dieses Problem mit RMI lösen möchten, könnten Sie zwei Interfaces definieren:

- Bankkonto.java

```
package rmi.bank;
interface Bankkonto extends Remote {
      public float abfragenKontostand();
      public void abheben(float betrag);
      public void einzahlen(float betrag);
}
```

- Kontomanager.java

```
package rmi.bank;
interface Kontomanager extends Remote {
      public Konto eröffnen(String name, float startKapital);
}
```

Hinweis
Diese Interfaces sind so definiert, dass Sie ein Konto nur mit Hilfe des Kontomanagers eröffnen können. Sie erhalten vom Kontomanager (Objekt) ein Konto (Objekt). Der Kontomanger liefert Ihnen eine aktuelle Instanz eines Kontos, falls dieses bereits existiert.

Diese Art, Objekte zu kreieren, ist ein Beispiel für den Einsatz eines bestimmten *'Design Pattern*' (Entwurfsmuster) in der objektorientierten Programmiermethodologie. Dieses Muster wird als **Factory Methode** bezeichnet.

Die Factory Methode gestattet es einem Objekt andere Objekte zu kreieren. Das ist in unserem Fall genau das, was wir benötigen.

Der Kontomanager kontrolliert das Anlegen eines neuen Kontos. Falls Sie in eine Bank gehen möchten und ein neues Konto eröffnen möchten, sollten Sie eine Hinweis darauf erhalten, ob Sie bereits (und allenfalls welche) ein Konto bei dieser Bank besitzen.

1.3.9.1. Ablauf zum Kreiern einer RMI Applikation

Um eine remote verfügbare Applikation in RMI zu kreieren, gehen Sie folgenermassen vor:

1. Definieren Sie die remote Objekte, mit denen gearbeitet werden soll, als Java Interfaces.
2. Kreieren Sie Implementationsklassen für die Interfaces.
3. Übersetzen Sie Interfaces und Implementationsklassen.
4. Kreieren Sie Stub und Skeleton Klassen mit Hilfe des `rmic` Befehls, angewandt auf die Implementationsklassen.
   Achtung:
   *bei CORBA oder einigen anderen Techniken müssen Sie **zuerst** die Interfacebeschreibung übersetzen und **dann** die Implementationsklassen kreieren (und den generierten Programmcode einbauen).*
5. Kreieren Sie eine Serverapplikation, welche die remote Objekte administriert, und übersetzen Sie die Serverapplikation
6. Starten Sie die `RMIregistry` (oder bauen Sie diese gleich selbst als Teil der Serverapplikation: in diesem Fall entfällt dieser Schritt).
7. Testen Sie den Client.

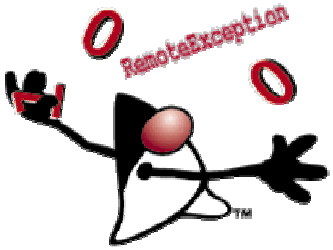
Das vollständige Beispiel für unser Problem sieht folgendermassen aus:

1.3.9.1.1. Das Konto Interface

```
package Bankkonto;

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

/**
 * Definition der Methoden für unser Bankkonto
 * abfragen des Kontostandes
 * einzahlen eines bestimmten Betrages
 * abheben eines bestimmten Betrages
 */

public interface Konto
    extends Remote
{
    /**
     * ebfragen des Kontostandes : liefert als float den Kontostand
     */
    public abstract float kontostand()
        throws RemoteException;

    /**
     * einzahlen eines (float) Betrages f auf das Konto
     */
    public abstract void einzahlen(float f)
        throws ungültigerBetragException, RemoteException;

    /**
     * abheben eines (float) Betrages f vom Konto
     */
    public abstract void abheben(float f)
        throws ungültigerBetragException, RemoteException;
}
```

Unser `Konto` Interface muss `java.rmi.Remote` erweitern und als `public` deklariert sein, um remote (für Clients in anderen virtuellen Maschinen) verfügbar zu sein.

Beachten Sie
Alle Methoden werfen die `java.rmi.RemoteException`. Diese Ausnahme wird immer dann geworfen, falls ein Problem beim Aufruf einer Methode eines remote Objekts auftritt.
Die Methoden `einzahlen(...)` und `abheben(...)` werfen zudem eine `ungültigerBetragException`, falls ein negativer Betrag einbezahlt oder mehr Geld abgehoben werden soll, als auf dem Konto ist.



Jede Methode in einem Remote Interface muss mindestens eine `RemoteException` (nicht eine Unterklasse von `RemoteException`) werfen. Dies wird von `rmic` überprüft.

1.3.9.1.2. Das KontoManager Interface

Das vollständige `KontoManager` Interface sieht folgendermassen aus:

```
package Bankkonto;

/**
 * Der Kontomanager kreiert (im Auftrag, als Objekt Factory) ein
 * neues Konto für einen Kunden
 * Parameter: Kundenname und Startbetrag.
 */

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

public interface KontoManager
    extends Remote
{

    /**
     * eröffnen eines Kontos für den Kunden s mit Anfangskontostand f
     */
    public abstract Konto eröffnen(String s, float f)
        throws ungültigerBetragException, RemoteException;
}
```

Argumente oder Rückgabewerte einer remote Methode können von irgend einem Java Datentyp sein, inklusive Objekte, solange diese Objekte entweder serialisierbar oder externalizable (im Falle einer `Externalizable` Klasse delegiert die Objekt Serialisierung die Kontrolle über das externe Format und die Art und Weise, wie die Supertypen gespeichert und wiederhergestellt werden, vollständig an das Objekt).

Auch dieses Interface erweitert `java.rmi.Remote` und die Methode `eröffnen(...)` kann die Ausnahme `RemoteException` (plus `ungültigerBetragException`, falls der Anfangsbetrag negativ ist) werfen.

Das `Konto` Interface wird von einer Klasse implementiert, welche alle deren Methoden implementiert und `UnicastRemoteObject` erweitert.

Hier hat sich eine Namenskonvention eingebürgert: alle Klassen, welche Interfaces implementieren, übernehmen den Namen des Interfaces plus dem Zustz "Impl".

```
// KontoImpl - Implementation des Konto Interfaces
//
// Instanzen dieser Klasse implementieren die Methoden:
// kontostand, abheben, einzahlen
// welche im Interface Konto definiert wurden
package Bankkonto;

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class KontoImpl extends UnicastRemoteObject implements Konto {
        // aktueller Kontostand
        private float kontostand = 0;
        // neues Konto eröffnen : Konstruktor
        public KontoImpl(float neuerKontostand) throws RemoteException {
                System.out.println("KontoImpl: Konstruktor");
                kontostand = neuerKontostand;
        }
        // Methoden
        // Kontostand abfragen
        public float kontostand() throws RemoteException {
                System.out.println("KontoImpl.kontostand(): Kontostand abfragen;
                                    Kontostand ="+kontostand);
                return kontostand;
        }
        // Betrag einbezahlen, sofern Betrag positiv
        public void einzahlen(float betrag) throws ungültigerBetragException, RemoteException {
                System.out.println("KontoImpl.einzahlen(): Betrag einzahlen;");
                System.out.println("KontoImpl: Kontostand alt: "+kontostand);
                if (betrag<0) {
                        throw new ungültigerBetragException("Fehler :
                                            Versuch einen negativen Betrag einzubezahlen!");
                } else {
                        kontostand = kontostand + betrag;
                }
                System.out.println("KontoImpl.einzahlen(): Kontostand neu: "+kontostand);
        }
        // Betrag abheben, falls das Konto darüber verfügt
        public void abheben(float betrag) throws ungültigerBetragException, RemoteException {
                System.out.println("KontoImpl.abheben():
                           Betrag abheben; Betrag ="+betrag+" Kontostand alt: "+kontostand);
                System.out.println("KontoImpl.abheben(): Kontostand alt: "+kontostand);
                if (betrag < 0) {
                        throw new ungültigerBetragException("KontoImpl.abheben() Fehler :
                             Versuch einen negativen Betrag abzuheben!");
                } else {
                        if ((kontostand - betrag)<0) {
                                throw new ungültigerBetragException("KontoImpl.abheben()
                                                    Fehler : Versuch Konto zu überziehen!");
                        } else {
                                kontostand = kontostand - betrag;
                        }
                }
                System.out.println("KontoImpl.abheben(): Kontostand neu: "+kontostand);
        }
}
```

Der Kontomanager wird durch eine Klasse implementiert, welche in der Lage ist, neue Konten anzulegen und die Konten abzuspeichern bzw. zu verwalten.

Konkret verwendet der Kontomanager einen Vektor als Speicher. Die Objekte werden mit Hilfe einer Klasse `KontoInfo` gebildet. Diese Klasse ist eine reine Hilfsklasse! Sie wird auch benutzt, um schnell nach bestehenden Konton suchen zu können.

```
package Bankkonto;

/**
 * Der Kontomanager kreiert (im Auftrag, als Objekt Factory) ein
 * neues Konto für einen Kunden
 * Parameter: Kundenname und Startbetrag.
 */
import java.io.PrintStream;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

public class KontoManagerImpl extends UnicastRemoteObject
    implements KontoManager
{

    private static Vector konten = new Vector();

    public KontoManagerImpl()
        throws RemoteException
    {
      System.out.println("KontoManager: Konstruktor");
    }
    public Konto eröffnen(String name, float startKapital)
        throws RemoteException, ungültigerBetragException
    {
      System.out.println("KontoManager.eröffnen():
                          Startkapital= "+startKapital);
            KontoInfo a;
            for(int i = 0; i < konten.size(); i++) {
                  a = (KontoInfo)konten.elementAt(i);
                  if(a.name.equals(name))
                  return a.konto;
            }
            if(startKapital < (float)0)
                  throw new ungültigerBetragException("Fehler :
                              negativer Startbetrag!");
            a = new KontoInfo();
            try {
                  a.konto = new KontoImpl(startKapital);
            }
            catch(RemoteException e) {
                  System.err.println("Fehler :
                              neues Konto konnte nicht kreiert werden!");
                  System.err.println(" " + e.getMessage());
                  throw e;
            }
            a.name = name;
            konten.addElement(a);
            return a.konto;
    }
}
```

Die `KontoInfo` Klasse ist eine Art Container Klasse, welche vom KontoManager (der Implementationsklasse davon) eingesetzt wird.

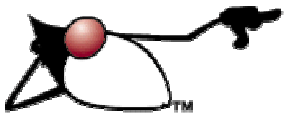
```
package Bankkonto;
// Container Klasse zum Verwalten der Konten
class KontoInfo {
      String name;
      KontoImpl konto = null;
}
```

Der Klassenpfad ist eine der Knackpunkte für verteilte Java Applikationen!
Sie sollten sich überlegen, die -d Option beim übersetzen der Java Programme einzusetzen, um die Klassendateien klar zu lokalisieren.

In unseren Beispielen werden deswegen in der Regel die BAT Dateien explizit angegeben und zusätzlich der CLASSPATH auf vernichtet (auf leer gesetzt):

```
javac –d . *.java
```

Mit diesem Flag werden auch die Package Verzeichnisse angelegt. In den BAT Dateien wird dies anders gelöst.



Nachdem Sie die Interfaces implementiert haben, müssen Sie Stubs und Skeletons kreieren. Mit diesen wird auf die jeweiligen remore Objekte zugegriffen. Dies geschieht mit Hilfe des `rmic`, des RMI Compilers. Dies geschieht nachdem die Implementationen geschrieben wurden, aber bevor die Server Applikation gestartet wird.

```
rmic  Optionen Package.InterfaceImpl ...
```

Zum Beispiel:

```
@echo off
cd ..
echo RMI Compile Bankkonto.KontoImpl
rmic –classpath . –keep Bankkonto.KontoImpl
echo RMI Compile Bankkonto.KontoManagerImpl
rmic –classpath . –keep Bankkonto.KontoManagerImpl
cd bankkonto
echo Kreieren eines Archives (Bankkonto.jar)
jar cvf Bankkonto.jar *_*.class
copy Bankkonto.jar ..\*.*
echo Das Archives (Bankkonto.jar) steht in den Verzeichnissen Banken und
Bankkonto
pause
```

Der RMI Compiler kreiert vier zusätzliche Dateien:

```
KontoImpl_Skel.java
KontoImpl_Stub.java
KontoManagerImpl_Skel.java
KontoManagerImpl_Stub.java
```

Bemerkung
Falls Sie eine Implementationsklasse ändern, müssen Sie diese neu übersetzen, der RMI Compiler muss Stubs und Skeletons neu generieren, die Registry muss gestoppt und neu gestartet werden und der Server ebenfalls.

Der Bankserver (`BankServer.java`) ist eine Applikation, welche die Kontomanager Implementationsklasse (`KontoManagerImpl.java`) jeweils bei Kundenanfragen instanziert und dem Client zur Verfügung stellt.

```
1. // Bank Server - Klasse, welche den RMI Server darstellt
2. //
3. package Bank;
4. import java.rmi.*;
5. import Bankkonto.*;

6. public class BankServer {
7. public static void main(String args[] ) {
8. System.out.println("BankServer :
                          start und setzen des SecurityManagers");
9. // kreieren und installieren eines SecurityManagers
10.   //System.setSecurityManager(new RMISecurityManager() );
11.   try {
12.   // Instanzen : Objekte,welche registriert werden
13.   System.out.println("BankServer.main() :
                                    kreieren eines KontoImpl Objekts");
14.   KontoManagerImpl kmi = new KontoManagerImpl();

15.   // binden der Instanz an die Registry
16.   System.out.println("BankServer.main() :
              binden des KontoManagers an den Bankserver [Registry]r");
17.   // zum Testen : Abfrage der Registry
18.   for (int i=0; i< Naming.list("rmi://localhost/").length; i++)
19.   System.out.println(" "+Naming.list("rmi://localhost/")[i]);

20.   Naming.bind("KontoManager",kmi);
21.   System.out.println("KontoManager Server ist gebunden!");
22.   } catch (Exception e) {
23.   System.err.println("BankServer.main() :
                    eine Ausnahme wurde geworfen - "+e.getMessage() );
24.   e.printStackTrace();
25.   }
26.   System.out.println("BankServer:main()... und Tschuess!");
27.   }
28.   }
```

Der Server publiziert' die Instanz der Kontomanager Implementationsklasse, indem er das Objekt mit einem Namen verbindet, der in einer Loopup Applikation abgespeichert wird, der rmiregistry. Sie könnten diese Registry auch direkt im Anwendungsprogramm kreieren:

```
LocateRegistry.createRegistry( PORT );
```

Das "Binden" geschieht auf Zeile 20 oben:
`Naming.bind("KontoManager",kmi);`

(`java.rmi.Naming.rebind(...)`).
Diese Methode ordnet dem Namen "KontoManager" das Objekt `kmi` zu, sie "bindet" die beiden zusammen. Dabei wird jede bereits vorhandene Bindung mit dem selben Namen in der Registry einfach überschrieben.

Die `Naming` Klasse verfügt über zwei Methoden, mit denen Objekte an Namen gebunden werden können: `bind` und `rebind`. Der Unterschied ist der, dass im Falle von `bind` die

Exception `java.rmi.AlreadyBoundException` geworfen wird, falls das Objekt bereits registriert ist.

Argument für bind und rebind sind URL ähnliche Zeichenketten und der Name der Instanz der Objektimplementation (siehe oben für ein Beispiel). Das Format des URL Strings ist :

```
rmi://<i>host:port/name
```

wobei:
`rmi` das Protokoll,
`host` der Name des RMI Servers (DNS Notation),
`port` die Portnummer ist, auf der der Server auf Anfragen wartet,
`name` der exakte Namen ist, den der Client im Aufruf `Naming.lookup` verwenden muss, falls er dieses Objekt benötigt.

Falls das Protokoll nicht angegeben wird, ist der Standardwert `rmi,` bei `host` ist der Standardwert `localhost` und bei Port `1099`.

Aus **Sicherheitsgründen** kann eine Applikation lediglich an eine Registry gebunden werden, welche lokal auf dem selben Rechner läuft.

Die RMIRegistry ist eine Applikation, welche einen einfachen Namens- Lookup Dienst zur Verfügung stellt. In unserem Beispiel stellt die Applikation der RMI Registry einen Namen und eine Objektreferenz zur Verfügung. Wie Sie oben gesehen haben, besteht das Programm aus wenigen Zeilen. Falls Sie die Registry in den Server integrieren, besteht die RMI Registry aus genau einer Zeile. Die RMI Registry liefert dem Garbage Collector auch Informationen über Referenzen. Das Programm `rmiregistry` muss laufen, bevor der Server gestartet wird und versucht seine remote Objekte zu binden:

```
rmiregistry
java –classpath . –Djava.security.policy=java.policy Bank.BankServer
```

Bemerkung
Auf der Win32 Plattform können Sie die Registry auch mit `start rmiregistry` starten. Schauen Sie sich in einem DOS Fenster die Syntax an. Damit haben Sie einige Optionen in Bezug auf Prioritäten, schliessen des DOS Fensters usw.

Auf der Kommandozeile (siehe oben) können Sie auch Properties setzen. Im obigen Beispiel wurde die Security Policy Datei auf diese Art und Weise angegeben. Weitere Optionen wären:

- Die Angabe der Codebase:
  `java.rmi.server.codebase`
  gibt die URL an, von der die benötigten Klassen heruntergeladen werden können, falls sie benötigt werden.
- Falls Sie eine Logdatei mitführen möchten, könenn Sie den Property Parameter `java.rmi.server.logCalls` auf `true` setzen. Standardmässig ist dieser Wert auf `false` gesetzt. Die Ausgabe erfolgt auf stderr.

Zwei Beispiele für das Einschalten des Loggings wären:

```
java -Djava.rmi.server.logCalls=true bank.BankServer

java -classpath . -Djava.security.policy=java.policy
                  -Djava.rmi.server.logCalls=true
                                                  Bank.BankServer
```

Nachdem Sie die Implementation angemeldet haben 'exportiert' die Registry dieses Objekt: sie bietet es Clients an. Der Client kann eine URL an die Registry senden. Die Registry liefert dann eine Referenz auf das remote Objekt. Der Lookup geschieht mittels eines Aufrufes:

```
Naming.lookup(...);
```

wobei als Argument eine Zeichenkette mitgegeben wird, den URL.

```
rmi:// host:port/name
```

Als nächstes schauen wir uns nun an, wie der Client diese Information verwertet!
Der Bankkunde (`Kunde.java`) versucht ein `Kontomanager` Objekt mittels eines Registry Lookups zu finden. Die Registry befindet sich auf dem Host und am Port gemäss URL Angabe, welche an `Naming.lookup(...)` übergeben wurde. Das Objekt, welches zurückgegeben wird, muss noch in ein `Kontomanager` Objekt gecastet werden. Es kann dann eingesetzt werden, um ein Konto auf einen Namen mit einem bestimmten Startbetrag zu eröffnen und Bankgeschäfte zu starten (Einzahlen, Abhabe, ...).

```
// BankClient - Test Programm für unser RMI Bankkonto Beispiel
//
// Diese Klasse sucht ein "KontoManager" RMI Objekt,
// bindet dieses und öffnet eine KontoManager Instanz
// auf dem Server
//
// Dann legt es ein Konto mit demNamen <name> und einem oder
// keinem Startkapital an
//
// Das Konto wird dann getestet, indem einfach Beträge einbezahlt
// und abgehoben werden, sowie der Kontostand abgefragt wird.
//
package Bankkunde;

import java.rmi.*;
// Interfaces importieren
import Bankkonto.*;
import Bankkonto.*;
```

```
public class Kunde {
      public static void main(String args[] ) {
            // Argumente prüfen
            if (args.length < 2) {
                  System.err.println("Usage : ");
                  System.err.println("java Bankkunde
                              <server> <Kontonamen> [Startkapital]");
                  System.exit(1);
            }

            // kreieren und installieren des SecurityManagers
            System.setSecurityManager(new RMISecurityManager() );

            // Kontomanager suchen
            try {
                  System.out.println("Kunde.main() ; lookup KontoManager");
                  String url = new
                                    String("rmi://"+args[0]+"/KontoManager");
                  KontoManager ktm = (KontoManager)Naming.lookup(url);
                  // Initialisieren des Bankkontos
                  float startBetrag = 0.0f;
                  // ... falls ein Betrag angegeben wurde
                  if (args.length ==3) {
                        Float F = Float.valueOf(args[2]);
                        startBetrag = F.floatValue();
                  }

                  // ... und nun das Konto anlegen oder nachsehen
                  Konto konto = ktm.eröffnen(args[1], startBetrag);

                  // Jetzt spielen wir mit dem Konto
                  System.out.println("Kunde.main() :
                        aktueller Kontostand = "+konto.kontostand() );

                  System.out.println("Kunde.main() : 50.00 abheben");
                  konto.abheben(50.00f);

                  System.out.println("Kunde.main() :
                        aktueller Kontostand = "+konto.kontostand() );

                  System.out.println("Kunde.main() : 100.00 einzahlen");
                  konto.einzahlen(100.00f);

                  System.out.println("Kunde.main() :
                        aktueller Kontostand = "+konto.kontostand() );

                  System.out.println("Kunde.main() : 25.00 einzahlen");
                  konto.einzahlen(25.00f);

                  System.out.println("Kunde.main() :
                        aktueller Kontostand = "+konto.kontostand() );
            } catch (Exception e) {
                  System.err.println("Kunde :
                        eine Ausnahme wurde geworfen; "+e.getMessage());
                  e.printStackTrace();
            }
            System.exit(1);
      }
}
```

Nachdem das Konto angelegt wurde, lassen wir den Client noch einige Operationen ausführen. Natürlich sollten diese Operationen mit einem GUI erledigt werden können. Aber dafür hatte ich bisher noch keine Zeit.

Den Bankclient können Sie auf irgendeinem Rechner starten, welcher mit dem Server über TCP/IP verbunden ist. Zusätzlich muss derZugriff auf die Klassendateien des Clients, dessen Stub und Interfaces garantiert sein (sonst können Sie den Client schlecht starten).

```
@echo off
Rem
Rem modifiziert
Rem
java -cp . -Djava.security.policy=java.policy Bankkunde.Kunde localhost
Joller 10000
Rem : jetzt folgt die Ausgabe
pause
```

Falls Sie die Registry und den Bankserver gestartet haben, liefert dieser Aufruf Ihnen folgende Ausgabe (vermutlich möchten Sie ein eigenes Konto):

```
Kunde.main() ; lookup KontoManager
Kunde.main() : aktueller Kontostand = 10000.0
Kunde.main() : 50.00 abheben
Kunde.main() : aktueller Kontostand = 9950.0
Kunde.main() : 100.00 einzahlen
Kunde.main() : aktueller Kontostand = 10050.0
Kunde.main() : 25.00 einzahlen
Kunde.main() : aktueller Kontostand = 10075.0
Taste drücken, um fortzusetzen . . .
```

Beim zweiten Aufruf des selben Clients sieht die Situation anders aus: jetzt sollte das Konto bereits angegelegt sein und auch ein Kontostand vorliegen, der sich vom Anfangskontostand (1000.00) unterscheidet. Der zweite Aufruf liefert folgende Ausgabe:

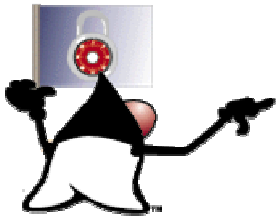
```
Kunde.main() ; lookup KontoManager
Kunde.main() : aktueller Kontostand = 10075.0
Kunde.main() : 50.00 abheben
Kunde.main() : aktueller Kontostand = 10025.0
Kunde.main() : 100.00 einzahlen
Kunde.main() : aktueller Kontostand = 10125.0
Kunde.main() : 25.00 einzahlen
Kunde.main() : aktueller Kontostand = 10150.0
Taste drücken, um fortzusetzen . . .
```

Das Witzige am Kundenprogramm ist, dass Sie dieses Verhalten eigentlich im Programmcode nicht sehen. RMI erledigt diese Arbeit für Sie!

Der Grund für dieses Verhalten ist folgender:
Der Server kreiert einen KontoManager. Dieser legt die Konten an. Der Server wartet auf Client Anfragen und leitet diese über RMI an die Instanz der KontoManagers weiter. Der KontoManager ist also auch nach dem Abarbeiten der Server Applikation noch aktiv in der Server JVM.

1.3.10. RMI Security

Um den RMI ClassLoader benutzen zu können, muss ein Security Manager installiert sein.



Dieser überprüft alle Klassen, welche über das Netzwerk geladen werden in Bezug auf Sicherheit und Sicherheitsverletzungen. Falls Sie keinen Security Manager starten / laden, kann Ihre Applikation keine Klassen über eine Netzwerkverbindung von einem anderen Host laden.

Beim Starten einer RMI Applikation können Sie zwei sicherheitsrelevante Parameter, Properties, setzen.

1) `java.rmi.server.codebase`, eine URL, welche angibt, von wo der Client die Klassen herunterladen muss. Natürlich muss der Server diese Adresse auch kennen und darauf Zugriff haben. Wir werden später im Praxisteil ein Beispiel kennen lernen, bei dem explizit mit Hilfe eines sehr einfachen HTTP Servers, bestehend aus im Wseentlichen zwei Java Klassen, die Dateien zum Client und vom Client auch zum Server verschiben werden.

2) Falls `java.rmi.server.useCodebaseOnly` auf `true` gesetzt wird, können Klassen nicht mehr über URLs, die der Client angibt, geladen werden (das Laden der Klassen ist dann nur noch über die Codebase möglich).

Falls der RMI Client ein Applet ist, verlangt der Browser bereits über seinen Security Manager und Class Loader die Sicherheit, die aus Sichtdes Browsers, zusammen mit den Policies auf Benutzerebene nötig sind.

Falls der Client eine Applikation ist, werden lediglich die remote Interface Definitionen, Stub Klassen und die Erweiterungen davon, herunter geladen. Falls ein Client zusätzliche Klassen vom Server laden möchte, dann kann er die Methode

```
RMIClassLoader.loadClass(...)
```

zusammen mit der selben URL, die bereits beim Aufruf von `Naming.lookup` verwendet wurde.

Der Transport Layer versucht normalerweise eine direkte Socket Verbindung vom Client zum Server aufzubauen. Falls dies nicht möglich ist, versucht die `createSocket` Methode der `java.rmi.server.RMISocketFactory` Klasse, eine Hypertext Transfer Protocol (HTTP) Verbindung aufzubauen, um RMI Calls als einen HTTP POST Request zu senden.

Falls die `createServerSocket` Methode eine neue Verbindung als eine HTTP Verbindung erkennt, wird die Antwort auf den POST Request wieder passend umgewandelt.

Auch für die Kommunikation durch eine Firewall werden keine speziellen Konfigurationen benötigt.

Bemerkung
Die Client Applikation kann verbieten, dass RMI Calls als HTTP Requests weitergeleitet werden. Dies geschieht durch das Setzen der

```
java.rmi.server.disableHTTP
```

Property als `true`.

1.3.11. Übung - Bauen einer Remote Method Invocation Applikation

In dieser einfachen Übung schauen Sie lediglich ein Programm an, welches eine Flugreservation durchführt. Die Applikation wurde mit RMI implementiert.

Der Kunde kann dabei die Angaben in ein GUI eingeben. Die Flugdaten werden dann zu einem remote Server gesandt, auf eine remote JVM.

Die Daten werden in einer Datenbank abgespeichert. Zudem stehen verschiedene Methoden zur Verfügung, mit deren Hilfe die Daten abgefragt, manipuliert und gelöscht werden können.

Der Kunde gibt zum Beispiel Daten zum Flug (Abflugshafen, Zielflughafen, Flug) ein. Alle persistenten Daten werden durch die Klasse `Datenbank.java` verwaltet.

Und hier eine mögliche Liste der Methoden für die Datenbank:

```
public KundenInfo [] liesKunde(String nName, String vName)
public void neuerKunde (String id, String nName, String vName, String addr)
public void bestehenderKunde(String id, String nName, String vName, String addr)
public void taetigeReservation(String id, String flug, String confirm, String sKlasse)
public String [] bestimmeAbflughafen()
public String [] bestimmeDestination() {
public boolean   legeSitzFest (KundenReservationsInfo res)
public FlugInfo [] bestimmeFlug(String origin, String dest, String datum)
```

Als Nächstes schauen wir uns die drei wichtigsten Interfaces an, die wir implementieren müssen:

- Kunden Interface
- KundenReservation Interface
- Flug Interface

Das Kunden Interface ist eines der wichtigsten Interfaces dieses Programms. Es erweitert `Remote` und alle seine Methoden können `RemoteException` werfen, was Bedingung für alle remote Server sind.

```
package flugreservation;

import java.rmi.*;
import java.rmi.server.*;

/**
 * Title:
 * Description:
 */

public class Kunde_Impl extends UnicastRemoteObject implements Kunde {

      private Datenbank db = null;

      // Konstruktor
      public Kunde_Impl(Datenbank db) throws RemoteException {
            this.db = db;
      }

      // Kundenrecord
      public KundenInfo liesKundenInfo(String kundenID) throws
                                    RemoteException {
            KundenInfo kunde = null;
            try {
                  kunde = db.liesKunde(kundenID);
            } catch(RecordNotFoundException e) {
                  System.err.println("liesKunde Exception : "+e);
            }
            return kunde;
      }

      // Liste der möglichen Kunden (Vorname, Name)
      public KundenInfo[ ] liesKundenInfo(String vorName, String nachName)
                                          throws RemoteException {
            KundenInfo[ ] kunde = null;
            try {
                  kunde = db.liesKunde(vorName, nachName);
            } catch(Exception e) {
            System.err.println("liesKundenInfo(array) Exception: "+e);
            }
            return kunde;
      }
      // Eintrag eines Kundenrecords in die Kundentabelle
      public void neueKundenInfo(String id, String nachName,
         String vorName, String strassenAdresse) throws RemoteException {
            try {
                  db.neuerKunde(id, nachName, vorName, strassenAdresse);
            } catch(InvalidTransactionException e) {
                  System.err.println("neuerKunde Exception : "+e);
            } catch(DuplicateIDException e) {
                  System.err.println("neuerKunde Exception : "+e);
            }
      }

      public void bestehendeKundenInfo(String id, String nachName,
         String vorName, String strassenAdresse) throws RemoteException {
            try {
                  db.bestehenderKunde(id, nachName, vorName,
                                                strassenAdresse);
            } catch(RecordNotFoundException e) {
                  System.err.println("bestehendeKundenInfo Exception :
                        "+e);
            }
```

```
	}

	// produziere eine neue Customer ID, mittels einer Formel,
	// die nur ein Programmierer kennen kann
	public String produziereID(String nachName, String vorName) throws
                                                        RemoteException {
		char EN = Character.toUpperCase(vorName.charAt(0) );
		char LN = Character.toUpperCase(nachName.charAt(0) );
		int ID = (int)(Math.random()*100000);
		String kundenID = new String(
			new
		   StringBuffer().append(EN).append(LN).append(Integer.toString(ID)
));
		return kundenID;
	}

	// KundenInfo Objekt als String
	public String produziereKundenString(KundenInfo info) throws
                                    RemoteException {
		String KundenString = info.kundenID + "-" + info.nachName + "-"
                            + info.vorName + "-" + info.strassenAdresse;
		return KundenString;
	}
}
```

Das dritte Interface beschreibt die Flüge. Auch dieses Interface erweitert `Remote` und wirft `RemoteExceptions`.

```
package flugreservation;

import java.rmi.*;
import java.rmi.server.*;
/**
 * Title:
 * Description:
 */

public class Flug_Impl extends UnicastRemoteObject implements Flug {

    private Datenbank db = null;

    // Konstruktion mit super(),
    public Flug_Impl (Datenbank db) throws RemoteException {
        this.db = db;
    }

    // Liste der Abflughäfen
    public String [] bestimmeAlleAbflughafen () {
      String [] originStaedte = db.bestimmeAbflughafen();
      return originStaedte;
    }

    // Liste der Destinationen
    public String [] bestimmeAlleDestinationen () {
      String [] destStaedte = db.bestimmeDestination();
      return destStaedte;
    }

    // FlugInfo Objekte
    public FlugInfo[] bestimmeVerfuegbareFluege (String origin,
                String dest, String datum) {
      try {
          FlugInfo[] verfuegbareFluege = db.bestimmeFluege(origin,
                dest, datum);
          return verfuegbareFluege;
      } catch (Exception e) {
          System.err.println("Fehler beim Bestimmen der verfügbaren
                              Flüge in KundenReservation_Impl " + e);
      }
      return null;
    }

    // Zeichenkette mit der Fluginfo
    public String produziereFlugString (FlugInfo flug) {
        String FlugString = flug.flugID + "-" + flug.originStadt +
          "-" + flug.destStadt + "-" + flug.abflugDatum + "-" +
            flug.abflugZeit + "-xxx-xxx-xxx-" +
          flug.flugzeugID;

      return FlugString;
    }
}
```

Auch das Kundenreservations Interface wird remote implementiert.

```
package flugreservation;

import java.rmi.*;
import java.rmi.server.*;
/**
 * Title:
```

```
 * Description:

 */

public class KundenReservation_Impl extends UnicastRemoteObject implements KundenReservation {

   private Datenbank db = null;

   // Aufruf von super(),
   // super exportiert das Objekt
   KundenReservation_Impl (Datenbank db) throws RemoteException {
         this.db = db;
   }

   // neue Reservation einfügen
   public void festlegenEinerReservation (String kundenID, String flugNummer, String
         bestaetigungsNummer, String serviceKlasse) throws RemoteException {

         try {
            db.taetigeReservation(kundenID, flugNummer, bestaetigungsNummer, serviceKlasse);
      } catch (InvalidTransactionException e) {
         System.err.println ("Exception in festlegenEinerReservation : " + e);
         }
   }

}
```

Wie geht's weiter?
Falls alle Interfaces definiert und implementiert sind:

1. übersetzen aller Klassen zum Testen und Debuggen der RMI Implementationen.
2. Kreieren von Stubs und Skeletons, welche zur Implementation der RMI Applikation benötigt werden.
3. Starten der RMI Registry.
4. Starten des JVM Servers.
5. Einsatz der Applikation mit einem GUI.

Dabei geht es um folgende Kommandos:

```
javac –d . *.java
```

Zusätzlich müssen wir die Stubs und Skeletons kreieren:

```
rmic –d . flugreservation.Kunde_Impl
rmic –d . flugreservation.Flug_Impl
rmic –d . flugreservation.KundenReservation_Impl
```

Falls Sie nun die Applikation testen möchten, müssen Sie als erstes die RMI Registry starten:

```
rmiregistry
```

dann den Server:

```
java –D java.rmi.server.codebase=/...
```

und schliesslich die Applikation:

```
appletviewer Airline.html
```

1.3.11.1. Aufgabe

Die folgende Aufgabe muss abgegeben werden.

Vervollständigen Sie die Flugreservationsskizze zu einer lauffähigen Applikation.
a) unter Zuhilfenahme von Sockets
b) als RMI Applikation

Beide Male sollten Sie ein GUI dazu entwerfen und natürlich möglichst viel Programmcode wiederverwenden.

Hinweis: diese Aufgabe entspricht in etwa einer Teilprüfung der (ganztägigen) Java Developer Certification.

1.3.12. Fragen - Quiz

Nach dem Durcharbeiten der Unterlagen sollten Sie in der Lage sein, die folgenden Fragen zu beantworten:

1. Welche der folgenden Aussagen über RMI ist korrekt?
   RMI gestattet es dem Entwickler, Programme zu schreiben, welche auf entfernte Objekte zugreifen und das genau so, als ob die Objekte lokal wären.

   Antwort:
   Diese Aussage ist teilweise korrekt!

2. RMI abstrahiert die Socket Verbindung und Datenströme für die Kommunikationzwischen Hosts.

   Antwort:
   Diese Aussage ist teilweise korrekt!

3. Beide der obigen Antworten sind korrekt!

   Antwort:
   Perfekt.

4. Als Entwickler einer RMI Applikation sind Sie für die Entwicklung einer Java RMI Interface Definition und deren Implementierungen zuständig, inklusive Stubs und Skeleton Generierung.

   a) trifft zu.
   b) Sie müssen sich auch noch um den Transport und die Verfolgung der remote Referenzen kümmern.

   Antwort:
   a) korrekt.
   b) Der RRL und der Transport Layer sind im RMI System bereits vorhanden und kümmern sich um den Transport und die Verfolgung der remote Objekte.

5. Welche der folgenden Aussagen über den Remote Reference Layer (RRL) trifft **nicht** zu?

   a) Der RRL managed die Kommunikation zwischen Stubs/ Skeletons und den unteren Transport Layern mit Hilfe eines spezifischen Remote Reference Protokolls, welches von Stubs und Skeletons unabhängig ist.

   b) Die clientseitige Komponente enthält Informationen über den Server und den Client und kommuniziert mit dem Transport Layer zur Serverseite.

   c) Der RRL definiert Endpunkte, mit deren Hilfe eine Verbindung in die Adressräume aufgebaut werden, in denen die remote Objekte vorhanden sind

   Antwort: c) ist die falsche Aussage : dies ist Aufgabe des Transport Layers.

d) Der RRL ist verantwortlich für das Management der remote Referenzen, also der Referenzen auf remote Objekte. Dazu gehören auch Strategien, mit deren Hilfe Verbindungsunterbrüche überbrückt werden können, falls das Objekt plötzlich nicht mehr erreichnar sein sollte.

6. Welche der folgenden Aussagen beschreibt nicht, wie Skeletons mit dem serverseitigen RRL kommunizieren?

   a) Das Stub/Skeleton führt einen remote Methodenaufruf aus und übergibt alle Argumente des Aufrufes an den Stream.

   Antwort: korrekt
   Der Stub führt den remote Methodenaufruf aus und übergibt alle Argumente an den Stream. Er arbeitet mit dem Skeleton zusammen..

   b) Das Skeleton unmarshals alle Argumente aus dem Stream I/O , welcher durch den RRL aufgebaut wurde.

   c) Das Skeleton führt den Up-Call zur aktuellen Server Implementation durch.

7. Wer ist dafür verantwortlich, dass die über das Netzwerk geladenen Klassen die Java Security Anforderungen erfüllen?

   a) Class Loader

   Antwort:
   Denken Sie nochmals nach! Der Class Loader lädt die Klassen.

   b) SecurityManager

   Antwort:
   Korrekt. Damit der RMI Class Loader überhaupt aktiv werden kann, muss der ein Security Manager installiert sein.

   c) RMI Class Loader

   Antwort:
   Siehe oben.

1.3.13. Zusammenfassung - Remote Methoden Invocation

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- die RMI Architektur beschreiben zu können, inklusive ihren Layern und dem Garbage Collection Prozess.
- einen RMI Server und Client in Java zu implementieren.
- Stubs und Skeletons mit Hilfe von RMI Compilern zu generieren
- zu definieren, welche Aufgabe die RMI Registry hat und wie sie grundsätzlich arbeitet.
- einige der Sicherheitsfragen im Zusammenhang mit RMI, dem Laden von Klassen über ein Netzwerk, zu beschreiben.
- eine RMI Aufgabe zu lösen.

1.4. Modul 3 : Objekt Serialisierung

In diesem Modul

- Modul 3 : Objektserialisierung
  - Modul Einleitung
  - Serialisierungs- Architektur
  - Lesen und Schreiben von Objektströmen
  - Serialisierungsbeispiel
  - Serialisierung versus Externialisierung
  - Objektserialisierung Praktische Übung
  - Quiz
  - Zusammenfassung

1.4.1. Einleitung

Das Remote Method Invocation (RMI) API gestatte es dem Java Entwickler Programme zu schreiben, welche auf remote Objekte zugreifen, genauso wie wenn diese Objekte lokal wären.

Analog zu den sogenannten Remote Procedure Calls (RPC) abstrahiert RMI die Socket Verbindung und die Datenumwandlungen, welche für eine Kommunikation mit einem entfernten Rechner benötigt werden. Dadurch werden entfernte Methodenaufrufe genau so gemacht, wie lokale Methodenaufrufe.

Das Objektserialisierungs-API gestattet es dem Entwickler, Java Code zu schreiben, welcher die persistente Speicherung der Java Objekte erlaubt.

Viele Applikationen verwenden Datenbanken für die Speicherung oder Persistenz der Daten. Allerdings werden Datenbanken typischerweise nicht eingesetzt, falls Sie Objekte, speziell Java Objekte abspeichern möchten. Der Zustand der Java Objekte muss in einer solcher Darstellung festgehalten werden, dass der Zustand leicht gespeichert und wieder leicht rekonstruiert werden kann.

Ziel des Java Objekt Serialisierungs- API ist es, Java einen einfachen Mechanismus für die persistente Speicherung der Objekte zur Verfügung zu stellen. Die Java RMI Architektur benutzt die Serialisierung, um die Objekte bei remote Methodenaufrufen zu übermitteln.

Die Objektserialisierung ist grundsätzlich ein einfacher Datenstrom, der den Zustand eines Objekts darstellt. Das API stellt Methoden zur Verfügung, mit deren Hilfe dieser Datenstrom produziert und konsumiert werden kann.

Objekte, welche als Container für Daten dienen, welche permanent gespeichert werden sollen, implementieren ein Interface, mit dessen Hilfe Objekte als Datenströme gespeichert und rekonstruiert werden können. Dies sind:

- `java.io.ObjectOutput` und
- `java.io.ObjectInput`

Um einen persistenten Speichermechanismus mit Hilfe des Objekt Serialisierungs-API aufzubauen, müssen Sie:

- ein flexibles Schema zur Benennung der Objekte, die Sie speichern wollen, definieren. Die Namensgebung sollte auf eine plausible Art und Weise die gespeicherten Objekte so benennen, dass Sie auch später noch erkennen können, welche Objekte Sie wo abgespeichert haben.
- eine Standardmethode definieren, mit der Sie die Werte jedes Feldes im Objekt auf kanonische Art und Weise in den Speicherstrom (und daraus heraus) bringen können.
- eine Methode kreieren, mit der Sie Namen und Art und Weise der persistenten Speicherung der Objekte kontrollieren können.
- Mechanismen definieren und kreieren, mit denen Sie bestimmen können, welche Datenfelder keine guten Kandidaten für die persistente Speicherung sind.

Das Remote Method Invocation (RMI) API verwendet beispielsweise die Objekt Serialisierung, um die Objekte auf die Leitung zu legen. Wann immer ein Objekt als Parameter in einem Methodenaufruf verwendet wird, wird das Objekt serialisiert bevor es mit dem Methodenaufruf an das remote Objekt gesandt wird. Dort wird das Objekt deserialisiert bevor das Objekt in der (remote) implementierten Methode als Parameter verwendet wird.

1.4.1.1. Lernziele

Nach dem Durcharbeiten dieser Unterlagen sollten Sie in der Lage sein:

- zur Erklären, wozu das Objekt Serialisierungs- API eingesetzt werden kann.
- die Beziehung zwischen dem Objekt Serialisierungs- API und dem RMI API zu erklären.
- persistente Speicherungen von Objekten mittels des Objekt Serialisierungs APIs zu realisieren.

1.4.1.2. Referenzen

Teile dieses Moduls stammen teilweise oder ganz aus

- "The Java Remote Method Invocation Specification"
  `http://java.sun.com/products/jdk/rmi/index.html`
- "Java RMI Tutorial"
  `http://java.sun.com/products/javaspaces/index.html`
- "Frequently Asked Questions, RMI and Object Serialization"
  `http://java.sun.com/products/javaspaces/index.html`
- "Java™ Object Serialisization Specification"

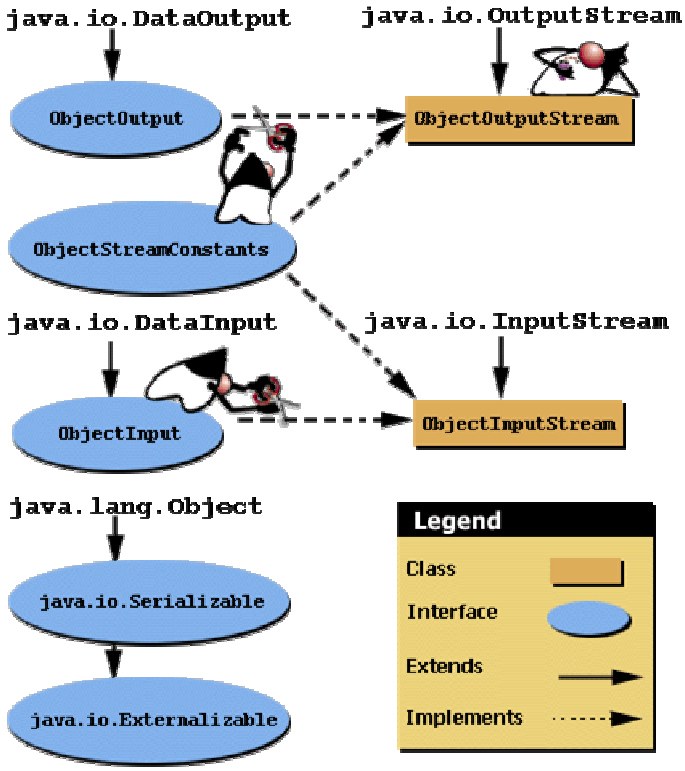
1.4.2. Serialisierungs- Architektur

1.4.2.1. Das `java.io` Package

Das Serialisierungs- API baut auf zwei Interfaces:

- `java.io.ObjectOutput` und
- `java.io.ObjectInput`

Diese Interfaces sind abstrakte Stream-basierte Interfaces, die definiert wurden, um Objekte in einen I/O Stream zu schreibe oder aus einem I/O Stream zu lesen.



1.4.2.2. Das `ObjectOutput` Interface

Das `ObjectOutput` Interface erweitert `DataOutput`, um `write` Methoden. Die wichtige Methode dieses Interfaces ist die `writeObject()`Methode, mit deren Hilfe Objekte geschrieben werden können. Exceptions können während dem Zugriff auf das Objekt, der Objektdatenfelder oder beim Schreiben des Speicherstroms passieren.

```
package java.io;
public interface ObjectOutput extends DataOutput {

      public void writeObject(Object obj)throws IOException;
      public void write(byte b[]) throws IOException;
      public void write(byte b[], int off, int len) throws IOException;
      public void flush() throws IOException;
      public void close() throws IOException;
}
```

1.4.2.3. Das `ObjectInput` Interface

Die `readObject` Methode wird benutzt, um Speicherströme zu lesen und Objekte wieder zurück zu gewinnen. Falls der Klassenname des serialisierten Objekts nicht gefunden werden kann, wird eine Ausnahme geworfen.

Der Klassenname ist Teil der Datei, in der das serialisierte Objekt abgespeichert wird. Dieser Name muss im `CLASSPATH` der Applikation, die das serialisierte Objekt liest, gefunden werden können. Sonst wird eine `ClassNotFoundException` geworfen.

```
package java.io;
public interface ObjectInput extends DataInput {

      public Object readObject()
            throws ClassNotFoundException, IOException;
      public int read() throws IOException;
      public int read(byte b[]) throws IOException;
      public int read(byte b[], int off, int len)
            throws IOException;
      public long skip(long n) throws IOException;
      public int available() throws IOException;
      public void close() throws IOException;
}
```

1.4.2.4. Das `Serializable` Interface

Mit Hilfe des `Serializable` Interfaces werden Klassen gekennzeichnet, als serialisierbar.

```
package java.io;
public interface Serializable {};
```

Im Prinzip können Sie jede Klasse serialisieren, sofern die Klasse folgende Kriterien erfüllt:

- die Klasse oder eine Klasse in der Klassenhierarchie dieser Klasse muss `java.io.Serializable` implementieren.
- Datenfelder, welche *nicht* serialisiert werden sollen, müssen als `transient` gekennzeichnet werden.
  Beispiele nicht serialisierbarer Klassen sind:
  `java.io.FileOutputStream` und `java.lang.Threads`.
  Falls Datenfelder nicht als `transient` gekennzeichnet sind aber nicht serialisierbar sind, wird eine `NotSerializableException` geworfen.

Alle (Daten-) Felder eines serialisierbaren Objekts werden in den Speicherstrom geschrieben. Dies umfasst insbesondere alle primitiven Datentypen, Arrays und Referenzen auf andere Objekte. Dabei werden lediglich die Daten (plus Klassennamen) des referenzierten Objekts gespeichert.

`static` Datenfelder werden nicht serialisiert!

Beachten Sie, dass die anderen Zugriffsmodifier, beispielsweise `private`, `protected` und `public` keinen Effekt auf die serialisieten Datenfelder haben. Als Entwickler sollten Sie sich überlegen, ob Ihre privaten Datenfelder nicht in Wirklichkeit transient und privat sind.

Da die Methoden nicht veränderlich sind, werden diese nicht serialisiert. Wir können ja später nach der De-Serialisierung wieder auf die Methoden zugreifen.

1.4.3. Schreiben und Lesen von Objektströmen

Das Lesen und Schreiben von Objektströmen ist denkbar einfach. Als Beispiel betrachten wir ein einfaches Code Fragment, welches eine Instanz der Datumklasse in eine Datei schreibt.

```
package ioobjektserialisierung;

import java.io.*;
import java.util.*;
public class SchreibenUndLesenEinesObjekts implements Serializable{
...

1  Date d = new Date();
2  FileOutputStream f = new FileOutputStream("date.ser");
3  ObjectOutputStream s = new ObjectOutputStream (f);
4  try {
5     s.writeObject (d);
6     s.close ();
7  } catch (IOException e) {
8     e.printStackTrace();
9  }
```

Das Lesen eines Objekts geschieht genau so einfach, allerdings mit einer Finesse: beim Lesen eines serialisierten Objekts wird einfach ein Objekt zurück gegeben. Dieses muss noch gecastet werden. Und dazu benötigen wir die Informationen über das Objekt, seine Klasse. Nach dem Casten stehen auch alle Methoden zur Verfügung.

```
1  Date d = null;
2  FileInputStream f = new FileInputStream ("date.ser");
3  ObjectInputStream s = new ObjectInputStream (f);
4  try {
5    d = (Date)s.readObject ();
6  } catch (IOException e) {
7    e.printStackTrace();
8  }
9  System.out.println ("Datum serialisiert um: "+ d);
```

1.4.4. Serialisierungsbeispiel

Schauen wir uns ein Beispiel für eine Objektserialisierung an. Als Beispiel wählen wir das Draw Applet, welches mit dem JDK als Demo mitgeliefert wird.

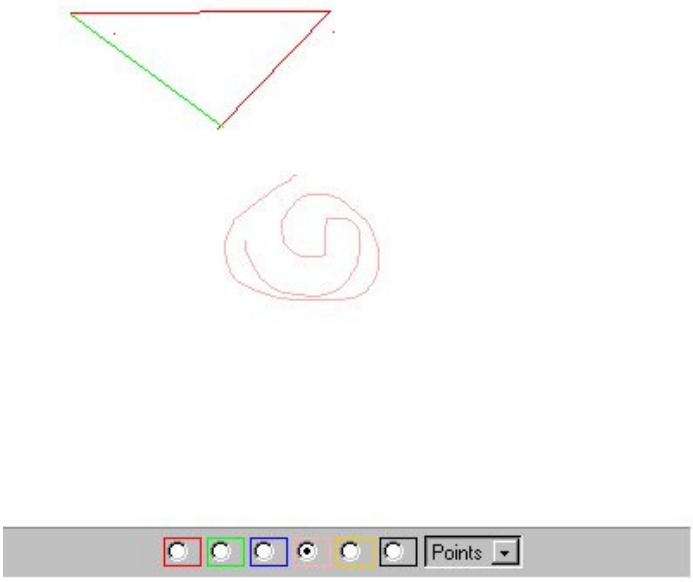
Hier ist der Quellcode:

```
/*
 * @(#)DrawTest.java    1.7 99/05/28
 *
*/

import java.awt.event.*;
import java.awt.*;
import java.applet.*;

import java.util.Vector;

public class DrawTest extends Applet{
    DrawPanel panel;
    DrawControls controls;

    public void init() {
      setLayout(new BorderLayout());
      panel = new DrawPanel();
        controls = new DrawControls(panel);
      add("Center", panel);
      add("South",controls);
    }

    public void destroy() {
        remove(panel);
        remove(controls);
    }

    public static void main(String args[]) {
      Frame f = new Frame("DrawTest");
      DrawTest drawTest = new DrawTest();
      drawTest.init();
      drawTest.start();

      f.add("Center", drawTest);
      f.setSize(300, 300);
      f.show();
    }
    public String getAppletInfo() {
        return "A simple drawing program.";
    }
}

class DrawPanel extends Panel implements MouseListener, MouseMotionListener
{
    public static final int LINES = 0;
    public static final int POINTS = 1;
    int        mode = LINES;
    Vector lines = new Vector();
    Vector colors = new Vector();
    int x1,y1;
    int x2,y2;
```

```
    public DrawPanel() {
      setBackground(Color.white);
      addMouseMotionListener(this);
      addMouseListener(this);
    }

    public void setDrawMode(int mode) {
      switch (mode) {
        case LINES:
        case POINTS:
          this.mode = mode;
          break;
        default:
          throw new IllegalArgumentException();
      }
    }


    public void mouseDragged(MouseEvent e) {
        e.consume();
        switch (mode) {
            case LINES:
                x2 = e.getX();
                y2 = e.getY();
                break;
            case POINTS:
            default:
                colors.addElement(getForeground());
                lines.addElement(new Rectangle(x1, y1, e.getX(),
e.getY()));
                x1 = e.getX();
                y1 = e.getY();
                break;
        }
        repaint();
    }

    public void mouseMoved(MouseEvent e) {
    }

    public void mousePressed(MouseEvent e) {
        e.consume();
        switch (mode) {
            case LINES:
                x1 = e.getX();
                y1 = e.getY();
                x2 = -1;
                break;
            case POINTS:
            default:
                colors.addElement(getForeground());
                lines.addElement(new Rectangle(e.getX(), e.getY(), -1, -
1));
                x1 = e.getX();
                y1 = e.getY();
                repaint();
                break;
        }
    }

    public void mouseReleased(MouseEvent e) {
        e.consume();
```

```
        switch (mode) {
            case LINES:
                colors.addElement(getForeground());
                lines.addElement(new Rectangle(x1, y1, e.getX(),
e.getY()));
                x2 = -1;
                break;
            case POINTS:
            default:
                break;
        }
        repaint();
    }

    public void mouseEntered(MouseEvent e) {
    }

    public void mouseExited(MouseEvent e) {
    }

    public void mouseClicked(MouseEvent e) {
    }

    public void paint(Graphics g) {
      int np = lines.size();

      /* draw the current lines */
      g.setColor(getForeground());
      for (int i=0; i < np; i++) {
          Rectangle p = (Rectangle)lines.elementAt(i);
          g.setColor((Color)colors.elementAt(i));
          if (p.width != -1) {
            g.drawLine(p.x, p.y, p.width, p.height);
          } else {
            g.drawLine(p.x, p.y, p.x, p.y);
          }
      }
      if (mode == LINES) {
          g.setColor(getForeground());
          if (x2 != -1) {
                g.drawLine(x1, y1, x2, y2);
          }
      }
    }
}


class DrawControls extends Panel implements ItemListener {
    DrawPanel target;

    public DrawControls(DrawPanel target) {
      this.target = target;
      setLayout(new FlowLayout());
      setBackground(Color.lightGray);
      target.setForeground(Color.red);
      CheckboxGroup group = new CheckboxGroup();
      Checkbox b;
      add(b = new Checkbox(null, group, false));
      b.addItemListener(this);
      b.setForeground(Color.red);
      add(b = new Checkbox(null, group, false));
      b.addItemListener(this);
```

```
      b.setForeground(Color.green);
      add(b = new Checkbox(null, group, false));
      b.addItemListener(this);
      b.setForeground(Color.blue);
      add(b = new Checkbox(null, group, false));
      b.addItemListener(this);
      b.setForeground(Color.pink);
      add(b = new Checkbox(null, group, false));
      b.addItemListener(this);
      b.setForeground(Color.orange);
      add(b = new Checkbox(null, group, true));
      b.addItemListener(this);
      b.setForeground(Color.black);
      target.setForeground(b.getForeground());
      Choice shapes = new Choice();
      shapes.addItemListener(this);
      shapes.addItem("Lines");
      shapes.addItem("Points");
      shapes.setBackground(Color.lightGray);
      add(shapes);
    }

    public void paint(Graphics g) {
      Rectangle r = getBounds();
      g.setColor(Color.lightGray);
      g.draw3DRect(0, 0, r.width, r.height, false);

        int n = getComponentCount();
        for(int i=0; i<n; i++) {
            Component comp = getComponent(i);
            if (comp instanceof Checkbox) {
                Point loc = comp.getLocation();
                Dimension d = comp.getSize();
                g.setColor(comp.getForeground());
                g.drawRect(loc.x-1, loc.y-1, d.width+1, d.height+1);
            }
        }
    }

  public void itemStateChanged(ItemEvent e) {
    if (e.getSource() instanceof Checkbox) {
      target.setForeground(((Component)e.getSource()).getForeground());
    } else if (e.getSource() instanceof Choice) {
      String choice = (String) e.getItem();
      if (choice.equals("Lines")) {
      target.setDrawMode(DrawPanel.LINES);
      } else if (choice.equals("Points")) {
      target.setDrawMode(DrawPanel.POINTS);
      }
    }
  }
}
```



In diesem Beispiel sehen Sie, wie das Serialisierungs-API eingesetzt werden kann, um den Zustand eines einfachen Zeichneprogramms abzuspeichern. Mit dem obigen Applet können Sie einfache Linien farbig zeichnen oder Punkte aneinanderreihen.

Dabei malen Sie auf ein Objekt, ein `DrawPanel`, welches `Panel` erweitert Diese Klasse ist serialisierbar, da `java.awt.Component` serialisierbar ist und in der Klassenhierarchie von `DrawPanel` ist.

Hinweis
Beachten Sie, dass `DrawPanel` keine Referenzen auf nicht serialisierbare Objekte, wie beispielsweise `java.io.FileOutputStream`, besitzt, da sonst die Objekte als `transient` gekennzeichnet werden müssten.

Nun wollen wir das Beispiel so erweitern, dass wir je einen Knopf zur Verfügung haben, um Objekte zu serialisieren oder ab Festplatte zu lesen und zu rekonstruieren. Dazu definieren wir eine `SerializationButton` Klasse, welche `java.awt.Button` erweitert. Diese Klasse ist ein Button Objekt mit zwei zusätzlichen Methoden: `saveObject` und `restoreObject`. Diese sind für das Schreiben und Lesen des `DrawPanel` Objects in / aus einem File zuständig.

Die Klasse `SerializationButton` ist ein Beispiel einer Klasse, welche das GUI eines Button mit der Funktionalität zweier Methoden verknüpft, welche für die Serialisierung und De-Serialisierung von Objekten eingesetzt werden können.

Die Methoden:

- `writeObject()`
- `readObject()`

in den Klassen `ObjectOutputStream` und `ObjectInputStream`, beschränken sich auf das Lesen und Schreiben von Objekten. Der Serialisierungsknopf überlässt es der aufrufenden Klasse, die Objekte zu casten.

```
package drawtestprojekt;

// SerializationButton - dieser Button dient der Serialisierung
// eines Objekts, welches mit saveObject gespeichert und mit
// restoreObject wieder geladen wird
//
// Diese Klasse kümmert sich nicht um die Details des Objekts.
// restoreObject liefert also ein Objekt, welches gecastet werden muss

import java.awt.*;
import java.io.*;

public class SerializationButton extends Button {
   String fileName = null;

   FileOutputStream os = null;
   FileInputStream f = null;

   // Serialization streams
   ObjectOutputStream s = null;
   ObjectInputStream is = null;

  // Kreiere einen Button mit Label
  // fileName ist ein String Dateiname, in den das Objekt gespeicht
  // aus dem das Objekt gelesen werden soll.

  public SerializationButton(String label, String fileName){
    super(label);
    this.fileName = fileName;
  }
  // Schreibe das Objekt in die Datei fileName
     public void saveObject(Object obj){

     // öffne den Output Stream, kreiere die DAtei
     // schreibe das serialisierte Objekt
     try {
        os = new FileOutputStream(fileName);
        System.out.println("FileOutputStream war OK");
        s = new ObjectOutputStream(os);

     } catch (Exception e) {
       System.out.println(e);
       e.printStackTrace();
       System.out.println("Der File Stream kann nicht geöffnet werden: " +
          fileName + " als Objektstrom");
     }

     // schreibe das Objekt in den Objekt Ausgabestrom
     try {
        s.writeObject(obj);
        System.out.println("Das Objekt wurde gespeichert.");
        os.close();
     } catch ( Exception e ) {
        System.out.println(e);
        System.out.println("Das Objekt konnte nicht gespeichert werden.");
      }

    }
```

```
    // Lies das Objekt
    public Object restoreObject(){
       Object generic = null;
       try {
          f = new FileInputStream(fileName);
          is = new ObjectInputStream(f);
       } catch ( Exception e ) {
          System.out.println("Filestream kann nicht geöffnet werden: " +
             fileName + " als Objektstrom.");
      }
      // Lies das Objekt aus dem Strom und liefere es ab.
      try {
         generic = is.readObject();
      } catch ( Exception e ) {
       System.out.println("Das Objekt kann nicht gelesen werden.");
      }
      return generic;
  }
 }
```

Auch im Hauptprogramm ergeben sich dadurch verschiedene Änderungen. Sie finden die vollständige Lösung auf der Server / der CD. Der Einfachheit halber wurde die gesamte Applikation basierend auf der Demo Applikation neu entwickelt.

Hier die Hauptklasse:

```
package zeichnenundserialisieren;

import javax.swing.UIManager;
import java.awt.*;

/**
 * Title:        Zeichnen und Serialisieren
*/

public class ZeichnenUndSerialisierenApp {
  static String motif = new
            String("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
  static String metal = new
            String("com.sun.java.swing.plaf.metal.MetalLookAndFeel");

  boolean packFrame = false;

  /**Construct the application*/
  public ZeichnenUndSerialisierenApp() {
    ZeichnenUndSerialisieren frame = new ZeichnenUndSerialisieren();
    //Validate frames that have preset sizes
    //Pack frames that have useful preferred size info, e.g. from their
layout
    if (packFrame) {
      frame.pack();
    }
    else {
      frame.validate();
    }
    //Center the window
    Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
    Dimension frameSize = frame.getSize();
    if (frameSize.height > screenSize.height) {
      frameSize.height = screenSize.height;
    }
    if (frameSize.width > screenSize.width) {
      frameSize.width = screenSize.width;
    }
```

```
    frame.setLocation((screenSize.width - frameSize.width) / 2,
(screenSize.height - frameSize.height) / 2);
    frame.setVisible(true);
  }
  /**Main method*/
  public static void main(String[] args) {
    try {
      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
      UIManager.setLookAndFeel(motif);
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    new ZeichnenUndSerialisierenApp();
  }
}
```

Und hier die Klasse ZeichnenUndSerialisieren:

```
package zeichnenundserialisieren;

import java.awt.*;
import com.borland.jbcl.layout.*;
import java.awt.event.*;

/**
 * Title:        Zeichnen und Serialisieren
*/

public class ZeichnenUndSerialisieren extends Frame {
  ZeichnenPanel restored;
  Panel northPanel = new Panel();
  Panel southPanel = new Panel();
  ZeichnenPanel centerPanel = new ZeichnenPanel();
  //public SerializationButton(String label, String fileName)
  SerializationButton restoreButton = new
SerializationButton("Restore","Painting.ser");
  SerializationButton storeButton = new
SerializationButton("Store","Painting.ser");
  CheckboxGroup colourCheckboxGroup = new CheckboxGroup();
  FlowLayout flowLayout1 = new FlowLayout();
  Checkbox rotCheckbox = new Checkbox();
  Checkbox greenCheckbox = new Checkbox();
  Checkbox blueCheckbox = new Checkbox();
  Checkbox blackCheckbox = new Checkbox();

  FlowLayout flowLayout2 = new FlowLayout();
  MenuBar menuBar1 = new MenuBar();
  Button clearButton = new Button();
  Button exitButton = new Button();
  Choice zeichnungsmodusChoice = new Choice();

  public ZeichnenUndSerialisieren() {
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
  public static void main(String[] args) {
    ZeichnenUndSerialisieren zeichnenUndSerialisieren = new
ZeichnenUndSerialisieren();
  }
  private void jbInit() throws Exception {
    this.setSize(new Dimension(628, 343));
    this.setTitle("Zeichnen und Serialisieren");
    centerPanel.setBackground(Color.red);
    northPanel.setBackground(Color.pink);
```

```
    northPanel.setLayout(flowLayout2);
    southPanel.setBackground(Color.orange);
    southPanel.setLayout(flowLayout1);
    //restoreButton.setLabel("Restore");
    //storeButton.setLabel("Store");
    rotCheckbox.setBackground(Color.orange);
    rotCheckbox.setCheckboxGroup(colourCheckboxGroup);
    rotCheckbox.setForeground(Color.red);
    greenCheckbox.setBackground(Color.orange);
    greenCheckbox.setCheckboxGroup(colourCheckboxGroup);
    greenCheckbox.setForeground(Color.green);
    blueCheckbox.setBackground(Color.orange);
    blueCheckbox.setCheckboxGroup(colourCheckboxGroup);
    blueCheckbox.setForeground(Color.blue);
    blackCheckbox.setBackground(Color.orange);
    blackCheckbox.setCheckboxGroup(colourCheckboxGroup);
    clearButton.setLabel("Clear");
    clearButton.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        clearButton_actionPerformed(e);
      }
    });
    exitButton.setLabel("Exit");
    exitButton.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        exitButton_actionPerformed(e);
      }
    });
    restoreButton.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        restoreButton_actionPerformed(e);
      }
    });
    storeButton.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        storeButton_actionPerformed(e);
      }
    });
    this.add(centerPanel, BorderLayout.CENTER);
    this.add(southPanel, BorderLayout.SOUTH);
    southPanel.add(clearButton, null);
    southPanel.add(restoreButton, null);
    southPanel.add(rotCheckbox, null);
    southPanel.add(greenCheckbox, null);
    southPanel.add(blueCheckbox, null);
    southPanel.add(blackCheckbox, null);
    southPanel.add(zeichnungsmodusChoice, null);
    southPanel.add(storeButton, null);
    southPanel.add(exitButton, null);
    this.add(northPanel, BorderLayout.NORTH);
  }

  void exitButton_actionPerformed(ActionEvent e) {
    System.exit(0);
  }

  void clearButton_actionPerformed(ActionEvent e) {
    ZeichnenPanel restored = new ZeichnenPanel();
    remove(centerPanel);
    this.add(restored);
    centerPanel = restored;
  }

  void restoreButton_actionPerformed(ActionEvent e) {
    ZeichnenPanel restored =
(ZeichnenPanel)(restoreButton.restoreObject());
    System.out.println("[restoreButton]Panel wurde De-Serialisiert");

    for (int i=0; i<restored.linien.size(); i++) {
```

```
System.out.println("[ZeichnenUndSerialisieren.restoreButton_actionPerformed
] Vector : "+restored.linien.get(i));

System.out.println("[ZeichnenUndSerialisieren.restoreButton_actionPerformed
] Vector : "+restored.farben.get(i));
    }
    remove(centerPanel);
    this.add("Center", restored);
    System.out.println ("[restoreButton]restored und added centerPanel
[ZeichnenPanel]"+restored.getName());
    this.add(restored);
    centerPanel = restored;
    System.gc();
  }

  void storeButton_actionPerformed(ActionEvent e) {
    storeButton.storeObject(this.centerPanel);
  }
}
```

Das gespeicherte Objekt heisst Painting.ser und befindet sich im Verzeichnis, aus dem die Applikation gestartet wird.

Wichtig ist, dass Sie erkennen, wie die eigentliche Information über Ihre Linien gespeichert wurden: als sogenannte Vektorgrafik, da die einzelnen Punkte in einen Vector abgespeichert und serialisiert wurden:

```
package zeichnenundserialisieren;

import java.awt.*;
import java.awt.event.*;
//import java.applet.*;
import java.util.*;
/**
 * Title:        Zeichnen und Serialisieren
 * Description:   nicht fertig!!!!!!! aber funktioniert
*/
```

```
class ZeichnenPanel extends Panel implements MouseListener,
MouseMotionListener {
    // Kodierung des Zeichenmodus
    public static final int LINIEN = 0;
    public static final int PUNKTE = 1;
    int        zeichnungsModus = LINIEN;
    Vector linien = new Vector();
    Vector farben = new Vector();
    int x1,y1;
    int x2,y2;

    public ZeichnenPanel() {
      setBackground(Color.white);
      addMouseMotionListener(this);
      addMouseListener(this);
    }

    public void setzenDesZeichnungsModus(int zeichnungsModus) {
      switch (zeichnungsModus) {
        case LINIEN:
        case PUNKTE:
          this.zeichnungsModus = zeichnungsModus;
          break;
        default:
          throw new IllegalArgumentException();
      }
    }


    public void mouseDragged(MouseEvent e) {
        // Event Tracking : ignoriere das MausDragging
        e.consume();
        switch (zeichnungsModus) {
            case LINIEN:
                x2 = e.getX();
                y2 = e.getY();
                break;
            case PUNKTE:
            default:
                farben.addElement(getForeground());
                linien.addElement(new Rectangle(x1, y1, e.getX(),
e.getY()));
                x1 = e.getX();
                y1 = e.getY();
                break;
        }
        repaint();
    }

    public void mouseMoved(MouseEvent e) {
    }

    public void mousePressed(MouseEvent e) {
        // Event Tracking : ignoriere MousePressed
        e.consume();
        switch (zeichnungsModus) {
            case LINIEN:
                x1 = e.getX();
                y1 = e.getY();
                x2 = -1;
                break;
            case PUNKTE:
            default:
                farben.addElement(getForeground());
                linien.addElement(new Rectangle(e.getX(), e.getY(), -1, -
1));
                x1 = e.getX();
                y1 = e.getY();
                repaint();
```

```
                break;
            }
        }

        public void mouseReleased(MouseEvent e) {
          // Event Tracking
            e.consume();
            switch (zeichnungsModus) {
                case LINIEN:
                    farben.addElement(getForeground());
                    linien.addElement(new Rectangle(x1, y1, e.getX(),
e.getY()));
                    x2 = -1;
                    break;
                case PUNKTE:
                default:
                    break;
            }
            repaint();
        }

        public void mouseEntered(MouseEvent e) {
        }

        public void mouseExited(MouseEvent e) {
        }

        public void mouseClicked(MouseEvent e) {
        }

        public void paint(Graphics g) {
          int np = linien.size();

          /* aktuelle Linie zeichnen */
          g.setColor(getForeground());
          for (int i=0; i < np; i++) {
              Rectangle p = (Rectangle)linien.elementAt(i);
              g.setColor((Color)farben.elementAt(i));
              if (p.width != -1) {
                g.drawLine(p.x, p.y, p.width, p.height);
              } else {
                g.drawLine(p.x, p.y, p.x, p.y);
              }
          }
          if (zeichnungsModus == LINIEN) {
              g.setColor(getForeground());
              if (x2 != -1) {
                    g.drawLine(x1, y1, x2, y2);
              }
          }
        }
}
```

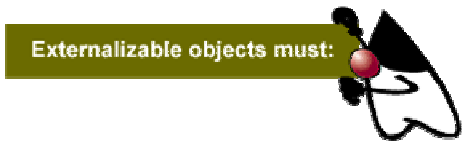
Geplant war eine Version, bei der auch noch die Controls (hier : Farbe; allgemeiner : geometrische Figuren und Texte) serialisiert werden und dadurch die Funktionalität der Applikation dynamisch verändert werden könnte, wie im MS- Office, in dem Sie auch beim Objektwechsel ein anderes Menü sehen.

Die Lösung dieser Aufgabe folgt (vermutlich) noch.

1.4.5. Serialisierung versus Externalisierung

Klassen, welche das `Serializable` Interface automatisch implementieren, können den Zustand des Objekts speichern und wieder restoren. Daneben gibt es ein weiteres Interface, das `Externalizable` Interface, welches zur Folge hat, dass die entsprchende Klasse Methoden für die Speicherung und das Lesen der Objektzustände zur Verfügung stellt.

```
package java.io;

public interface Externalizable extends Serializable {

    public void writeExternal (ObjectOutput out)
        throws IOException;

    public void readExternal (ObjectInput in)
        throws IOException, ClassNotFoundException;
}
```



Klassen die `Externalizable` sein sollen müssen also:

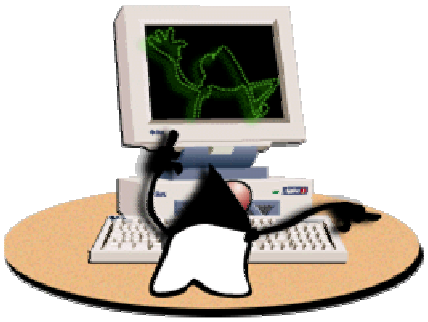
- das `java.io.Externalizable` Interface implementieren
- die Methode `writeExternal` implementieren.
  Mit dieser Methode wird der Zustand des Objekts abgespeichert. Die Methode muss sich explizit darum kümmern, dass auch alleOberklassen berücksichtigt werden.
- die Methode `readExternal` implementieren.
  Mit dieser Methode werden die Daten aus einem Stream gelesen und der Zustand des Objekts rekonstruiert. Dabei müssen auch alle Oberklassen rekoonstruiert werden.
- ein externes Format definieren.
  Die readExternal und writeExternal Methoden sind für dieses Format verantwortlich.

Jeder Externalizable Klasse ist auch serialisierbar, muss aber die Methoden zum Lesen und Schreiben der Objekte selber zur Verfügung stellen. Diese fehlen bei der Definition vollständig.

1.4.6. Objektserialisierung - Praktische Übung

In dieser Übung speichern Sie den Inhalt eines Kontos und den Zustand des Kontomanagers als Objekte, mittels Serialisierung, statt in einer Datenbank.

Immer wenn das Konto verändert wird, müssen Sie das Objekt aus der Datei lesen; immer wenn sich das Konto verändert, müssen Sie es serialisieren und damit neu abspeichern.



Hier ist der Hava Code, mit dem die üblichen Funktionen eines Kontos implementiert werden. Die Daten selber werden serialisiert abgespeichert, immer dann, wenn sich die Bilanz verändert hat.

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.RMISecurityManager;
import java.io.*;

public class AccountManagerImpl
        extends UnicastRemoteObject
        implements AccountManager {

        // The file and object input streams
        private FileInputStream fis = null;
        private ObjectInputStream inStream = null;
        private AccountImpl newAcct = null;

        // This empty constructor is required to create an
        // instance of this class in the server
        public AccountManagerImpl () throws RemoteException {
        }

        // Implement method from AccountManager interface

        // Create an instance of an Account - if the account name
        // already exists, return that account instead of creating
        // a new one
        public Account open (String name, float initialBalance)
                throws BadMoneyException, RemoteException {

                // Check the initial account value...
                if (initialBalance < 0) {
                        throw new BadMoneyException
                        ("Negative initial balance!");
                }

                // See if this account exists already on
                the local filesystem
                // file names are .acct
```

```
                String fileName = name + ".acct";
                File f = new File (fileName);

                // debug message
                System.out.println ("Looking for object file: " + fileName);

                // Does the account exist?
                if (f.exists()) {
                        // ok, deserialize the account
                        System.out.println ("Deserializing: " + fileName);
                        try {
                                fis = new FileInputStream (f);
                                inStream = new ObjectInputStream (fis);
                        } catch (Exception e) {
                                System.err.println
                                ("Unable to open stream as an object stream");
                        }

                        AccountImpl acct = null;

                        // Attempt to create a new object from the stream
                        try {
                                acct = (AccountImpl)inStream.readObject();
                        } catch (ClassNotFoundException e) {
                                System.err.println ("Class not found");
                        } catch (InvalidClassException e) {
                                System.err.println ("Invalid class");
                        } catch (StreamCorruptedException e) {
                                System.err.println ("Stream is corrupted");
                        } catch (OptionalDataException e) {
                                System.err.println
                                ("Primitive data on stream - not object");
                        } catch (IOException e) {
                                System.err.println
                                ("Unable to read Object file " + e.getMessage());
                        }

                        return (acct);
                }

                // Try to create a new account with the starting balance
                try {
                        newAcct = new AccountImpl (initialBalance, fileName);
                } catch (java.rmi.RemoteException e) {
                        System.err.println ("Error opening account: "
                        + e.getMessage());
                }

                // Return and instance of an AccountImpl object
                return (newAcct);
        }

}
```

Selbsttestaufgabe 1 Warum wird die RMIRemoteException nicht abgefangen?

In diesem Beispiel werden jede Menge Exceptions abgefangen. Das Beispiel beschreibt aber eine lokale Situation, also ohne RMI!

1.4.7. Quiz

Die folgenden Fragen überprüfen Ihr Wissen zumThema Serialisierung von Objekten.
Falls Sie Probleme beim Lösen haben, sollten Sie das Skript erneut lesen!

1. Auf welchen beiden Interfaces baut die Objektserialisierung auf?[11]
   a) java.io.ObjectOutput und java.io.ObjectInput
   b) java.io.DataInput und java.io.DataOutput
   b) java.io.Serializable und java.io.Externalizable

2. Klassen, welche das Serializable Interface implementieren, können den Zustand eines Objekts nicht speichern und lesen.[12]

[11] a)
[12] falsch

1.4.8. Zusammenfassung

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein,

- anzugeben, warum die Objekt Serialisierung wichtig ist und wofür Sie eingesetzt werden kann und wie das API aussieht.
- den Zusammenhang zwischen RMI und Objektserialisierung anzugeben.

1.5. Modul 4 : Einführung in Java IDL

In diesem Modul

- Module 4: JavaIDL
  - Einleitung
  - Object Management Group
  - Object Management Architecture
  - Portable ORB Core und JavaIDL
  - Wrapping von Legacy Code mit CORBA
  - Was ist IDL?
  - Wie funktioniert JavaIDL?
  - IDL Übersicht
  - IDL Grundlagen
  - Module Deklaration
  - Interface Deklaration
  - Operations und Parameter Deklarationen
  - Attribute Deklaration
  - Exceptions
  - Data Type Naming
  - IDL struct
  - Sequence
  - Array
  - enum und const
  - Praktische Übung - Java Interface Definition Language (JavaIDL)
  - Quiz
  - Zusammenfassung

1.5.1. Einleitung

JavaIDL ist eine Technologie zur Programmierung verteilter Systeme. JavaIDL wurde von Sun / JavaSoft entwickelt, um den Clients maximale Flexibilität bei der Integration in verteilte Systeme zu erlauben. Eines der Ziele ist es, den Client so flexibel zu bauen, dass die Technologie des Servers und der Kommunikation geändert werden kann - ohne dass der Client davon betroffen sein muss.

1.5.1.1. Lernziele

Nach dem Durcharbeiten dieser Unterlagen sollten Sie in der Lage sein:

- zu erklären, wer die OMG ist und in welchem Zusammenhang mit CORBA die OMG steht, sowie, welches typische Aufgaben der OMG sind.
- Stubs und Skeletons für eine CORBA Umgebung zu kreieren, mit Hilfe des `idlgen` Utilities.
- Java Objekte auf IDL abzubilden und mit Hilfe von JavaIDL (dem Java - IDL Mapping) zu beschreiben.

1.5.1.2. Referenzen

Teile dieses Moduls stammen teilweise oder ganz aus

- "A Note on Distributed Computing" `http://www.sunlabs.com/technical-reports/1994/abstract-29.html` (vollständig als PDF auf dem Server).
- ***Teach Yourself CORBA in 14 Days***
- Robert Orfali & Dan Harkey ***Client / Server Programming with JAVA and CORBA*** 2nd Edition (enthält sehr viele nette Bilder zum Thema).
- ***Design Patterns-Elements of Reusable Object-Oriented Software*** Gamma, Helm, Johnson und Vlissides (Addison-Wesley, 1995).

1.5.2. Die Object Management Group - OMG

Die Object Management Group (OMG) mit Hauptsitz in Framingham, Massachusetts, ausserhalb Boston (am berühmtesten Highway der Software Industrie: an dieser Umfahrungsautobahn von Boston finden Sie sogut wie alles was Rang und Namen in der Software Industrie hat) ist ein internationales nonprofit Konsortium, welches sich mit der Promotion der Theorie und der Praxis der Objekttechnologie (OT) für die Entwicklung verteilter Systeme. Ziel der OMG war ursprünglich, zu helfen, die Komplexität bei der Entwicklung solcher Systeme und deren Kosten zu reduzieren.

Innerhalb der Software-Standisierungsgremien spielt die OMG eine spezielle Rolle:

- es ist die grösste Standardisierungsgruppe der Welt, mit über 700 Mitgliedern weltweit.
- sie verkauft keine Software. Als nonprofit Organisation muss die OMG herstellerneutral sein. Natürlich versucht jeder Hersteller als Mitglied verschiedener Gremien, dominanten Einfluss zu gewinnen.
- Referenzmodelle und Architekturen, welche den Kern der OMG Standards bilden, umabhängig von irgendeiner speziellen kommerziellen Implementation.

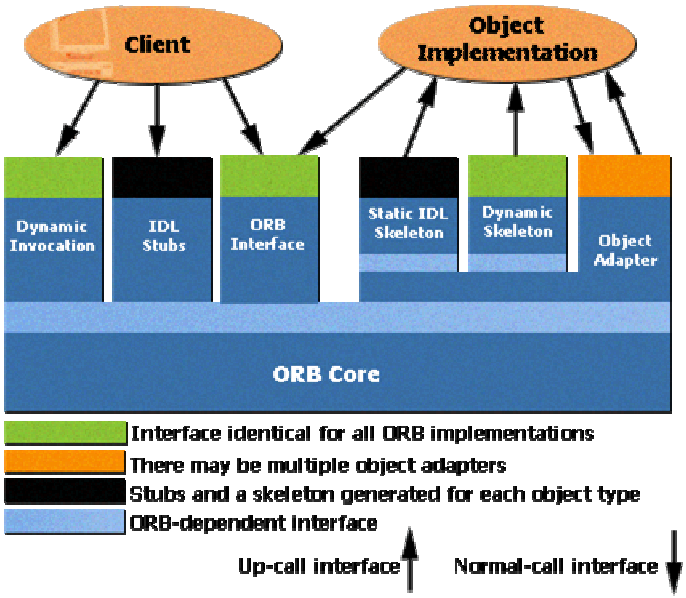
In der Startphase der OMG fokussierte sie sich darauf, die technische Infrastruktur aufzubauen und Werkzeuge zu definieren, welche die Interoperabilität garantieren: im Speziellen waren dies CORBA (Common Object Request Broker) und die OMG IDL (Interface Definition Language).

In den letzten Jahren befasste sich die OMG schwerpunktmässig mit der Verbreitung des Know Hows und der Technologien und deren Anwendungen in unterschiedlichen Anwendungsgebieten. Dies führte zu einer grösseren Reorganisation:
neben dem Platform Technical Committee wurde ein Domain Technical Commmittee gegründet.

1.5.3. Die Object Management Architektur

Der ORB ist das Kernstück des Referenzmodells. Er stellt die Kommunikationsinfrastruktr zur Verfügung, mit dessen Hilfe Objekte transparent Requests und Responses in einer verteilten Umgebung verarbeiten kann. Der ORB ist die Grundlage für den Bau von Applikationen mit verteilten Objekten und um die Interoperabilität zwischen Applikationen in heterogenen Umgebungen erreichen zu können.

Konsistent mit dem Fokus auf Interfaces spezifiziert CORBA nicht, wie der ORB implementiert wird: dies kann mit Hilfe eines Daemon Prozesses, einer Bibliothek oder mittels irgendeiner Kombination geschehen.



Bemerkung
Welche Rolle der ORB konkret in einer Applikation spielt, wird im Standard nicht festgehalten. In einigen Applikationen kann es sein, dass der ORB nur bei der Initialisierung benötigt wird, um eine erste Verbindung aufzubauen. In anderen Fällen kann der ORB eine bleibende zentrale Rolle in der Applikation spielen.

Die Object Management Architektur wird bestimmt durch die folgenden Bestandteile, die Sie auch in der Skizze sehen können:

1.5.3.1. Static und Dynamic Invocation

CORBA definiert zwei Mechanismen für den Aufruf von Objektmethoden: statische und dynamische, Static und Dynamic Invocation.

Die statischen Aufrufe sind nur möglich, falls die Interfaces zur Zeit der Übersetzung der Applikationen bekannt sind.

Die dynamischen Aufrufe gestatten es dem Client die Interfaces zur Laufzeit durch Befragung des ORBs zu bestimmen. Diese Fähigkeit ist sehr mächtig und ist eine der vitalen Vorbedingungen, um flexible Systeme zu bauen, welche häufig verändert werden. Allerdings bezahlen Sie einen sehr hohen Overhead!

1.5.3.2. Interface Repository

Die Schlüsselkomponente dank dem die Dynamic Invocation möglich wird, ist das Interface Repository. In seiner einfachsten Form besteht das Repository aus einer Datenbank, oder beispielsweise einer Hashtabelle, in die der IDL Compiler die Interface Beschreibungen einträgt. Von der Architektur her gesehen ist das Interface Repository eine der kritischsten Komponenten in CORBA. Es enthält die Metadaten für die gesamte Objektföderation, prüft die Datentypen bei den Methodenaufrufen und überprüft die Konsistenz der Methodenaufrufe auch wenn diese mehrere ORBs betreffen.

In CORBA 2.0 wurden Standard Interfaces für den Zugriff auf das Interface Repository festgelegt.

1.5.3.3. Object Adapter

Im Allgemeinen gestattet ein Objektadapter Objekten mit inkompatiblen Interfaces miteinander zu kommunizieren. CORBA definiert den Objekt Adapter als eine ORB Komponente, welche den Objektimplementationen Aktivierung / Activation, Objektreferenzierung und zustandsabhängige Services zur Verfügung stellt.

Die folgenden Aktivitäten betrachtet man als zustandsabhängige Services:

- Registrierung von Implementationsobjekten / Serverobjekten beim ORB
- interpretieren und übersetzen von Objektreferenzen
- lokalisieren der Implementationsobjekte / Serverobjekte.
- aktivieren und deaktivieren der Implementationsobjekte
- Methodenaufrufe

Der Objektadapter unterhält einen Implementations- Repository für das Speichern von Informationen, welche die Objektimplementationen beschreiben. Irgendwo muss ja die Information über das Serverobjekt gespeichert werden!

Der ORB selber muss mindestens über einen Basic Object Adapter (BOA) verfügen, so wie er in CORBA definiert wird. Daneben kann der ORB aber auch noch weitere spezielle Objektadapter definieren. Beispielsweise könnte ein spezieller Objektadapter für die Handhabung der Persistenz definiert werden.

1.5.3.4. CORBA Services

Die CORBA Services unterstützen grundlegenden Funktionen, die bei der Implementation und dem Betrieb verteilter Applikationen benötigt werden, unabhängig von der spezifischen Anwendung.

Beispielsweise definiert der Lifecycle Service Interfaces, mit deren Hilfe Objekte kreiert, gelöscht, kopiert oder verschoben werden können. Wie die Objekte in der Applikation implementiert werden müssen, bleibt dabei offen.

Die Details dieses und weiterer Dienste wurde in einem OMG Dokument *CORBAservices* festgehalten.

1.5.3.5. CORBA Facilities

CORBA Facilities sind eine Sammlung von Diensten, die gemeinsam genutzt werden können. In der Architektur sind sie auf einer höheren als die Grunddienste angeordnet.

In der Regel geht es bei horizontalen Facilities um solche, die von mehreren Applikationen genutzt werden können. Beispiele wären: das Drucken oder electronic mail Dienste.

Vertikale Facilities betreffen spezielle Anwendungsbereiche, beispielsweise Finanzapplikationen oder Produktionsplanungsfacilities.

Mehrere Facilities werden in einem OMG Dokument CORBAfacilities festgehalten.

Zu den horizontalen oder generischen Facilities gehören folgende:

- **User Interface Facilities**
  Diese umfassen das Desktop Management, Skripting, Darstellung der Komponenten und User Support Facilities.

- **Information Management Facilities**
  Austausch von Daten, Informationen, Informationsmodellierung, Informationsspeicherung und Retrieval, Datenverschlüsselung, Zeitkoordination.

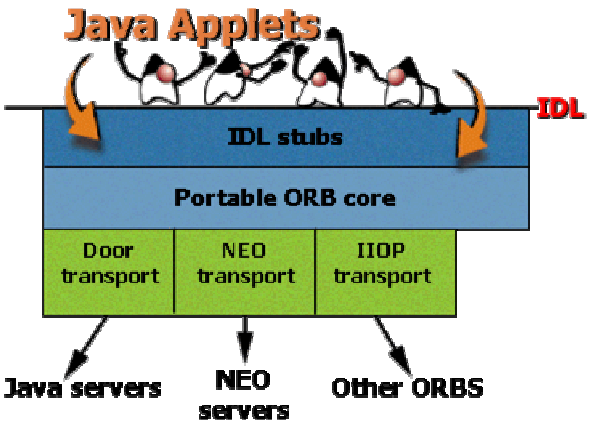
- **System Management Facilities**
  Event Management, Policy Management, Quality of Service Management, Scheduling Management, Security Facilities und Konsistenz der Daten.

- **Task Management Facilities**
  Workflow Management, Automation, OMG CORBA Agenten

Zu den vertikalen CORBA Facilities gehören zurzeit einige wenige durch die OMG beschriebene Anwendungsbereiche, konkret handelt es sich dabei um: Rechnungswesen, Anwendungsentwicklung, CIM Computer Intergrated Management, Währungen, verteilte Simulation, Information Superhighways, Internationalisierung, Geo-Mapping, Oel und Gas Exploration und Produktion, Security und Telekommunikation.

1.5.4. Portable ORB Core und JavaIDL

JavaIDL ist eine CORBA ähnliche Implementation, bei der zusätzliche Möglichkeiten von Java, vom Java Objektmodell, berücksichtigt wurden. Die JavaIDL Architektur übersetzt IDL Dateien in flexiblen und portablen Programmcode, der mit dem Kernsystem, dem Core, zusammenarbeitet. Die Coredateien werden anschliessend auf mögliche Transportsysteme (IIOP, ...) abgebildet.



Sie sollten beachten, dass JavaIDL nicht für "Java Interface Definition Language" steht - JavaIDL ist der Name eines Produkts! NEO ist der Name der CORBA Implementation auf Sun Solaris. JOE steht für die CORBA Implementation von Sun in Java.

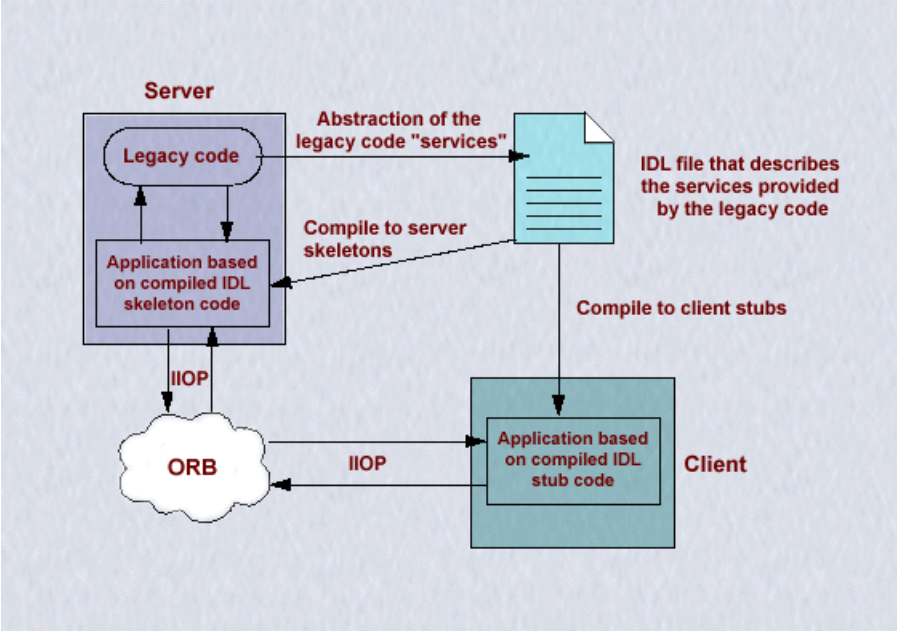
Verschiedene, eigentlich die meisten, Produkte im CORBA Umfeld unterstützen auch das Internet- Inter ORB Protokoll (IIOP).

Das obige Diagramm sollten Sie nicht überinterpretieren. Es soll lediglich veranschaulichen, dass unterschiedliche Transportprotokolle eingesetzt werden können. Dies wird durch die Aufteilung der Aufgaben in Transport Layer und Stub und Skeleton Layer möglich.

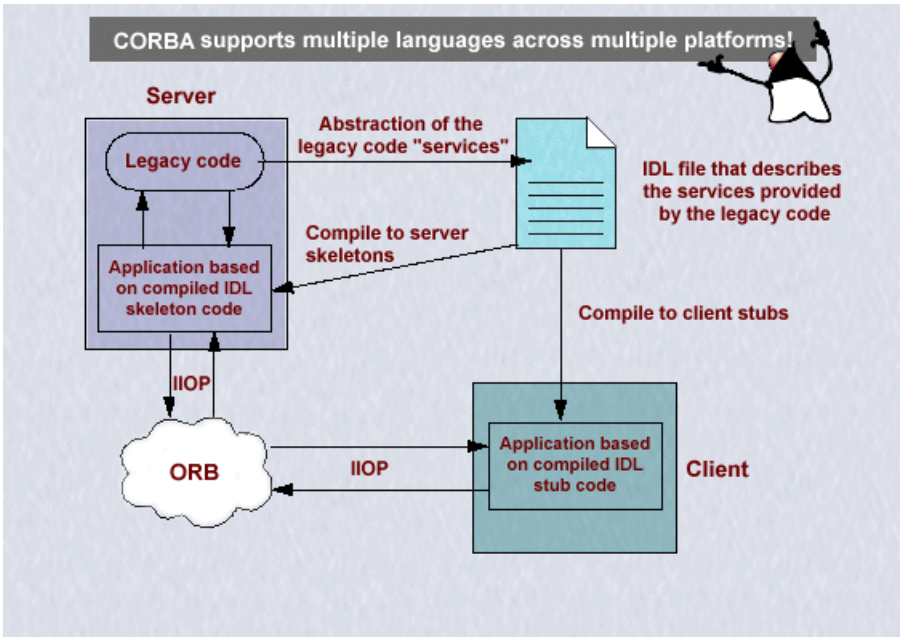
Sinnvollerweise werden Sie in Ihren Applikationen einen Standard wie IIOP einsetzen, um mehrere ORBs verbinden zu können.

1.5.5. Wrappen von Legacy Code mit CORBA

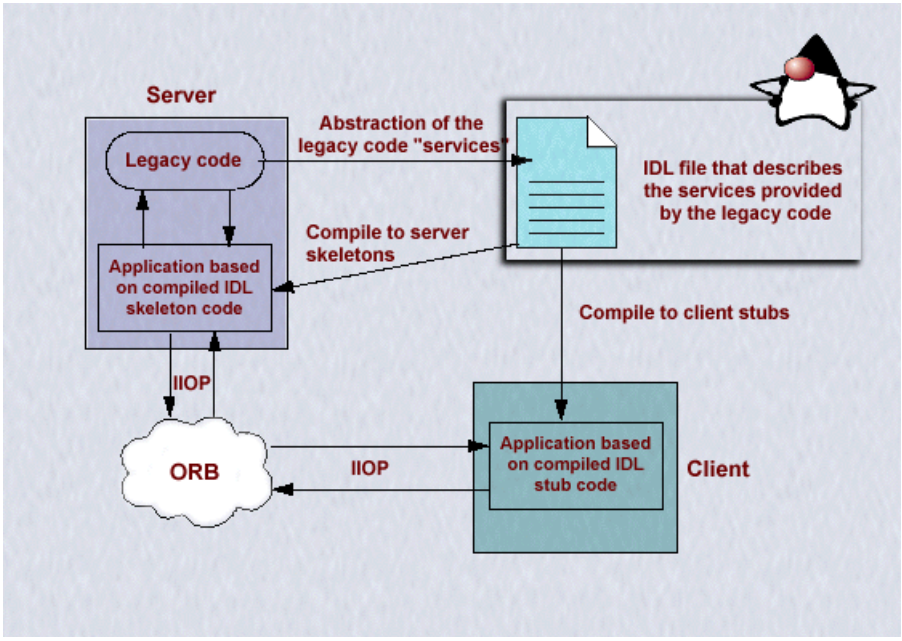
Eine der wichtigsten Fähigkeiten von CORBA ist die Unterstützung der unterschiedlichsten Programmiersprachen auf unterschiedlichsten Plattformen. Das folgende Schema zeigt, wie Legacy Code "gwrapped" werden kann, also in ein CORBA Umfeld integriert werden kann:



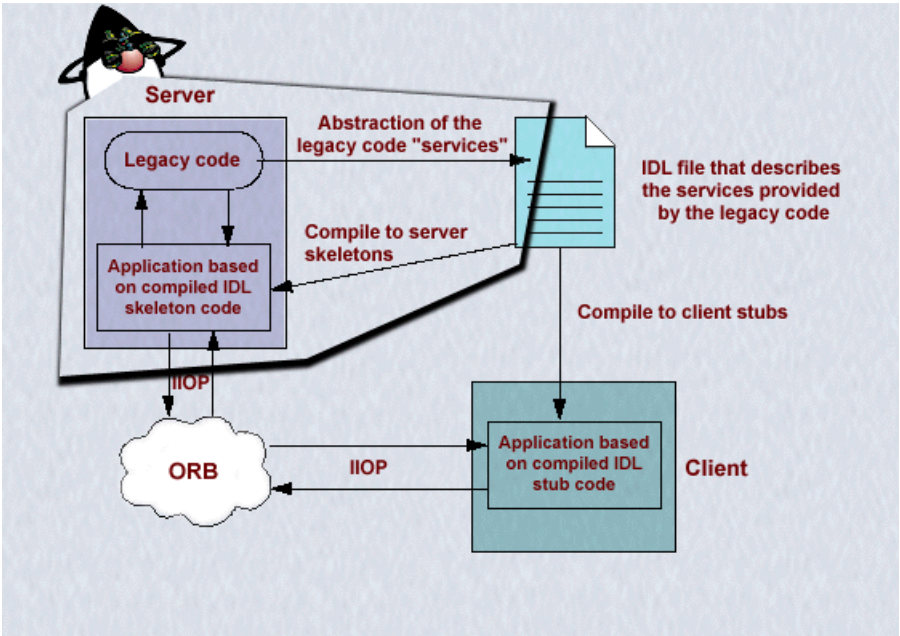
CORBA Anbieter kreierten die unterschiedlichsten Compiler für die unterschiedlichsten Plattformen und Programmiersprachen. Viele dieser Systeme dienen dem Wrappen, also demeinkapseln bestehender Anwendungen, der sog. Legacy Systeme, in die OT Welt.



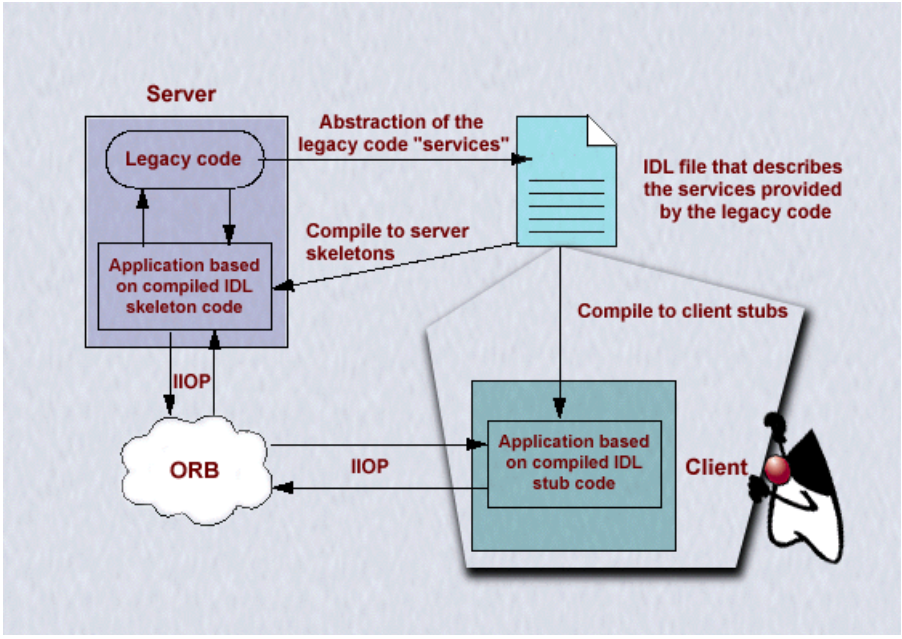
Falls Sie ein IDL Modell des Legacy Codes erstellen, stellt dies einen "Contract", eine Definition der möglichen Services der Serverimplementation (dem Legacy Code). In der Regel werden Sie Kompromisse machen müssen; aber die Grundfunktionalität wertden Sie sicher mittels IDL anbieten können.



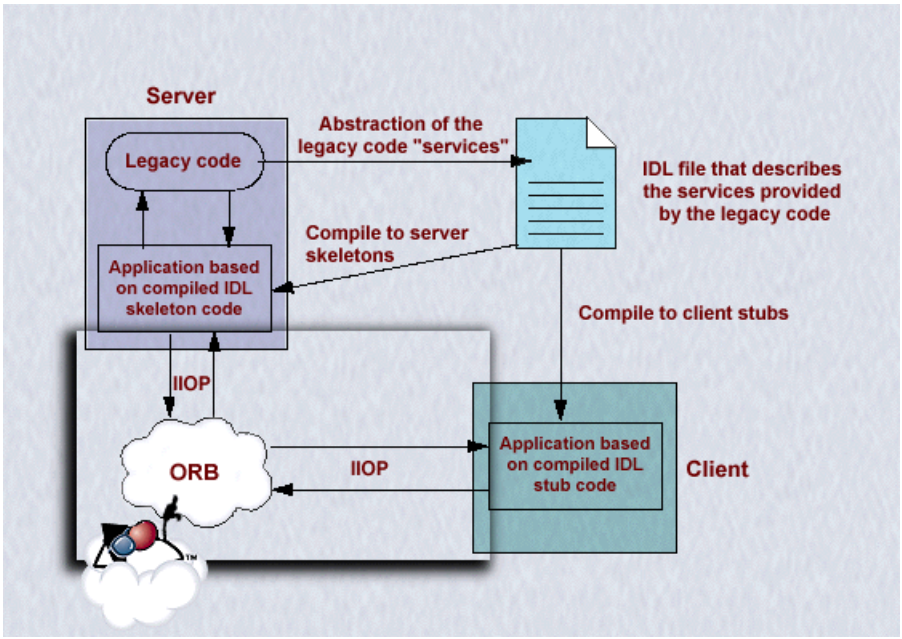
Auf der Serverseite wird IDL übersetzt und mit dem Legacy Code verbunden. Dies kann auf unterschiedliche Art und Weise geschehen: indem Legacy Bibliotheken aufgerufen werden, oder indem der Legacy Code direkt in die Skeletons, welche vom IDL Compiler generiert werden, eingebaut wird.



Auf der Client Seite wird die IDL Beschreibung mit der gewünschten Zielsprache übersetzt, beispielsweise für Java, und ab dann kann die Clientapplikation die in IDL definierten Methoden des Legacy Codes aufrufen.

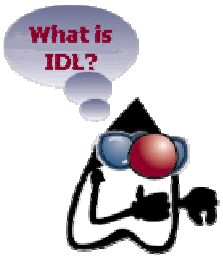


Auf beiden Seiten (Client und Server) ist die Kommunikation zwischen Stub Code und Skeleton Code und dem ORB transparent.



Die meisten ORB Implementationen offerieren einen direkten Verbindungsservice: der Client verlangt einen *'named service*' und der ORB kümmert sich um die Verbindung zum Server und das '*marshalling*' und '*unmarshalling*' des Requests.Nach dem erfolgreichen Aufbau der Verbindung, tritt der ORB zurück und "gibt" die Verbindung dem Client, so dass der Client direkt mit dem Server sprechen kann.

1.5.6. Was ist IDL?



JavaIDL ist ein System, mit dem Sie ein Set von remote Interfaces mit Hilfe der CORBA Interface Definition Language (IDL) beschreiben können. Der IDL Standard ist eine einfache Sprache, welche es Ihnen erlaubt, Interfaces für Objektsysteme zu deklarieren, ohne sich um Details der Implementation zu kümmern.

Beispiel:

```
interface MeinService {
      void operation ();
};
```

Diese Interface Definition wird an den IDL Compiler übergeben. Dieser generiert:

- Code für den Client, um einen Proxy zu kreieren, welcher eine Objektreferenz zum aktuellen `MeinService` Objekt enthält. Der Proxy Code enthält auch Stubs, mit deren Hilfe die Methode mit einer lokalen Semantik aufgerufen werden kann: es erscheint so, als ob das Objekt lokal vorhanden wäre.

- Code für die Serverseite (Skeletons), mit deren Hilfe die Interaktion zwischen Client Proxies und der aktuellen Implementation gemanaged werden kann.

Alle Programmteile, die für die Kommunikation benötigt werden, werden automatisch und vollständig durch den IDL Compiler generiert. Client und Server Code können jeweils unabhängig für eine bestimmte Zielsprache generiert werden, sofern der ORB Hersteller die entsprechende Programmiersprache und deren IDL Mapping unterstützt.

Zurzeit sind verschiedene Sprachmappings implementiert, unter anderem für C, C++, Smalltalk und Ada '95. Die meisten Produkte unterstützen die Generierung des Codes für Server und Client Seite.

1.5.7. Wie funktioniert JavaIDL?

JavaSoft's JavaIDL kreiert Stub und Skeleton Code aus einer IDL Beschreibung. Dazu wird der IDL zu Java Compiler verwendet, `idlgen`.

Synopsis
`idlgen [ Optionen ] Dateiname ...`

Beschreibung
Das `idlgen` Kommando übersetzt IDL Programmcode in Java Programmcode. Anschliesend wird mit dem `javac` Compiler dieser plus zusätzlicher Applikationscode in den Bytecode übersetzt.

Die IDL Deklarationen werden von IDL in Java übersetzt, mit Hilfe des IDL-2-Java Mappings. Der IDL Compiler generiert insbesondere Stubs und Skeletons mit deren Hilfe viele Funktionen, unabhängig vom Transport, implementiert und verwendet werden können:

- gemeinsame Dienste mit deren Hilfe Objekte für den Transport verpackt und nach dem Transport entpackt werden können (marshalling, unmarshalling).
- gemeinsame Dienste, mit deren Hilfe die aktuellen Methodenaufrufe ausgeführt werden können.
- gemeinsame Dienste, mit deren Hilfe die Resultate wieder passend eingepackt und ausgepackt werden können und dies speziell für die beim Client verwendete Programmiersprache.

`idlgen` kennt viele Optionen und Umgebungsvariablen, die Sie kontrollieren können. Auf den folgenden Seiten finden Sie eine Zusammenstellung einiger Optionen und Steuermöglichkeiten für die Generierung der Stubs und Skeletons.

1.5.7.1. Umgebungsvariablen für `idlgen`

Der `idlgen` Compiler kann die folgenden Umgebungsvariablen auch direkt aus einem sogenannten Environment File lesen. Die Syntax für diese Datei ergibt sich aus der folgenden Beschreibung. Die Datei besteht aus einer Sequenz von Anweisungen in der Form:

```
idlSymbol ':' { attribute [ '=' value ] }* ';'
```

Eine Anweisung besteht also aus einem IDL Symbol, einem Doppelpunkt und einer beliebigen Anzahl Attribute und deren Werte.

Einige der gängigen Attribute:

`RepositoryIdentifier`="*identifier*"
: Der Identifier charakterisiert das Repository.

`RepositoryPrefix`="*prefix*"
: Requests können "prefix" verwenden, um damit ein bestimmtes Repository auszuwählen. Der Standard Präfix kann auch mit
`#pragma pntifier refix "requested prefix"`
in der IDL Spezifikation angegeben werden.

`RepositoryVersion`="*major.minor*"
: Spezifiziert major.minor als Repository Version.

`serverless`
: Diese Anweisung verlangt, dass das Objekt als "serverless" Objekt behandelt wird. Serverless Objekte werden als IDL Interfaces spezifiziert und übersetzt, ausser der Compiler kann kein Stubs und Skeletons generieren. Sie können dieselbe Angabe mit
`#pragma serverless interfaceName`
am Anfang in der IDL Spezifikation machen.

`builtin`
: Diese Anweisung verlangt, dasss das Objekt als built-in Objekt behandelt wird. Built-in Objekte werden als Pseudo-Objekte behandelt: der Compiler generiert keinen Code dafür. Sie können die selben Angaben mit
`#pragma builtin interfaceName`
am Anfang in der IDL Spezifikation machen.

`javaClass`="*className*"
: Diese wird nur von built-in Objekten interpretiert. Sie bedeutet, dass das Objekt so behandelt wird, als ob es sich dabei um eine Java Klasse mit dem Klassennamen *className* handle. Die Java Klasse kann man auch mit:
`#pragma builtin interfaceName javaClass = className`
am Anfang der IDL Datei angeben.

1.5.7.2. idlgen Optionen

- `-j` *javaDirectory* - Spezifiziert, dass die generierten Java Dateien in das bezeichnete Verzeichnis geschrieben werden sollen. Dieses Verzeichnis ist unabhängig von der im Folgenden beschriebenen -p Option, falls diese überhaupt eingesetzt wird.

- `-p` Package - Spezifiziert, dass die generierten Java Symbole im angegebenen Package eingetragen werden sollen. Sie können als Trennzeichen entweder als Java Symbol spezifizieren (beispielsweise mit "." als Trennzeichen) oder aber als IDL Symbol (beispielsweise mit dem "::" Separator) oder als Verzeichnisname (mit dem "/" Trennzeichen). Package Angaben sind unabhängig von den Verzeichnisangaben in der obigen Option.

`-J` Dateiname - Spezifiziert, dass eine Liste generierter Java Dateien in eine bestimmte Datei geschrieben werden soll. Diese Datei kann eingesetzt werden, um die generierten Dateien besser kontrollieren zu können. Die Datei kann auch eingesetzt werden, um effizient aufräumen zu können, da es eine Liste der generierten Dateien enthält.

Die folgenden Optionen können entweder spezifiziert oder ausgeschaltet werden, einfach mit dem Präfix `no-`.

- `-flist-flags` - Alle `-f` Flags werden angezeigt.
  Standardwert: `off`.

- `-fcpp` - IDL Dateien werden durch den C++ Präcompiler geschleust.
  Standardwert: `on`.

- `-fclient` - Verlangt die Generierung der Stubs (für den Client)
  Standardwert: `off`.

- `-fserver` - Verlangt die Generierung der Serverdateien (Skeletons und allfällige Helper Klassen).
  Standardwert: `off`.

- `-fverbose` - Anzeige des Ablaufs.
  Standardwert: `off`.

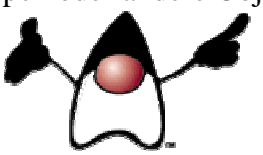
- `-fversion` - Anzeige der Version.
  Standardwert: `off`.

1.5.8. IDL Übersicht

Die Interface Definition Language (IDL) kann man sich als ein Werkzeug vorstellen, mit dem Kontrakte, Verträge zwischen Client und Server definiert werden. IDL ist aber, trotz aller Ähnlichkeit mit Standard Programmiersprachen, keine Sprache zur *Implementation* von Systemen. Schauen Sie sich das folgende Beispiel an:

```
module bank
{
      exception UngültigerBetrag
      {
            string Grund;
      };
      interface Konto
      {
            attribute string Name;
            attribute string KontoNummer;
            attribute double Kontostand;
            void Einzahlung(in double Betrag)
                  raises (UngültigerBetrag);
            void Abheben(in double Betrag)
                  raises (UngültigerBetrag);
      };
      interface Kontoabfrage: Konto
      {
            attribute double MonatsGebühr;
      };
};
```

Ein Modul definiert einen Betrachtungsbereich, einen semantischen Gültigkeitsbereich, ähnlich wie beim Konzept eines Subsystems. Ein Modul kann mehrere Interfaces enthalten. Ein Interface kann von anderen Interfaces erben (ähnlich wie beim Klassenkonzept in Java). Ein Interface kann Attribute , Ausnahmen und Operationen definieren. Attribute, wie auch Rückgabewerte sind von einem bestimmten Typus.
Attributtypen können Standard IDL Typen oder andere Objekte (inklusive Interfaces) sein,



welche in IDL definiert werden. Eine Operation besitzt einen Rückgabewert, einen Namen (unter dem sie aufgerufen werden kann) und kann Ausnahmen werfen.

Jedes Argument wird mit einer

- Richtungsangabe versehen:
  - `in` für Eingabewerte, `out` für Ausgabewerte,`inout` für Wert- Resultat Argumente (Eingabe und Ausgabe)
- Typ und
- Name.

wobei weitere IDL Schlüsselworte möglich sind.

Die IDL Deklaration wird mittels idlgen in Java Interfaces übersetzt. Mit dazugehörigen Klassen werden dann die Applikationen (Client, Server) implementiert.

1.5.9. IDL Grundlagen

Das Format von IDL wurde durch die OMG festgelegt (Section 3 der Common Object Request Broker Architektur und Spezifikation). Die Abbildung auf Java finden Sie bei Sun Microsystems, zusammen mit der Spezifikation von JavaIDL:
http://java.sun.com/products/jdk/idl/index.html *IDL to Java Language Mapping Specification*oder dort in der Nähe, falls Sun ihren Web Site wieder mal reorganisiert hat.

IDL besitzt folgende Charakteristiken:

- die gesamte IDL Datei bildet einen Namensraum, in dem jedes Element eine bestimmte Rolle spielt.
- Bezeichner, Identifier, dürfen innerhalb eines Namensraumes lediglich einmal definiert werden
- IDL Identifier können in Grossbuchstaben oder Kleinbuchstaben speziffiziert werden (es wird nicht zwischen Gross- und Kleinschreibung unterschieden).
- Operationen innehalbe eines Namensraumes können nicht überschrieben werden, da es Programmiersprachen gibt, welche das Überschreiben von Methoden / Prozeduren verbieten.

Bemerkung
Das Java Language Mapping wurde von Sun, nicht von der OMG definiert. Sun hat das Mapping eingereicht, aber es liegt bei der OMG ab wann und ob überhaupt, dieses Mapping ein offizieller Bestandteil von CORBA ist.

Diese Anerkennung kann jederzeit passieren: Sie können sich direkt bei der OMG informieren: http://www.omg.org (wobei der Server oft nicht funktioniert).

Unabhängig davon halten wir uns an die von Sun publizierte Version von JavaIDL. Sollten Sie eine Anbindung von C/C++ an CORBA im Auge haben, sollten Sie auf die OMG Literatur zurückgreifen.

1.5.10. Module Deklaration

Das Modul Konstrukt wird benutzt, um die Gültigkeitsbereiche der IDL Identifiers zu definieren. Im folgenden Beispiel ist Bank der definierende Bereich:

```
1  module Bank {
2     interface Konto {
3     ...
4     };
5  };
```

In Java wird ein Modul auf ein Package abgebildet. Im Falle des obigen Beispiels wird Konto zu Bank.Konto.

1.5.11. Interface Deklaration

Ein Interface definiert einen IDL Grunddienst. Interfaces bestehen aus Attributen, Ausnahmen und Operationen, welche von einem Object durch einen Client verlangt werden. JavaIDL bildet IDL Interfaces auf Java Interfaces ab. Der Punkt ist, dass IDL Mehrfachvererbung von Interfaces unterstützt. In Java trifft dies für Interfaces ebenfalls noch zu; allerdings nicht bei Java Klassen.

Diese Abbildung reicht noch nicht aus, um dem Client die volle Proxy Funktionalität zur Verfügung zu stellen.

Der IDL Compiler generiert für jedes IDL Interface:

1. ein Java Interface, welches die Client Sicht des IDL Interfaces definiert und die Funktionalität des Client Interfaces festhält.
2. eine Java Klasse, welche dieses Interface implementiert und die Proxy Funktionalität zur Verfügung stellt.

Schauen wir und ein Beispiel an:

```
module Bank {
  exception UngenuegenderKontostand {
    float aktuellerKontostand;
  };

  interface Konto {
    attribute wstring name;
    readonly attribute unsigned long ssn;
    readonly attribute float kontostand;
    void abheben (in float betrag) raises (UngenuegenderKontostand);
    void einzahlen (in float betrag);
  };
};
```

Diese IDL Beschreibung generiert mit folgendem Befehl:

```
@echo off
%JAVA_HOME%bin\idlj -v -fall Bank.idl
Rem @echo Die generierten Dateien stehen im Unterverzeichnis Bank
```

folgende Java Interface Beschreibungen:

```
package Bank;


/**
* Bank/Konto.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Bank.idl
* Mittwoch, 18. April 2001 15.19 Uhr GMT+02:00
*/

public interface Konto extends KontoOperations, org.omg.CORBA.Object,
org.omg.CORBA.portable.IDLEntity
{
} // interface Konto
```

sowie die Interface Beschreibung der Operationen:

```
package Bank;
/**
* Bank/KontoOperations.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Bank.idl
* Mittwoch, 18. April 2001 15.19 Uhr GMT+02:00
*/
public interface KontoOperations
{
  String name ();
  void name (String newName);
  int ssn ();
  float kontostand ();
  void abheben (float betrag) throws Bank.UngenuegenderKontostand;
  void einzahlen (float betrag);
} // interface KontoOperations
```

und die Beschreibung der Ausnahme als finale Klasse:

```
package Bank;
/**
* Bank/UngenuegenderKontostand.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Bank.idl
* Mittwoch, 18. April 2001 15.19 Uhr GMT+02:00
*/

public final class UngenuegenderKontostand extends
org.omg.CORBA.UserException implements org.omg.CORBA.portable.IDLEntity
{
  public float aktuellerKontostand = (float)0;

  public UngenuegenderKontostand ()
  {
  } // ctor

  public UngenuegenderKontostand (float _aktuellerKontostand)
  {
    aktuellerKontostand = _aktuellerKontostand;
  } // ctor

} // class UngenuegenderKontostand
```

und die folgende Java Implementation (Stub):

```
package Bank;


/**
* Bank/_KontoStub.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Bank.idl
* Mittwoch, 18. April 2001 15.19 Uhr GMT+02:00
```

```
*/

public class _KontoStub extends org.omg.CORBA.portable.ObjectImpl
implements Bank.Konto
{
  // Constructors
  // NOTE:  If the default constructor is used, the
  //        object is useless until _set_delegate (...)
  //        is called.
  public _KontoStub ()
  {
    super ();
  }

  public _KontoStub (org.omg.CORBA.portable.Delegate delegate)
  {
    super ();
    _set_delegate (delegate);
  }

  public String name ()
  {
    org.omg.CORBA.portable.InputStream _in = null;
    try {
       org.omg.CORBA.portable.OutputStream _out = _request ("_get_name",
true);
       _in = _invoke (_out);
       String __result = _in.read_wstring ();
       return __result;
    } catch (org.omg.CORBA.portable.ApplicationException _ex) {
       _in = _ex.getInputStream ();
       String _id = _ex.getId ();
       throw new org.omg.CORBA.MARSHAL (_id);
    } catch (org.omg.CORBA.portable.RemarshalException _rm) {
       return name ();
    } finally {
        _releaseReply (_in);
    }
  } // name

  public void name (String newName)
  {
    org.omg.CORBA.portable.InputStream _in = null;
    try {
       org.omg.CORBA.portable.OutputStream _out = _request ("_set_name",
true);
       _out.write_wstring (newName);
       _in = _invoke (_out);
    } catch (org.omg.CORBA.portable.ApplicationException _ex) {
       _in = _ex.getInputStream ();
       String _id = _ex.getId ();
       throw new org.omg.CORBA.MARSHAL (_id);
    } catch (org.omg.CORBA.portable.RemarshalException _rm) {
       name (newName);
    } finally {
        _releaseReply (_in);
    }
  } // name

  public int ssn ()
  {
    org.omg.CORBA.portable.InputStream _in = null;
    try {
       org.omg.CORBA.portable.OutputStream _out = _request ("_get_ssn",
true);
       _in = _invoke (_out);
       int __result = _in.read_ulong ();
       return __result;
    } catch (org.omg.CORBA.portable.ApplicationException _ex) {
       _in = _ex.getInputStream ();
```

```
        String _id = _ex.getId ();
        throw new org.omg.CORBA.MARSHAL (_id);
      } catch (org.omg.CORBA.portable.RemarshalException _rm) {
        return ssn ();
      } finally {
         _releaseReply (_in);
      }
    } // ssn

    public float kontostand ()
    {
      org.omg.CORBA.portable.InputStream _in = null;
      try {
        org.omg.CORBA.portable.OutputStream _out = _request
("_get_kontostand", true);
        _in = _invoke (_out);
        float __result = _in.read_float ();
        return __result;
      } catch (org.omg.CORBA.portable.ApplicationException _ex) {
        _in = _ex.getInputStream ();
        String _id = _ex.getId ();
        throw new org.omg.CORBA.MARSHAL (_id);
      } catch (org.omg.CORBA.portable.RemarshalException _rm) {
        return kontostand ();
      } finally {
         _releaseReply (_in);
      }
    } // kontostand

    public void abheben (float betrag) throws Bank.UngenuegenderKontostand
    {
      org.omg.CORBA.portable.InputStream _in = null;
      try {
        org.omg.CORBA.portable.OutputStream _out = _request ("abheben",
true);
        _out.write_float (betrag);
        _in = _invoke (_out);
      } catch (org.omg.CORBA.portable.ApplicationException _ex) {
        _in = _ex.getInputStream ();
        String _id = _ex.getId ();
        if (_id.equals ("IDL:Bank/UngenuegenderKontostand:1.0"))
           throw Bank.UngenuegenderKontostandHelper.read (_in);
        else
             throw new org.omg.CORBA.MARSHAL (_id);
      } catch (org.omg.CORBA.portable.RemarshalException _rm) {
        abheben (betrag);
      } finally {
         _releaseReply (_in);
      }
    } // abheben

    public void einzahlen (float betrag)
    {
      org.omg.CORBA.portable.InputStream _in = null;
      try {
        org.omg.CORBA.portable.OutputStream _out = _request ("einzahlen",
true);
        _out.write_float (betrag);
        _in = _invoke (_out);
      } catch (org.omg.CORBA.portable.ApplicationException _ex) {
        _in = _ex.getInputStream ();
        String _id = _ex.getId ();
        throw new org.omg.CORBA.MARSHAL (_id);
      } catch (org.omg.CORBA.portable.RemarshalException _rm) {
        einzahlen (betrag);
      } finally {
         _releaseReply (_in);
      }
    } // einzahlen
```

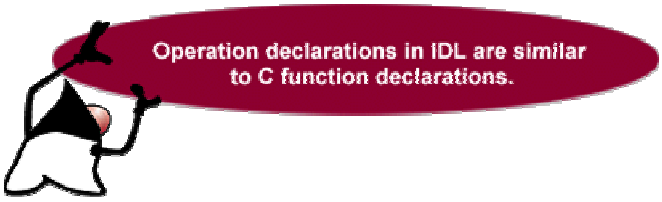
```
  // Type-specific CORBA::Object operations
  private static String[] __ids = {
    "IDL:Bank/Konto:1.0"};

  public String[] _ids ()
  {
    return (String[])__ids.clone ();
  }

  private void readObject (java.io.ObjectInputStream s)
  {
     try
     {
       String str = s.readUTF ();
       org.omg.CORBA.Object obj = org.omg.CORBA.ORB.init
().string_to_object (str);
       org.omg.CORBA.portable.Delegate delegate =
((org.omg.CORBA.portable.ObjectImpl) obj)._get_delegate ();
       _set_delegate (delegate);
     } catch (java.io.IOException e) {}
  }

  private void writeObject (java.io.ObjectOutputStream s)
  {
     try
     {
       String str = org.omg.CORBA.ORB.init ().object_to_string (this);
       s.writeUTF (str);
     } catch (java.io.IOException e) {}
  }
} // class _KontoStub
```

Zudem werden für die Klassen `Konto` und die Ausnahmeklasse `UngenuegenderKontostand` je eine Helper und eine Holder Hilfsklasse generiert:

```
Konto.java
KontoHelper.java
KontoHolder.java
KontoOperations.java
UngenügenderKontostand.java
UngenuegenderKontostandHelper.java
UngenuegenderKontostandHolder.java
_KontoImplBase.java
_KontoStub.java
```

Bevor wir auf diese Dateien weiter eingehen und den Server bzw. Client bauen, schauen wir uns die Definition von IDL noch genauer an.

1.5.12. Operationen und Parameter Deklarationen



- Eine Operation besteht aus einem optionalen Attribut, dem `oneway` Attribut. Dieses Attribut gestattet Operationen beim Server aufzurufen, welche keine Antwort liefern - eine mögliche Methode, um das Blockieren von CORBA Aufrufen zu umgehen.

- Eine Operation besteht aus einer Angabe des Rückgabetyps der Operation oder `void`

- Eine Operation besteht aus einem Bezeichner, der die Operation in ihrem Gültigkeitsbereich benennt.

- Eine Operation besitzt keinen oder mehrere Parameter.

- Eine Operation besteht aus einer optionalen `raises` Klausel.

Hier ein Beispiel:
```
void abheben (in float betrag) raises (UngenuegenderKontostand);
```

Ein Operationsparameter kann ein IDL Basisdatentyp oder ein benutzerdefinierter Datentyp sein.

Der Parameter muss ein direktionales Attribut besitzen, welches den Kommunikationsservice Client- und Server-seitig informiert, in welche Richtung der Parameter verschoben wird. Die Richtungsangabe kann sein:

- `in` falls der Parameter vom Client zum Server verschoben wird.
- `out` falls der Parameter vom Server zum Client verschoben wird.
- `inout` falls der Parameter in beide Richtungen verschoben wird.

Da Java nur die wertmässige Parameterübergabe (*by-value*), keine Referenzübergabe kennt, können `out` und `inout` nicht direkt auf Basisdatentypen abgebildet werden. Daher werden die Basisdatentypen `int, float, boolean, ...` auf entsprechende Holder Klassen im CORBA Package abgebildet.

Zum Beispiel wird `int` auf `IntHolder` abgebildet, falls die Variable das `out` oder `inout` Attribut aufweist.

1.5.13. Attribut Deklarationen

Ein Interface kann neben Operationen auch noch Attribute besitzen. Eine Attribut Definition ist logisch äquivalent zu einer Deklaration einer Zugriffs- und einer Mutationsfunktion. Diese lesen und setzen Attributwerte.

Das optionale `readonly` Schlüsselwort zeigt an, dass es sich lediglich um eine Zugriffsfunktion handelt.

Beispiel:
betrachten wir einmal folgende Interfacebeschreibung in IDL

```
interface Konto {
    readonly attribute float kontostand;
    attribute long kontoNummer;

    void einzahlen(in float betrag);
    void transfer(inout float transferbetrag);
    float auszahlen(out float auszahlung);
};
```

Dieses liefert folgende Java Interface Beschreibung:

```
/**
* KontoOperations.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Konto.idl
* Mittwoch, 18. April 2001 18.56 Uhr GMT+02:00
*/

public interface KontoOperations
{
  float kontostand ();
  int kontoNummer ();
  void kontoNummer (int newKontoNummer);
  void einzahlen (float betrag);
  void transfer (org.omg.CORBA.FloatHolder transferbetrag);
  float auszahlen (org.omg.CORBA.FloatHolder auszahlung);
} // interface KontoOperations
```

Die Aufteilung der Kontonummer in eine Methode, welche einen Integer Wert zurück liefert und eine reine Aufrufmethode ist aus dem Beispiel klar ersichtlich.

Um die Abbildung der Basisdatentypen auch noch zu dokumentieren, wurden die zusätzlichen Methoden / (IDL) Operationen mit entsprechenden Parametern definiert.

1.5.14. Exceptions

Mit `raises` wird angegeben, dass eine Ausnahme geworfen werden kann, im Rahmen eines Aufrufes einer Operation.

Die Syntax dafür sieht folgendermassen aus:

```
raises (MeineException1 [, MeineException2 ...] )
```

Die Ausnahmen, welche durch eine Operation geworfen werden, können entweder operationsspezifisch sein, oder aber eine Standardausnahme. Diese Standardausnahmen brauchen nicht zwangsweise aufgelistet zu werden.

Eine Ausnahmedeklaration gestattet die Deklaration einer `struct` ähnlichen Datenstruktur, welche als Rückgabewert auftreten kann und das Auftreten einer aussergewöhnlichen Bedingung während der Ausführung des Aufrufes anzeigt.

Die Syntax einer solchen Deklaration sieht folgendermassen aus:

```
exception <identifier> { <Member>* }
```

Beispiel:

```
exception UngenuegenderKontostand {float aktuellerKontostand; };
```

Eine Exception besitzt einen Bezeichner (Identifier) und keine, einen oder mehrere Memberwerte. Falls eine Ausnahme bei einem Aufruf geworfen wird, dann ist ein Zugriff auf den Bezeichner möglich, um damit irgend eine Fehlerbehandlung einzuleiten.

Die Umsetzung in Java sehen Sie an folgendem generierten Interface mit obiger Ausnahme:

```
public interface KontoOperations
{
  String name ();
  void name (String newName);
  int ssn ();
  float kontostand ();
  void abheben (float betrag) throws Bank.UngenuegenderKontostand;
  void einzahlen (float betrag);
} // interface KontoOperations
```

Eine Java IDL Ausnahme wird wie oben ersichtlich auf eine Java Exception abgebildet. Diese selber ist wiederum eine Erweiterung von einer `org.omg.CORBA.UserException` Ausnahme:

```
public final class UngenuegenderKontostand extends
  org.omg.CORBA.UserException implements org.omg.CORBA.portable.IDLEntity {
  public float aktuellerKontostand = (float)0;
  public UngenuegenderKontostand ()  {
  } // ctor
  public UngenuegenderKontostand (float _aktuellerKontostand) {
    aktuellerKontostand = _aktuellerKontostand;
  } // ctor
} // class UngenuegenderKontostand
```

1.5.15. Bezeichnung der Datentypen

IDL enthält Konstrukte für die Benennung von Datentypen. Das `typedef` Schlüsselwort wird benutzt, um einen Datentyp mit einem Namen zu versehen.

Beispiele:

```
typedef     long        IDNummer
typedef     string      SSNummer
```

Die Java Programmiersprache kennt kein Sprachkonstrukt, welches der Typendefinition von IDL entspricht. Auch die Typendefinition für einfache Datentypen werden nicht direkt auf Java Konstrukte abgebildet.

Die folgende Zusammenstellung zeigt die Abbildung der IDL auf die Java Datentypen:

IDL	Java
float	float
double	double
unsigned long	int
long long	long
long	int
short	short
unsigned long	int
unsigned short*	int
char	char
wchar	char
boolean	boolean
octet	byte
string	String (Klasse)
wstring	String
enum	int
fixed	java.mathBigDecimal

IDL boolean Werte sind TRUE und FALSE; in Java werden daraus true und false.

1.5.16. IDL struct - Strukturen

Die IDL struct Anweisung wird eingesetzt, um Daten, die irgendwie zusammengehören, als eine Einheit verwendet zu können, beispielsweise als Parameter. Das `struct` Konstrukt wird auf eine Klasse abgebildet, welche Instanzen für die Felder und einen Konstruktor zur Verfügung stellt.

Am Besten sehen Sie wie der Mechanismus funktioniert, wenn Sie ein Beispiel anschauen:

```
module bank {
    struct KontoInfo {
        string name;
        float kontostand;
    };
};
```

und nun das Ganze in Java:

```
package bank;

public final class KontoInfo implements org.omg.CORBA.portable.IDLEntity
{
  public String name = null;
  public float kontostand = (float)0;

  public KontoInfo ()
  {
  } // ctor Konstruktor

  public KontoInfo (String _name, float _kontostand)
  {
    name = _name;
    kontostand = _kontostand;
  } // ctor Konstruktor mit Parametern

} // class KontoInfo

}
```

1.5.17. Sequenzen

Eine IDL Sequenz, `sequence`, ist ein eindimensionales Datenfeld mit einer maximalen Länge, die zur Übersetzungszeit festgelegt wird, sowie einer (aktuellen) Länge, welche zur Laufzeit festgelegt wird.

Eine Sequenz kann beschränkt oder unbeschränkt sein. Eine beschränkte Sequenz definiert ihre maximal erlaubte Länge.

Syntax:

```
sequence    <long>      UnbeschraenkteSeq;
sequence    <long, 10>  BeschraenkteSeq;
```

Eine so deklarierte Sequenz kann beispielsweise in Strukturen, `struct`, oder Vereinigungen, `union`, eingesetzt werden. Falls man eine Sequenz als Attribut oder als Parameter einer Operation verwendet werden soll, muss sie Teil einer Typendefinition, `typedef`, sein.

Hier ein Beispiel:

```
typedef     sequence    <long, 10>  longZehn;
            attribute   longZehn    vector;
```

Das folgende Beispiel zeigt einige weitere Aspekte der IDL Beschreibung:

```
module bank {
      struct KundenDetails {
            string Name;
            string Address;
      };
      typedef sequence<KundenDetails> UnbeschraenkteSeq;
      typedef sequence<KundenDetails, 5> BeschraenkteSeq;
};
```

Der IDL Compiler bildet diese Angaben auf eine Java Dateien ab. Schauen wir uns zuerst die Abbildung der Struktur in der Datei `KundenDetails.java` an:

```
package bank;

public final class KundenDetails implements
org.omg.CORBA.portable.IDLEntity
{
  public String Name = null;
  public String Address = null;

  public KundenDetails ()
  {
  } // ctor

  public KundenDetails (String _Name, String _Address)
  {
    Name = _Name;
    Address = _Address;
  } // ctor

} // class KundenDetails
```

Der IDL Compiler generiert auch eine Klasse pro Typendefinition, als 'Holder' Klasse.

```
package bank;

public final class BeschraenkteSeqHolder implements
org.omg.CORBA.portable.Streamable
{
  public bank.KundenDetails value[] = null;

  public BeschraenkteSeqHolder ()
  {  }

  public BeschraenkteSeqHolder (bank.KundenDetails[] initialValue)
  {
    value = initialValue;
  }

  public void _read (org.omg.CORBA.portable.InputStream i)
  {
    value = bank.BeschraenkteSeqHelper.read (i);
  }

  public void _write (org.omg.CORBA.portable.OutputStream o)
  {
    bank.BeschraenkteSeqHelper.write (o, value);
  }

  public org.omg.CORBA.TypeCode _type ()
  {
    return bank.BeschraenkteSeqHelper.type ();
  }
}
```

Sie sehen keinen Unterschied in der Holder Klasse! Aber in den Helper-Klassen treten Unterschiede auf, erwartungsgemäss.

1.5.18. IDL Arrays

IDL Arrays werden in Java genau so abgebildet wie beschränkte IDL Sequenzen. Die Grenze des Arrays, seine Dimension, wird beim Übermitteln, dem Marshalling, in einer IDL Operation überprüft. Falls man die IDL Dimension eines Arrays in der Java Umgebung benötigt, muss diese in der IDL Umgebung als Konstante definiert werden.

Beispiel:

```
module bank {
      struct KundenDetails {
            string Name;
            string Address;
      };
      typedef sequence<KundenDetails> UnbeschraenkteSeq;
      typedef sequence<KundenDetails, 5> BeschraenkteSeq;

      const long arrayGrenze = 10;
      struct namesListe {
            string namen[arrayGrenze];
      };

};
```

Und hier der generierte Java Programmcode:

```
package bank;

public final class namesListe implements org.omg.CORBA.portable.IDLEntity
{
  public String namen[] = null;

  public namesListe ()
  {
  } // ctor

  public namesListe (String[] _namen)
  {
    namen = _namen;
  } // ctor

} // class namesListe
```

zusätzlich wird eine Datei `arrayGrenze.java` mit den Angaben zu den Arraygrenzen:

```
package bank;

public interface arrayGrenze extends org.omg.CORBA.portable.IDLEntity
{
  public static final int value = (int)(10);
}
```

1.5.19. Die IDL `enum` und `const` Konstrukte

Die IDL enum (Aufzählung) Konstrukte werden in Java auf Klassen abgebildet mit je einer statischen finalen Variable pro Member des Auflistungstyps.

Beispiel:

```
...
      enum BankStandorte { Zuerich, Frankfurt, London, Paris, Tokyo,
Singapore, Boston, NewYork, Chicago };
...
```

Daraus wird folgende Java Datei:

```
package bank;

public class BankStandorte implements org.omg.CORBA.portable.IDLEntity {
  private        int __value;
  private static int __size = 9;
  private static bank.BankStandorte[] __array = new bank.BankStandorte
[__size];

  public static final int _Zuerich = 0;
  public static final bank.BankStandorte Zuerich = new
bank.BankStandorte(_Zuerich);
  public static final int _Frankfurt = 1;
  public static final bank.BankStandorte Frankfurt = new
bank.BankStandorte(_Frankfurt);
  public static final int _London = 2;
  public static final bank.BankStandorte London = new
bank.BankStandorte(_London);
  public static final int _Paris = 3;
  public static final bank.BankStandorte Paris = new
bank.BankStandorte(_Paris);
  public static final int _Tokyo = 4;
  public static final bank.BankStandorte Tokyo = new
bank.BankStandorte(_Tokyo);
  public static final int _Singapore = 5;
  public static final bank.BankStandorte Singapore = new
bank.BankStandorte(_Singapore);
  public static final int _Boston = 6;
  public static final bank.BankStandorte Boston = new
bank.BankStandorte(_Boston);
  public static final int _NewYork = 7;
  public static final bank.BankStandorte NewYork = new
bank.BankStandorte(_NewYork);
  public static final int _Chicago = 8;
  public static final bank.BankStandorte Chicago = new
bank.BankStandorte(_Chicago);

  public int value ()  {
    return __value;
  }

  public static bank.BankStandorte from_int (int value)  {
    if (value >= 0 && value < __size)
      return __array[value];
    else
      throw new org.omg.CORBA.BAD_PARAM ();
  }

  protected BankStandorte (int value)  {
    __value = value;
    __array[__value] = this;
  }
} // class BankStandorte
```

Bei Konstanten verhält es sich ähnlich: eine IDL Konstante, als `const` in IDL gekennzeichnet. Sie sind immer dann sinnvoll, wenn einfache Werte unveränderlich bleiben sollen. In Java werden die Konstanten auf eine Klasse abgebildet, welche `public final` ist.

In IDL

```
...
      const float pi=3.14159256;
...
```

und in Java (Datei `pi.java`):

```
package bank;

public interface pi extends org.omg.CORBA.portable.IDLEntity
{
  public static final float value = (float)(3.14159256);
}
```

Damit haben wir eine ganze Menge Dateien erzeugt, welche jeweils bestimmte Aspekte von IDL erläuterten. Dies ist jedoch nur ein kurzer Auszug aus der vollständigen IDLTOJAVA *IDL to Java Language Mapping Specifikation* der OMG.

Diese Spezifikation enthält auch viele weitere Beispiele, die teils sehr illustrativ sind. Falls Sie IDL Spezifikationen entwickeln wollen oder müssen, werden Sie auf diese Spezifikation zurück greifen müssen.

Wir haben auch jeweils nur einige der generierten Dateien angeschaut, speziell jene, die für das jeweilige Konzept typisch sind.

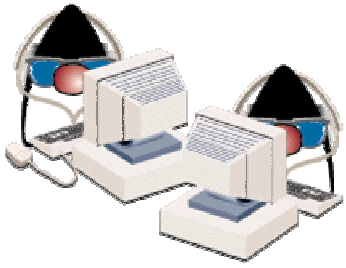
Zu beachten ist, dass in der Literatur oft noch von der (alten) IDLtoJava und dem idlgen Compiler die Rede ist. Diese wurden durch das offizielle Sprachmapping der OMG in Zusammenarbeit mit Sun abgelöst!

An Stelle von idlgen.exe wird nun idlj.exe zur Generierung der Java Dateien aus der IDL Beschreibung angewandt. Viele Umgebungsoptionen, die in idlgen wichtig waren und komplex beschrieben wurden, entfallen somit vollständig. idlgen existierte nie als Produkt, sondern lediglich als Beta / PreRelease.

1.5.20. Übung - Interface Definition Language Mapping für Java

In diesem Abschnitt schauen wir uns eine IDL Beschreibung für eine Fluggesellschaft an. Wir betrachten drei Interface Beschreibungen, je eine für die Kundeninformationen, die verfügbaren Flüge und Flugreservationsinformationen.

Die darin enthaltenen Informationen sind analog zur Beschreibung in der RMI Skizze für dieses bzw. ein analoges Informationssystem.



Die folgende Interface Beschreibung enthält drei Teile:

1. Kundeninformationen:
   KundenInfo definiert Zugriffsmethoden, um die Kundenliste abzufragen, individuelle Kundeninformationen zu verwalten und neue Kunden einzutragen
2. Fluginformationen:
   die Funktionalität des Flug Interfaces prüft, ob Flüge verfügbar sind, für bestimmte Abflug-Städte und Destinationen
3. Reservation:
   FlugReservation, die Reservationsschnittstelle beschreibt die Struktur einer Reservation und generiert eine Flugbestätigungsnummer

```
module Fluggesellschaft {
      // 1. Kundeninformationen
      struct KundenInfo {
            string kundenID;
            string nachName;
            string vorName;
            string strassenAdr;
      };
      typedef sequence <KundenInfo> KundenInfoListe;
      interface Kunde {
            // Lesen eines  Kundendatensatzes
            KundenInfo liesKunde (in string kundenID);
            // Lesen einer Liste der Kunden zu einem gegebenen Namen
            KundenInfoListe liesKundenListe (in string vorName,
                                             in string nachName);
            // Schreiben eines neuen Kundendatensatzes
            void schreibNeueKundenInfo (in string kundenID,
                                        in string nachName,
                                        in string vorName,
                                        in string strassenAdr);
            // Mutation eines bestehenden Datensatzes
            void mutiereBestehendeKundenInfo (in string kundenID,
                                              in string nachName,
                                              in string vorName,
                                              in string strassenAdr);

            // Produktion einer neuen ID für diesen neuen Kunden
            string produziereKundenID (in string vorName,
                                       in string nachName);
```

```
            // Produktion einer Zeichenkette für den Kundeninfo Container
            string produziereKundenString (in KundenInfo info);
      };

      // 2. Fluginformationen
      struct FlugInfo {
            string flugID;
            string abflugStadt;
            string destinationStadt;
            string abflugDatum;
            string abflugZeit;
            // FC=First Class; BC=Business Class; EC=Economy Class
            long anzahlVerfuegbarerFCSitze;
            long anzahlVerfuegbarerBCSitze;
            long anzahlVerfuegbarerECSitze;
            string flugzeugID;
      };

      typedef sequence <FlugInfo> FlugInfoListe;

      // unlimitiertes Arrays mit Abflug- und Destinations-Städten
      typedef sequence <string> AbflugStadtListe;
      typedef sequence <string> DestinationStadtListe;

      // 3. Flugreservation

      // Kunden Reservation Datensatz
      struct KundenReservationInfo {
            string kundenID;
            string flugNummer;
            string bestaetigungsNummer;
            string serviceKlasse;
      };

      interface Flug {

            // Ausgabe der Liste der Abflug-Sädte
            //
            AbflugStadtListe liesAlleAbflugStaedte ();

            // Ausgabe der Ziel-Flughäfen
            DestinationStadtListe liesAlleDestinationen ();

            // Ausgabe einer Liste mit Start und Ziel und Datum
            FlugInfoListe liesVerfuegbareFluege (in string abflugOrt,
                                                 in string destOrt,
                                                 in string datum);

            // Mutation der Flug Info
            boolean mutiereAnzahlVerfSitze (in KundenReservationInfo resv);

            // Kreieren einer Zeichenkette für den Flug
            string produziereFlugString (in FlugInfo flug);
      };

      // Reservations Agent
      interface Reservation {
            // Reservation durchführen
            void eingabeReservation (in string kundenID,
                                     in string flugNummer,
                                     in string bestaetigungsNummer,
                                     in string serviceKlasse);
            // Lies die Bestätigungs-Nummer
            string produziereBestaetigungsNummer();
      };
};
```

Selbsttestaufgabe 2

Inwiefern unterscheidet sich die obige IDL Definition vom Beispiel, das wir schrittweise entwickelten?[13]

Selbsttestaufgabe 3

Sie übersetzen eine IDL Beschreibung mit folgendem Befehl

```
idlj –fall Bank.idl
```

Was bedeutet die Option `–fall`?[14]

Selbsttestaufgabe 4

Im Folgenden sehen Sie drei Klassen, welche schematisch die Interface Beschreibung des Flugreservationssystems implementieren. Die Java Beschreibung ist in mehrerer Hinsicht unvollständig und könnte auch noch einige Tippfehler enthalten. Wichtig ist ein anderer Aspekt.

Ihre erste Aufgabe ist es, diese Listings genauer anzuschauen und zwar in Hinblick auf Exceptions.
Anschliessend, nach den Listings, sollten Sie die sehr leichte Frage problemlos beantworten können.

Listing 1 KundeImpl.java

```
// Implementationsklasse für den Customer Agent
// Implementation der IDL Operations
//
public class KundeImpl implements Kunden {

      private Database db = null;

      public KundeImpl (Database db) {
            this.db = db;
      }


      // lies einen Kundendatensatz
      public KundenInfo liesKunde(String kundenID)
      throws sunw.corba.SystemException {
            KundenInfo kunde = null;
            try {
                  kunde = db.liesKunde(kundenID);
            } catch (Exception e) {
                  System.out.println ("[liesKunde] Exception: " + e);
            }
            return kunde;
      }

      // lies ein Array mit Customer Records
      public KundenInfo[] liesKundenListe(String nachName, String vorName)
      throws sunw.corba.SystemException {
            KundenInfo [] kunden = null;
```

[13] wir haben keine Exceptions definiert
[14] damit werden Client und Server Dateien generiert (Stubs, Helper, Holder....)

```
            try {
                  kunden = db.liesKundenListe(vorName, nachName);
            } catch (Exception e) {
                  System.out.println ("[liesKundenListe] Exception: " + e);
            }
            return kunden;
      }
      // schreib einen neuen Customer in die Datenbank
      public void schreibNeueKundenInfo(  String kundenID,
                                          String nachName,
                                          String vorName,
                                          String strassenAdr)
      throws sunw.corba.SystemException   {
            try {
                  db.schreibNeueKundenInfo(     kundenID,
                                                nachName,
                                                vorName,
                                                strassenAdr);
            } catch (Exception e) {
            System.out.println ("[schreibNeueKundenInfo] Exception: " + e);
            }
      }
      // schreib einen bestehenden Customer in die Datenbank
      public void mutiereBestehendeKundenInfo(  String kundenID,
                                                String nachName,
                                                String vorName,
                                                String strassenAdr)
      throws sunw.corba.SystemException {
            try {
      db.mutiereBestehendeKundenInfo(     kundenID,
                                                      nachName,
                                                      vorName,
                                                      strassenAdr);
            } catch (Exception e) {
      System.out.println ("[mutiereBestehendeKundenInfo] Exception: " + e);
            }
      }
      // kreiere eine Zufalss Customer ID
      public String produziereKundenID(String nachName, String vorName)
      throws sunw.corba.SystemException {
            char FN = Character.toUpperCase(vorName.charAt(0));
            char LN = Character.toUpperCase(nachName.charAt(0));
            // was solls
            int ID = (int)(Math.random()*1000000);
            String kundenID = new String (new StringBuffer().
                                          append(FN).
                                          append(LN).
                                          append(Integer.toString(ID)));
            return kundenID;
      }
      // Kundeninfo als Zeichenkette
      public String produziereKundenString(KundenInfo info)
      throws sunw.corba.SystemException {
            String KundenString =   info.kundenID + "-" +
                                    info.nachName + "-" +
                                    info.vorName + "-" +
                                    info.strassenAdr;
            return KundenString;
      }
}
```

Nun die fast triviale Frage:

welche Exception wird bei solchen CORBA Anwendungen am meisten geworfen?[15]
a) ...corba.System.Exception
b) ...RemoteException
c) MsqlException

Und nun noch eine Skizze für den CORBA basierten Server. Der Programmcode funktioniert so nicht (Teile der Implementation fehlen)! Er dient lediglich der Illustration einzelner Konzepte!

Listing 2 FlugReservationsServer.java

```
// Skizze eines Flugreservations-Servers mit CORBA Anbindung
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
// weitere Imports
import ...;

class AirlineServer {
      static int servicePort = 4491;

      public static void main(String arg[]) {
            KundenRef               kunde = null;
            FlugRef                 flug  = null;
            KundenReservationsRef   resv = null;

            // Usage prüfen
            if (arg.length <=0) {
                  System.err.println("Usage:");
                  System.err.println("java AirlineServer
                                    <hostname der Datenbank>");
                  System.exit(1);
            }

            // DB
            Database db = null;
            try {
                  // Datenbank Instanz
                  db = new Database(arg[0]);
            } catch (Exception e) {
                  System.err.println([Datenbank] Error : DB Instanz");
                  System.exit(1);
            }

      //kreiere eine Implementation der Objekte und publiziere sie

            try {
                  kunde =
                        KundenSkeleton.kreiereRef(new KundenImpl(db) );
                  flug =
                        FlugSkeleton.kreiereRef(new FlugImpl(db) );
                  resv =
                        KundenReservationSkeleton.kreiereRef(
                              new KundenReservationImpl(db) );
            } catch (Exception e) {
```

[15] die corba Exception

```
                System.err.println("[Implementation der Objekte]
                                        Exception");
          }

          // Publish :extern bekanntmachen
          try {
                // binde jedes Objekt an einen Service Namen
                // kreiere und initialisiere den ORB
                ORB orb = ORB.init(servicePort, null);
                // registriere die Referenzen beim ORB
                orb.connect(kunde);
                orb.connect(flug);
                orb.connect(resv);

                // bestimme die Wurzel des Naming Context
                org.omg.CORBA.Object objRef =
                       orb.resolve_initial_references("NameService");
                NamingContext ncRef = NamingContextHelper.narrow(objRef);

                // binde die Objekt Referenz im Naming Context
                NameComponent nc = new NameComponent("FlugKunde", "");
                NameComponent nc = new NameComponent("Flug", "");
                NameComponent nc = new NameComponent("Reservation", "");

                NameComponent path[] = {nc};
                ncRef.rebind(path, kunde);
                ncRef.rebind(path, flug);
                ncRef.rebind(path, resv);

                // warte auf Kunden;
                // sync ist ein dummy Objekt zur Synchronisation
                java.lang.Object sync = new java.lang.Object();
                synchronized (sync) {
                      sync.wait();
                }
          } catch (Exception e) {
                System.err.println("[FlugReservationsServer]ERROR:" + e);
                e.printStackTrace(System.out);
          }
    }
}
```

Selbsttestaufgabe 5

Noch eine triviale Frage: wo, in welchem Block, wird die Datenbank initialisiert, an das Programm angebunden?[16]

[16] Block 2 (Database ...)

1.5.21. Abschluss Quiz

In diesem Quiz werden einfache Fragen zu IDL und der ORB Architektur gestellt und auch gleich beantwortet. Die Idee de Quiz ist es, Ihnen eine Möglichkeit zu geben, die Konzepte noch einmal aufzufrischen.

1. Welche der folgenden Aussagen beschreibt den ORB am besten?
   a) Der ORB stellt eine Kommunikations- Infrastruktur zur Verfügung, mit deren Hilfe Objekte transparent Methoden anderer Objekte in einer verteilten Umgebung aufrufen können.

      Diese Antwort ist partiell richtig.

   b) Der ORB stellt eine Basis dar, um verteilte objektorientierte Applikationen zu entwickeln und eine Interoperabilität zwischen Applikationen in heterogenen Umgebungen zu erreichen.

      Diese Antwort ist partiell richtig.

   c) Der ORB kann als Daemon Prozess, Bibliothek oder als Kombination dieser beiden Konstrukte implementiert werden.

      Diese Antwort ist partiell richtig.

   d) Die obigen Antworten sind alle korrekt

      Ja das stimmt!

2. Welcher Mechanismus für den Aufruf von Objektmethoden bietet die grösste Flexibilität in einer sich dauernd ändernden Umgebung?

1. statische Aufrufe

   Denken Sie nochmals nach: statische Aufrufe / static Invocations sind limitiert auf Interfaces, welche zum Zeitpunkt der Compilation von Client und Server genau bekannt sind

2. dynamische Aufrufe

   Genau: die dynamic invocation gestattet es dem Kunden die Details des Interfaces zur Laufzeit durch Abfrage des ORBs zu bestimmen

3. ein Interface Repository

   Denken Sie nochmals darüber nach: ein Interface Repository ist eine wichtige Komponenten, ein Facility, mit dem Dynamic Invocation erst möglich wird.

4. Welche der folgenden unterstützt Grundfunktionen, um verteilte Applikationen zu benutzen und zu implementieren, welche unabhängig vom Anwendungsgebiet sind?
   a) CORBAServices

      Korrekt!

   b) CORBAFacilities

      Sind Sie sicher?
      CORBAFacilities sind eine Sammlung von gemeinsam genutzten Services auf einem Architekturlevel, der oberhalb den CORBAServices liegt.

   c) Objekt Adapter

      Sind Sie sicher?
      Objekt Adapter gestatten es den Objekten mit inkompatiblen Interfaces miteinander zu kommunizieren. CORBA definiert den Objekt Adapter als eine ORB Komponente, welche Objektreferenzen, Objektaktivierung und zustandsbezogene Services für ein Implementationsobjekt zur Verfügung stellt.

5. Java IDL idlj generiert Stubs und Skeleton Programmskelette aus einer IDL Beschreibung mit Hilfe von idlgen (alt) oder idlj (neu ab Java 2).
   a) trifft zu

      korrekt
      Der IDL Compiler idlj (oder idlgen) generiert Stubs und Skeletons, mit deren Hilfe gemeinsame Funktionlitäten eines beliebigen Transportsystems implementiert werden können.

   b) trifft nicht zu

      warum nicht?

1.5.22. Zusammenfassung Java IDL

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- zu erklären, wie die OMG und CORBA zusammenhängen
- die Funktionsweise eines ORBs zu erläutern und seine Dienste zu kennen
- Stubs und Skeletons zu generieren, mit Hilfe des idlj Compilers.
- Client und Server Implementationen zu einer in (Java) IDL spezifizierten Schnittstelle zu entwickeln.
- die Abbildungsregeln von IDL in Java zu kennen.

Den Punkt "Client und Server Implementationen" werden wir noch vertiefen (müssen).

1.6. Zusammenfassung Java in Verteilten Systemen

Nach dem Durcharbeiten dieser Kursunterlagen sollten Sie in der Lage sein

- unterschiedliche Technologien für die Programmierung verteilter Systeme in Java zu kennen und zu vergleichen
- einfache Datenbank-Anwendungen zu schreiben
- Remote Methode Invocation Applikationen zu schreiben und Daten mittels Objektserialisierung langfristig zu speichern.
- einfache Java IDL Applikatione zu schreiben