

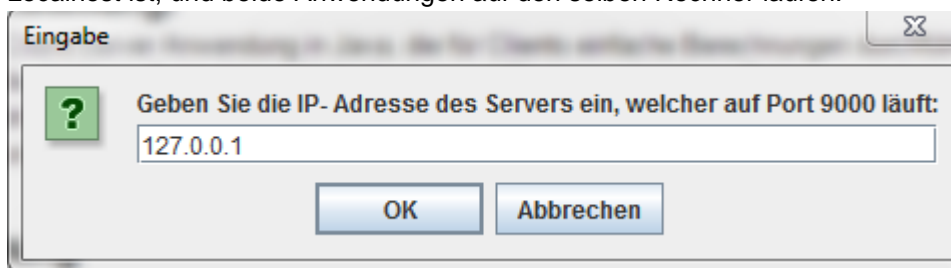
# Protokoll Aufgabe 7:

## Aufgabenstellung:

Erweitere Aufgabe 6 so, dass der Server einen sauberen Shutdown durchführt. D.h. das alle bereits gestarteten Anfragen sollten korrekt beendet, aber keine neue mehr gestartet werden. Alle auftretenden Exceptions sollten dabei korrekt behandelt werden. Verwende dabei das Interface ExecutorService.

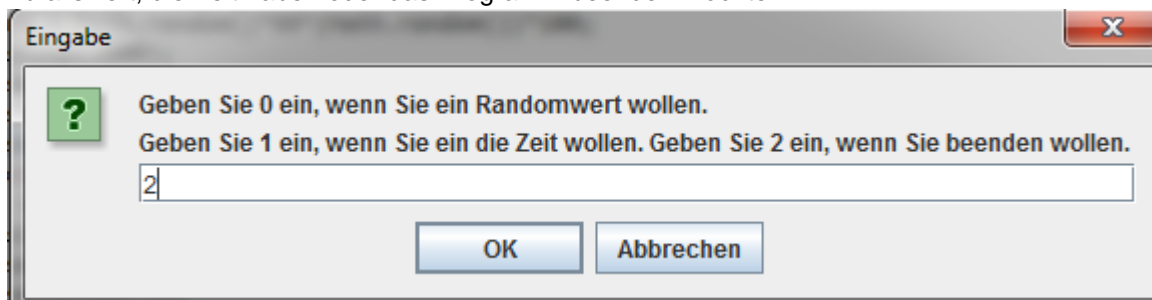
## Handhabung

Das Programm besteht aus zwei ausführbaren Dateien, eine Serverdatei und eine Client Datei. Wichtig ist, dass der Server vor dem Client ausgeführt wird, sonst kommt es zu Problemen. Sollte man es geschafft haben, die Dateien in richtiger Reihenfolge auszuführen, muss man im Client als ersten Schritt IP-Adresse eingeben. Zum testen, wird die IP-Adresse 127.0.0.1 genutzt, da dies der Localhost ist, und beide Anwendungen auf den selben Rechner laufen.

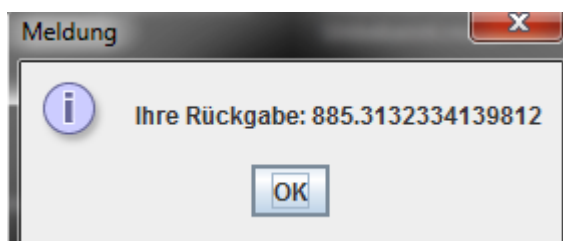


Eingabe der IP-Adresse

Nachdem man dies erledigt hat, kommt ein neues Fenster, welches Sie befragt, ob sie einen Zufallswert, die Zeit haben oder das Programm beenden möchten .



Als Ausgabe beim Zufallswert kommt folgendes zum Vorschein.



Jetzt ist aber neu, dass das Programm nicht beendet wird, sondern weiterläuft. Sollte der Benutzer das Programm beenden, muss er "2" eingeben. Dann werden auf dem Server die noch zu berechnenden dinge fertig berechnet, aber keine neuen Anfragen angenommen.

## Der Ablauf

Nachdem der Server erstellt wurde, bietet er dem Client ein `ServerSocket` an, mithilfe dessen sich der Client zum Server verbinden kann. Der Server wird über Port 9000 erreichbar sein.

```
ServerSocket listener = new ServerSocket(9000);
```

```
Socket socket = listener.accept();
```

Jetzt wartet der Server solange, bis sich ein Client verbindet. sollte sich ein Client verbinden wird dies durch `listener.accept()`; akzeptiert und zugelassen.

Im Client wird auch ein `Socket` erstellt, auch dies benutzt den Port 9000.

```
Socket s = new Socket(serverAddress, 9000);
```

Um etwas zu senden und zu empfangen, wurde jetzt eine abstrakte Klasse namens `Message` implementiert.

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public abstract class Message implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 312853181604924426L;

    public void send(ObjectOutputStream writer) {
        try {
            writer.writeObject(this);
            writer.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static Message fromStream(ObjectInputStream reader) {
        try {
            return (Message) reader.readObject();
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

In der Klasse `Message`, werden zwei Methoden implementiert, welche für uns ein Objekt versenden und empfangen. Um ein Objekt zu versenden wird die Methode `send` zur Verfügung gestellt. Um Ein Objekt zu lesen bzw. zu empfangen wird uns die Methode `from Stream` zur Verfügung gestellt.

Um jetzt ein Objekt zu versenden sind folgende Codezeilen von nöten:

```
ObjectOutputStream out = new ObjectOutputStream(s.getOutputStream());
ObjectInputStream in = new ObjectInputStream(s.getInputStream());
Authentifizierung auth = new Authentifizierung(getAuth());
auth.send(out);
```

**(BEISPIEL)**

Ein Objekt von `Authentifizierung` wird erstellt, dies muss von der Klasse `Message` erben, damit es diese nutzen kann. danach wird es mit `auth.send(out)`; gesendet. Wichtig ist, dass es Die `Authentifizierung` im Server gibt, damit man dort mit den dort drin gespeicherten Daten arbeiten kann. Um ein Objekt zu lesen ist folgende Zeile von Nöten:

```
Authentifizierung a = (Authentifizierung) Message.fromStream(in);
```

**(BEISPIEL)**

Hier wird der Datentyp, welcher vorher verschickt wurde in ein schon existierendes Objekt Authentifizierung kopiert, damit man mit dieser Klasse arbeiten kann.

Um die Frage der Aufgabenstellung beantworten zu können, wurden zwei Versionen dieser Anwendung erstellt. Eine mit Threats und eine ohne. Das Ergebnis war, dass wenn man mehrer Aufgaben auf einmal in der Anwendung mit den Threats berechnen ließ, diese viel schneller war, als diese ohne Threats. Diese Ergebniss kommt zustande, da Threads parallel arbeiten können, und nicht sequenziell.

Um das Programm korrekt zu beenden benötigen wir den ExecutorService.

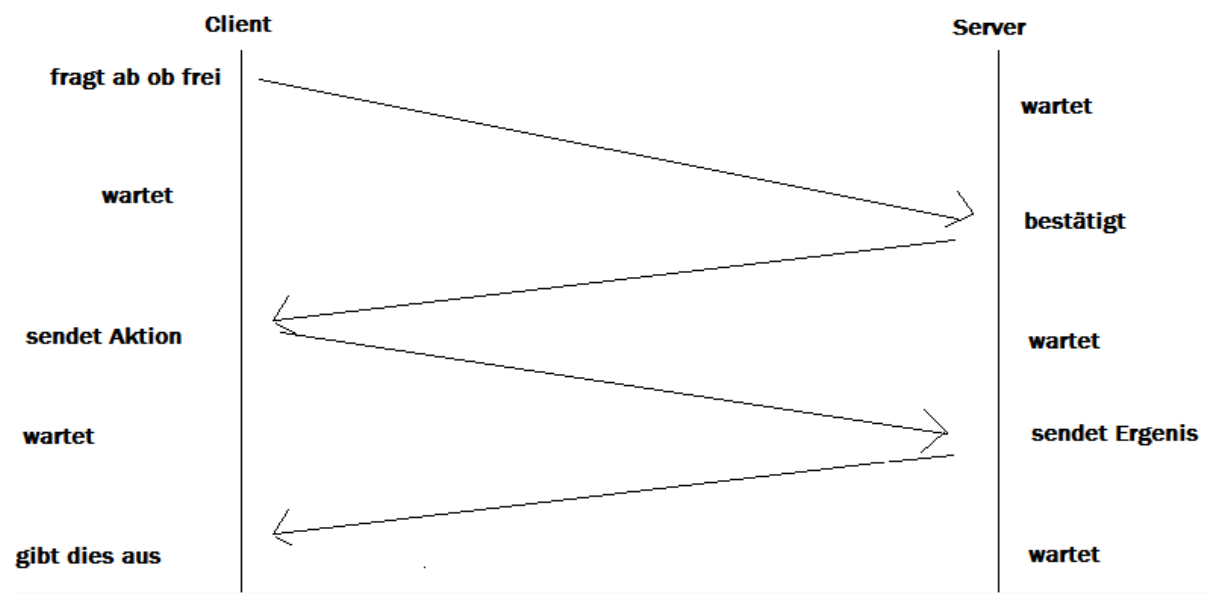
```
server.Shutdown();
```

Shutdown wird aufgerufen

```
setWeiter(false);  
executor.shutdown();  
socket.close();  
System.out.println("Server beendet");
```

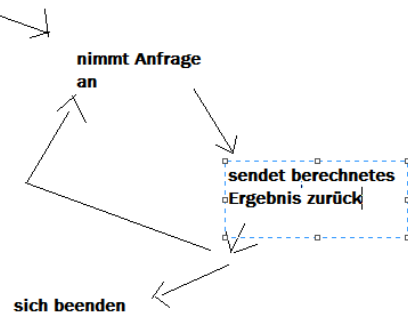
Server wird korrekt beendet.

## Grafiken:



## Server:

warte auf  
Clients



## Client:

IP Adresse  
eingeben

