Übungen zu Kapitel 23

23.1 DNS-Dienst

Verifizieren Sie, dass man mit der Klasse `InetAdress` auch den zu einer IP-Adresse gehörenden Host-Namen ermitteln kann, indem Sie der Methode `getByName()` die numerische Repräsentation einer IP-Adresse als `String` übergeben.

23.2 Local-Host

Verwenden Sie die statische Methode `getLocalHost()`, um den Host-Namen und die IP-Adresse des eigenen Rechners zu ermitteln. Was passiert, wenn man den Zugang zum Internet abschaltet?

23.3 Object-Channel über einem Socket

Bei einer auf Java-Sockets basierenden Client-Server Anwendung laufen nach dem Akzeptieren einer Client-Anfrage durch den Server auf beiden Seiten im Prinzip die gleichen Mechanismen ab: Über einen Socket werden Aus- und Eingabestrom erzeugt, die zum Senden bzw. Empfangen von Daten dienen und am Ende geschlossen werden. Die Details über die Ströme sind aus Sicht der beteiligten Klassen kaum von Interesse, insbesondere dann nicht, wenn - wie in unserem Beispiel des Echo-Servers - nur Objekte verschickt werden. Vervollständigen Sie die Klasse

```
public class ObjectChannel {
   private ObjectOutputStream out;
   private ObjectInputStream in;
   public ObjectChannel( Socket s ) throws IOException {
      out = new ObjectOutputStream( s.getOutputStream() );
      in = new ObjectInputStream( s.getInputStream() );
   }

   public void writeObject( Object o ) throws IOException {
      // "o" nach "out" schreiben und flush()
   }

   public Object readObject( )
               throws IOException, ClassNotFoundException {
      // aus "in" lesen
   }

   public void close() throws IOException {
      // Ströme schließen
   }
}
```

und modifizieren Sie `ObjectEchoClient` und `ObjectEchoWorker`, so dass in jeder Klasse anstelle des Ein- und Ausgabestroms nur eine Instanz von `ObjectChannel` verwendet wird, die für das Senden und Empfangen von Daten verantwortlich ist. Vergessen Sie nicht, den Channel am Ende zu schließen.

23.4 Wiederholtes Senden und Empfangen an den Echo-Server

Erweitern Sie das Protokoll zwischen Client und Worker, so dass der Worker nicht schon nach dem ersten Senden und Empfangen eines Objektes terminiert, sondern erst dann, wenn der Client keine Objekte mehr senden möchte und dies mitteilt. Erstellen Sie hierfür eine leere Klasse `ClientTermSignal` (ohne Attribute und Methoden), von der eine zum Server geschickte Instanz das Ende signalisiert. Alternativ kann man natürlich auch `String`-Instanzen wie etwa "QUIT" verwenden.

23.5 Viele Clients am Echo-Server

Das Starten mehrerer Clients innerhalb eines engen Zeitraums lässt sich am besten dadurch bewerkstelligen, dass man die Clients selbst als Threads implementiert und innerhalb einer Java-VM gleich mehrere davon startet. Erweitern Sie dahingehend die Object-Echo Anwendung. Verifizieren Sie durch geeignete Konsolenausgaben, dass serverseitig für jeden Client ein Thread gestartet wird, in dem sich eine Instanz von `ObjectEchoWorker` um ihn kümmert.

23.6 Beenden des Echo-Servers

Da der Echo-Server innerhalb einer Endlos-Schleife auf Client-Anfragen wartet, kann er in der ursprünglichen Version nicht ordentlich terminiert werden. Implementieren Sie den Server als Thread und stellen Sie im Server eine Methode

```
public void terminate()
```

bereit, mit der sich der Server selbst beendet. Der Worker erhält ein weiteres Attribut

```
private ObjectEchoServer t
```

so dass man den gestarteten Server-Thread via Konstruktor beim Worker registrieren kann. Analog zur Übung 23.3 kann man von einem Client eine Instanz einer speziellen Klasse (z.B. `ServerTermSignal`) an den Server senden, um beim Worker die Beendigung des Servers durch den Aufruf `t.terminate()` zu veranlassen.

Wundern Sie sich nicht, wenn die Methode

```
public void terminate(){
   running = false;
}
```

nicht das hält, was sie auf den ersten Blick verspricht.

Tipp: Erzeugen Sie in `terminate()` ein Dummy-Socket, das sofort geschlossen wird.

23.7 Client-Server Szenario "Rechenserver" (Teil 1)

Implementieren Sie das dargestellte Szenario des Rechenservers durch das Interface und die Klassen

```
public class IExec             // Schnittstelle
public class ExecuteServer     // identisch mit ObjectEchoServer
public class ExecuteWorker     // ObjectEchoWorker modifiziert
public class ExecuteClient     // ObjectEchoClient modifiziert
```

Die Implementierungen für die Berechnungen können als eigenständige Klasse oder als `private static` innerhalb von `ExecuteClient` erstellt werden. Ergänzen Sie die folgende Klasse zur näherungsweisen Berechnung der Eulerschen Zahl ***e***.

```
private static class Euler implements IExec { // Berechnung der Eulerschen Zahl
  private int n = 0;
  public Euler( int n ) {
    this.n = n;
  }
  public String exec( ) {   //...
  }
  public String[] getArgs() {  return null; }
}
```

wobei die Eulersche Zahl durch folgende Rekursion näherungsweise berechnet werden kann:

E(0) = 1

E(n+1) = E(n) + 1/n!

In der Anwendung soll der Client für einige ***n*** die Berechnung von ***E(n)*** am Server anfordern

```
out.writeObjekt( new Euler( n ) );
```

und anschließend ausgeben. Vergleichen Sie die Werte mit der Konstanten `Math.E`.

Falls sich die Klassen von Client und Server in unterschiedlichen Paketen befinden, ist Übung 23.8 zu beachten!

23.8 Client-Server Szenario "Rechenserver" (Teil 2)

Befinden sich die Klassen des Servers und die Implementierung von `IExec` in unterschiedlichen Paketen, dann wird beim Deserialisieren in `readObject()` eine `ClassNotFoundException` ausgelöst, da die implementierende Klasse nicht bekannt ist.

Die naheliegende Lösung, die Class-Datei vor dem Start des Servers manuell in dessen Projektverzeichnis zu kopieren, ist höchst unelegant, zumal die Dynamik des Client-Server Szenarios empfindlich gestört wird – insbesondere dann, wenn sich Client und Server auf unterschiedlichen Rechnern befinden.

Ein andere Lösung besteht darin, die Class-Datei innerhalb des Kommunikationsprotokolls vom Client zum Server zu übertragen und dort nachzuladen. Zum Senden erweitern wir den Client wie folgt

```
...
Object o = new Euler( 7 );                  // zu berechnende Instanz erzeugen
sendClass( out, o.getClass().toString() ); // Class-Datei zum Server senden
out.writeObject( o );                       // Instanz zum Server senden
...
```

sowie den Server auf der Gegenseite:

```
...
receiveAndLoadClass( in );                  // Class-Datei empfangen und nachladen
IExec execInfo = (IExec)in.readObject();    // Instanz empfangen
...
```

Der Client schickt zunächst den Namen der Klasse, dann die Anzahl der zu erwartenden Bytes und schließlich den Dateiinhalt selbst (der Einfachheit wegen gehen wir davon aus, dass sich die Klasse im Default-Paket, d.h. im aktuellen Verzeichnis, befindet):

```
private static void sendClass( ObjectOutputStream out, String name )
                                                        throws IOException {
   out.writeObject(name);                    //Namen der Klasse schicken
   File f = new File(name + ".class");       //File-Object der Class-Datei
   FileInputStream fin = null;
   fin = new FileInputStream(f);
   out.writeObject( new Long(f.length()) ); //Länge der Datei schicken
   int c;
   while( (c=fin.read())!=-1 )               //Dateiinhalt schicken
      out.write(c);
   // fin schließen
}
```

Beim Lesen auf der Server-Seite werden die Informationen gemäß dem Sende-Protokoll gelesen, die Datei gespeichert und die Klasse schließlich dynamisch nachgeladen:

```
private void receiveAndLoadClass( ObjectInputStream in )
                                  throws IOException, ClassNotFoundException {
      String name = (String)in.readObject();
      System.out.println("Name: " + name);
      File f = new File(name + ".class");   // File-Objekt für die Class-Datei
      FileOutputStream fout = new FileOutputStream( f );
      long len =  ((Long)in.readObject()).longValue();
      for(long i=0; i<len;i++){             // in Datei schreiben
         fout.write(in.read());
      }
   fout.close();
   Class.forName( name );           // Nachladen der Klasse
}
```

23.9 RMI und Security-Manager

Testen Sie das RMI-Beispiel als Anwendung, bei dem sich Client und Server auf unterschiedlichen Rechnern befinden. Binden Sie durch das Code-Fragment

```
System.setSecurityManager( new RMISecurityManager() );
```

sowohl am Server wie am Client einen Security-Manager ein (direkt zu Beginn des `try`-Blocks) und definieren Sie in einer Datei **`.java.policy`** im Verzeichnis **`user.home`** (auf beiden Rechnern) die Zugriffsrechte - sehr liberal - wie folgt:

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

Das Verzeichnis bestimmt man am besten durch den Ausdruck

```
System.out.println( System.getProperty( "user.home" ) );
```

Vorsicht: Mit dem Zugriffsrecht `AllPermission` wird das Sicherheitskonzept de facto abgeschaltet. Dies sollte nur zum Testen der RMI-Anwendung geschehen und anschließend wieder zurückgenommen werden.

Alternativ kann man auch folgenden (Pseudo-) Security-Manager verwenden, der keine Berechtigungsprüfung durchführt, sondern die an ihn gerichteten Anfragen lediglich protokolliert:

```
class PseudoSecurityManager extends RMISecurityManager{
   public void checkPermission( Permission perm ){
      System.out.println("Check: " + perm );
   }
}
```

Auf die Details des Java Security Konzepts können wir nicht eingehen und verweisen an dieser Stelle auf die Dokumentation unter `http://java.sun.com/j2se/1.5.0/docs/guide/security`.