Übungen zu Verteilte Systeme : Aufgabe zu Java-RMI (1)

- **Aufgabenstellung :**

  Es ist ein über RMI aktivierbarer **generischer Problemlösungs-Server** in Java zu entwickeln.
  Kern des Servers ist ein mittels RMI ansprechbares entferntes Objekt, das prinzipiell beliebige Aufgaben (= zu lösende Probleme) von Clients entgegennimmt, diese bearbeitet und das erzeugte Ergebnis zurückliefert.

  Die jeweils zu bearbeitende Aufgabe wird durch ein Objekt beschrieben (Aufgabenobjekt) , das der Problemlösungsfunktion des Server-Objekts vom Client als Parameter übergeben werden muss.
  Die Problemlösungsfunktion des Server-Objekts ruft ihrerseits die im Aufgabenobjekt implementierte eigentliche Bearbeitungsfunktion auf.
  Da bei RMI bei Bedarf auch der Code von (Parameter- und Rückgabe-) Objekten übertragen wird, muss die eigentliche Bearbeitungsfunktion dem Server a priori nicht bekannt sein.
  Der Server ist somit in der Lage beliebige Aufgaben zu bearbeiten, die zum Zeitpunkt seines Starts (und erst recht zum Zeitpunkt seiner Erstellung) noch nicht einmal definiert sein müssen.
  Einzige Voraussetzung ist, dass die Aufgabenobjekte ein Interface implementieren, über das ihre eigentliche Bearbeitungsfunktion aufgerufen werden kann.

  Clients können somit die Bearbeitung von Aufgaben, die ihre eigenen Fähigkeiten übersteigen (Zeitbedarf, Ressourcen), auf den Problemlösungs-Server auslagern.

  Der Problemlösungs-Server soll beispielhaft zur Ermittlung der Zahl `PI` mit einer angebbaren Genauigkeit eingesetzt werden. Hierfür ist ein geeigneter Client zu entwickeln. Zum Abschluss soll ein weiterer Client erstellt werden, der basierend auf den bereits erstellten Client für eine andere mathematische Problemlösung geeignet ist.

- **Hinweise zur Lösung (Interfaces und Server-Seite) :**

  ◇ Die Schnittstelle zum Aufruf der Server-Funktionalität wird durch das Remote-Interface **ProblemSolver** beschrieben.
  Es deklariert– als einzige Komponente – die vom Client aufrufbare Problemlösungsfunktion

  ```
  Object solveProblem(ProblemTask t);
  ```

  Da in Java alle Klassen von der Klasse `Object` – direkt oder indirekt – abgeleitet sind, kann eine konkrete Implementierung dieser Funktion ein beliebiges Ergebnis durch ein Objekt beliebigen Typs zurückgeben.

  ◇ Der Problemlösungsfunktion `solveProblem()` wird vom Client das Aufgabenobjekt als Parameter vom Typ **`ProblemTask`** übergeben.
  `ProblemTask` ist ein – non-remote – Interface, das – als einzige Komponente – die Bearbeitungsfunktion

  ```
  Object executeTask();
  ```

  deklariert.
  Das tatsächliche Aufgabenobjekt ist Instanz einer Klasse, das dieses Interface `ProblemTask` implementiert.
  Jede derartige Klasse repräsentiert eine zu bearbeitende konkrete Aufgabe. Die Bearbeitung der Aufgabe wird von der von der Klasse definierten Funktion `executeTask()` realisiert.
  Da das Aufgabenobjekt – als Kopie – vom Client zum Server übertragen wird, muss das Interface `ProblemTask` vom Interface `Serializable` abgeleitet sein.

  ◇ Beide Interfaces – `ProblemSolver` und `ProblemTask` – werden sowohl auf der Client- als auch auf der Server-Seite benötigt.

  ◇ Auf der Server-Seite wird das Interface `ProblemSolver` durch die Klasse **`ProblemSolverImpl`** implementiert.
  Diese Klasse definiert die Problemlösungsfunktion `solveProblem()`, die – als einzige wesentliche Aktivität – die Funktion `executeTask()` für das ihr als Parameter übergebene Aufgabenobjekt aufruft.
  Sie gibt deren Rückgabewert (Objekt beliebigen Typs) als eigenen Rückgabewert zurück

  Die Klasse `ProblemSolverImpl` definiert auch – in ihrer statischen `main()`–Methode – die Start-Funktionalität des Servers. Diese umfasst :
  - Installation eines RMISecurityManagers, falls noch kein SecurityManager in der JVM installiert ist.
    Hinweis: Er wird benötigt, sobald sich Client und Server auf unterschiedlichen Rechnern befinden und er interpretiert eine Richtliniendatei (`.java.policy`), die sich im Heimatverzeichnis des Benutzers befindet.
  - Erzeugung eines Objekts der – eigenen – Klasse `ProblemSolverImpl.`

Übungen zu Verteilte Systeme : Aufgabe zu Java-RMI (2)

- Registrierung dieses Objekts bei der RMI-Registry unter dem Dienstnamen "`ProblemSolver`".
- Ausgabe des Textes "`ProblemSolver bound in registry`" an der Konsole (= Standardausgabe).

- **Hinweise zur Lösung (Client-Seite) :**

◇ Die zu bearbeitende Aufgabe – Ermittlung der Zahl `PI` mit einer angebbaren Genauigkeit – wird durch die Klasse **`PiTask`** definiert. Diese Klasse implementiert das Interface `ProblemTask`.
Sie besitzt die folgenden Komponenten :

▻ Eine **Datenkomponente** vom Typ **`double`**, die die gewünschte Genauigkeit aufnimmt.

▻ Einen **Konstruktor** mit einem Parameter vom Typ `double`.
Der Konstruktor initialisiert die Genauigkeits-Datenkomponente mit diesem Parameter.

▻ Die die Bearbeitungsfunktionalität realisierende Memberfunktion

```
Object executeTask();
```

◇ Ein – auf der Potenzreihenentwicklung für den `arctan` beruhender – Algorithmus zur iterativen Ermittlung der Zahl `PI` mit der Genauigkeit `genauigkeit` kann wie folgt beschrieben werden :

```
pi_viertel = 1.0;
inc = 1.0;
inv_inc = 1/inc;
fak = 1;
while (4*inc > genauigkeit)
{  inv_inc += 2;
   inc = 1/inv_inc;
   fak = -fak;
   pi_viertel += fak*inc;
}
pi = 4*pi_viertel;
```

◇ Der Rückgabewert von `executeTask()` muss ein Objekt (und kein Wert eines elementaren Datentyps) sein. Der ermittelte Wert für `PI (double-Wert pi)` muss daher in einem Objekt der Wrapper-Klasse `Double` gekapselt werden. Dies kann erfolgen mit : `new Double(pi);`

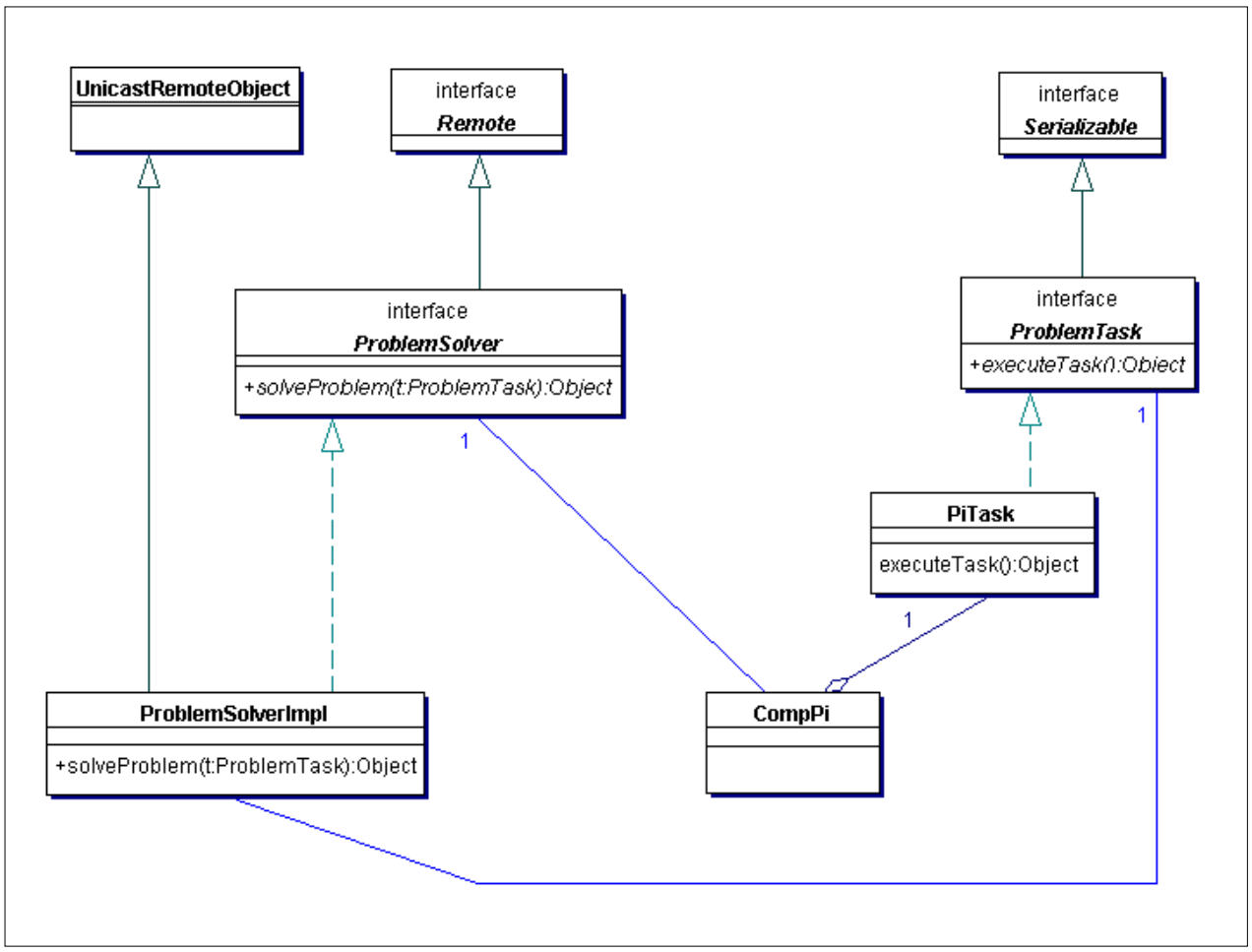
◇ Der Client wird durch die Klasse `CompPi` implementiert.
Diese Klasse besitzt nur eine statische `main()`-Methode, in der die gesamte Client-Funktionalität zusammengefaßt ist. Diese Funktionalität besteht aus :

- Installation eines RMISecurityManagers, falls noch kein SecurityManager in der JVM installiert ist.
- Überprüfung auf die richtige Anzahl von Kommandozeilenparametern.
  Das Client-Programm muss mit zwei Kommandozeilenparametern aufgerufen werden :
  Server (Name oder IP-Adresse) und gewünschte Genauigkeit.
  Bei einer falschen Anzahl von Kommandozeilenparametern ist ein Hinweis auf das richtige Aufrufformat an der Konsole (=Standardfehlerausgabe) auszugeben und das Programm zu beenden.
  Bei einer richtigen Anzahl von Kommandozeilenparametern sind die folgenden weiteren Aktivitäten auszuführen :
- Erfragen einer lokalen Referenz auf das entfernte RMI-Server-Objekt bei der RMI-Registry
- Ermittlung der internen Darstellung des Zahlenwerts der Genauigkeit (Typ `double`) aus dem entsprechenden Kommandozeilenparameter :
  ```
  double prec = Double.parseDouble(args[1]);
  ```
- Erzeugung eines Objekts der Klasse `PiTask` (Aufgabenobjekt).
- Aufruf der Methode `solveProblem()` für die erhaltene Referenz auf das RMI-Server-Objekt unter Übergabe des erzeugten Aufgabenobjekts als Parameter.
  Der Rückgabewert dieses Methodenaufrufs ist ein Objekt der Wrapper-Klasse `Double`, das den ermittelten Wert für `PI` (Typ `double`) kapselt.
- Entkapselung des ermittelten Werts `res` für `PI` aus dem Rückgabeobjekt `resObj` (der Klasse `Double`) :
  ```
  double res = resObj.doubleValue();
  ```
- Ausgabe des Ergebnisses an der Konsole (= Standardausgabe), begrenzt auf die genaue Stellenanzahl `digits`.
  Diese läßt sich aus der Genauigkeit `prec` wie folgt ermitteln :
  ```
  int digits = -(int)Math.round(Math.log(prec)/Math.log(10.0))-1;
  ```
- Die Ergebnisausgabe kann dann erfolgen mittels :
  ```
  System.out.println((new BigDecimal(res)).setScale(digits,
       BigDecimal.ROUND_HALF_UP));
  ```

Übungen zu Verteilte Systeme : Aufgabe zu Java-RMI (3)

- **Klassendiagramm zur Lösung :**

  Die statischen Beziehungen der für die Lösung benötigten Klassen (Server- und Client-Seite) können durch das folgende Klassendiagramm beschrieben werden :



- **Anmerkungen zur Entwicklungsumgebung:**

  ◇ Die Entwicklung kann sowohl unter **Windows** oder unter **Linux** statt. Diese Auswahl darf der Student/die Studentin selbst treffen, empfohlen wird die Entwicklung unter **Windows**, da sich die nachfolgende Beschreibung auf Windows-Verzeichnisse bezieht.
  Als Entwicklungswerkzeuge stehen **`Eclipse` oder** ein Editor **`SciTe/GEdit`** und die Tools des Java Development Kits (**JDK**) 1.x zur Verfügung.

  ◇ Server, Client und **`rmiregistry`** werden **immer** über die Eingabeaufforderung gestartet.

  ◇ **Bei der manuellen Erstellung gilt folgendes:**
  Legen Sie im Verzeichnis **`L:\Projects`** die Verzeichnisse **`Server`** und **`Client`** an.
  Das Directory **`L:\Projects\Server`** ist das **Arbeitsverzeichnis** für die Entwicklung und Start des **Servers**.
  Das Directory **`L:\Projects\Client`** ist das **Arbeitsverzeichnis** für die Entwicklung und Start des **Clients**.
  Legen Sie in beiden Arbeitsverzeichnisse jeweils die Directory-Struktur **`vs\rmiprobsolv`** an.

  Alle vom Editor/IDE und den JDK-Tools erzeugten Dateien für den Server bzw. den Client sind in dem jeweiligen Verzeichnis **`rmiprobsolv`** abzulegen.

  Die Übersetzungstools des JDK müssen über Kommandozeilenaufrufe aus der Eingabeaufforderung gestartet werden.

Übungen zu Verteilte Systeme : Aufgabe zu Java-RMI (4)

- **Anmerkungen zur Realisierung :**
  - ◇ **Allgemeines**
    - ▻ **Jedes** Interface und jede Klasse muss in einer **eigenen Datei** definiert werden.
      Jede Quell-Datei muss den jeweiligen Klassennamen als Hauptnamen und die Extension **`.java`** tragen.
    - ▻ Alle Interfaces und Klassen sollen zum Package **`vs.rmiprobsolv`** gehören.
      → jede Java-Quelldatei muss die folgende Package-Deklaration enthalten :
      `package vs.rmiprobsolv;`
    - ▻ Zur Erzeugung des Bytecodes (Erweiterung **`.class`**) ist sowohl die Verwendung der IDE (Eclipse) als auch der manuelle Erstellvorgang mit dem Java-Compiler **`javac`** über die Eingabeaufforderung erlaubt.

      Bei der Verwendung der IDE sind zwei Projekte (**Server** und **Client**) im Workspace unter `L:\Projects` zu erstellen.
      Die Quelldateien befinden sich dann im Packageverzeichnis **`vs\rmiprobsolv`** des jeweiligen Projektverzeichnisses, also unterhalb von `L:\Projects\Server\src` bzw `L:\Projects\Client\src`, der aufzurufende Bytecode befindet sich analog im Ausgabeverzeichnis unter `L:\Projects\Server\bin` bzw. `L:\Projects\Client\bin`.
      In **`Eclipse`** werden in der Standardkonfiguration die Quelldateien des Projekts beim Abspeichern automatisch in Bytecode (**`.class`**-Datei) übersetzt.
      Bei manuellem Erstellvorgang gilt folgendes zu beachten:
      Ausgehend vom jeweiligen Projektverzeichnis (`L:\Projects\Server` bzw `L:\Projects\Client`) ist die zu übersetzende Quelldatei durch eine relative Pfadangabe zu referieren,
      z.B. **`javac vs\rmiprobsolv\ProblemSolverImpl.java`**
      In diesem Fall wird der Bytecode (**`.class`**-Datei) im entsprechenden Quellcodeverzeichnis erstellt, d. h. Quellcode- und Ausgabeverzeichnis sind gleich!
      (Also `L:\Projects\Server\vs\rmiprobsolv` bzw. `L:\Projects\Client\vs\rmiprobsolv`)

  - ◇ **Erzeugung der Interfaces**
    - ▻ Die beiden Interfaces **`ProblemTask`** und **`ProblemSolver`** werden sowohl vom Client als auch vom Server benötigt.
    - ▻ Erzeugen Sie die Quellcode-Dateien der Interfaces für **beide** Projekte (Server und Client). Beginnen Sie mit der Codierung für den Server.
    - ▻ Übersetzen Sie die Quellcode-Dateien (s. oben, Allgemeines)
    - ▻ Nach erfolgreichem Übersetzen sind die erzeugten Quellen in das Client-Quellcodeverzeichnis zu kopieren und der Übersetzungsvorgang für den Client zu wiederholen.

.

  - ◇ **Erzeugung und Start des Servers**
    - ▻ Erzeugen Sie die Quellcode-Datei für die Server-Klasse **`ProblemSolverImpl`**
    - ▻ Übersetzen Sie die Quellcode-Datei (s. oben, Allgemeines)
    - ▻ Öffnen Sie eine Eingabeaufforderung und wechseln Sie in das Ausgabeverzeichnis des Serverprojekts.
      IDE: **`L:\Projects\Server\bin`**
      manuelle Erstellung: **`L:\Projects\Server`**
    - ▻ Erzeugen Sie Stub-Klasse für die Server-Implementierung mit dem RMI-Compiler **`rmic`**.
      Die – als Java-Bytecode-Datei vorliegende – Server-Klasse muss mit ihrem vollqualifizierten Package-Namen angegeben werden.
      `rmic vs.rmiprobsolv.ProblemSolverImpl`
    - ▻ Kopieren Sie die erzeugte Datei **`ProblemSolverImpl_stub.class`** zusammen mit den Java-Bytecode-Dateien für die Interfaces **`ProblemSolver.class`** und **`ProblemTask.class`** in das Packageverzeichnis unterhalb des Webspace-Verzeichnisses **`L:\public_html\ServerWebspace`**.
      (Es ist eine dem Package-Namen **`vs.rmiprobsolv`** entsprechende Verzeichnisstruktur anzulegen.)
      Damit liegen sie tatsächlich auf dem Server **`ti1`**, sind öffentlich erreichbar und können von dort (durch den Client) heruntergeladen werden.

Übungen zu Verteilte Systeme : Aufgabe zu Java-RMI (5)

- **Anmerkungen zur Realisierung, Forts. :**
  - ▻ Starten Sie die RMI-Registry auf dem Standardport (1099). in einem Konsolen-Fenster, in dem die Environment-Variable **CLASSPATH nicht** gesetzt ist :

    set CLASSPATH=

    Starten der RMI-Registry:

    rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false

  - ▻ Starten Sie in einem anderen Konsolenfenster (CLASSPATH muss wie oben gesetzt sein) aus dem Server-Ausgabeverzeichnis **L:\Projects\Server\bin** bzw. **L:\Projects\Server** den Server. Geben Sie beim Aufruf der JVM die von der RMI-Registry zu verwendende Codebase (= Server und Directory), die die Stub-Klasse enthält, an:

    java -Djava.rmi.server.codebase=http://ti1/~<user>/ServerWebspace/
    -Djava.rmi.server.useCodebaseOnly=false vs.rmiprobsolv.ProblemSolverImpl

    Hinweise:
    - <user> entspricht Ihrer Praktikumskennung (z. B. vs53).
    - Mit den Schaltern **-D** im Aufruf, werden Defines gesetzt, d. h. es findet **keine** Überprüfung statt, ob die Angabe danach (**java.rmi.server.codebase** bzw. **java.rmi.server.useCodebaseOnly**) korrekt geschrieben wurde.
    - Die Angabe der **java.rmi.server.codebase** muss mit einen **/** enden.

  - ◇ **Erzeugung und Start des Clients**:

    - ▻ Erzeugen Sie die Quellcode-Dateien für die Client-Klassen **PiTask** und **CompPi**.
    - ▻ Übersetzen Sie die Quellcode-Datei (s. oben, Allgemeines)
    - ▻ Kopieren Sie die Java-Bytecode-Datei für die Klasse des Aufgabenobjekts **PiTask.class** und die Java-Bytecode-Dateien der Interfaceklassen in das Package-Verzeichnis unterhalb von **L:\public_html\ClientWebspace**.
      (Es ist eine dem Package-Namen **vs.rmiprobsolv** entsprechende Verzeichnisstruktur anzulegen.)
      Damit liegen sie tatsächlich auf dem Server **ti1**, sind öffentlich erreichbar und können von dort (durch den Server) heruntergeladen werden.
    - ▻ Öffnen Sie eine Eingabeaufforderung und wechseln Sie in das Ausgabeverzeichnis des Serverprojekts.
      IDE: **L:\Projects\Client\bin**
      manuelle Erstellung: **L:\Projects\Client**
    - ▻ Starten Sie in der Eingabeaufforderung den Client.
      Geben Sie beim Aufruf der JVM die Codebase, in der sich die Bytecode-Datei **PiTask.class** befindet, an:

      java -Djava.rmi.server.codebase=http://ti1/~<user>/ClientWebspace/
      -Djava.rmi.server.useCodebaseOnly=false
      vs.rmiprobsolv.CompPi localhost <genauigkeit>

      Hinweise:
      - <user> entspricht Ihrer Praktikumskennung (z. B. vs53).
      - Mit den Schaltern **-D** im Aufruf, werden Defines gesetzt, d. h. es findet **keine** Überprüfung statt, ob die Angabe danach (**java.rmi.server.codebase** bzw. **java.rmi.server.useCodebaseOnly**) korrekt geschrieben wurde.
      - Die Angabe der **java.rmi.server.codebase** muss mit einen **/** enden.

    - ▻ Nun soll ihr Client mit dem Server Ihres Betreuers interagieren.
      Die für den Client-Aufruf benötigte **IP-Adresse des Servers** (1. Kommandozeilenparameter**)** wird Ihnen vom Betreuer zu Beginn des Praktikums mitgeteilt.

Übungen zu Verteilte Systeme : Aufgabe zu Java-RMI (6)

◇ **Ergänzungen (ab Sommersemester 2015)** :
Individualanteil:

Jeder Student/jede Studentin hat eine Individualteil zu leisten.
Dieser besteht darin, nachzuweisen, dass es sich um einen generischen Server handelt, der seinen Berechnungsalgorithmus durch den Client erhält.

▻ Erstellen Sie eine weitere Klasse **`SpecialTask`** die ebenfalls **`ProblemTask`** implementiert und **eine** individuelle Aufgabe aus dem unten aufgeführten Aufgabenkatalog bewältigt.
Welche Aufgabe Sie zu bewältigen haben wird vom Dozenten vorab festgelegt.
- Sollten Sie keine Zuteilung erhalten haben, fragen Sie rechtzeitig Ihren Dozenten.
- Sollten Sie eine eigene Idee implementieren wollen, dann besprechen Sie das vorab mit Ihrem Dozenten.

▻ Ändern Sie **`CompPi`** nun so ab, dass der implementierte Algorithmus der Klasse **`SpecialTask`** verwendet wird. Wiederholen Sie nun die Punkte in **Erzeugung und Start des Clients** mit der neuen Client-Klasse **`SpecialTask`**.

Dem Konstruktor wird - wie in **`PiTask`** - die Genauigkeit übergeben.

Aufgabenkatalog:

Aufgabe	Ziel	Hinweise
Monte-Carlo	Ermittlung der Zahl **`Pi`**	Anstelle der arctan-Iteration soll Pi mithilfe des Monte-Carlo-Algorithmus ermittelt werden. **Achtung:** Es kann sein, dass Pi trotz übergebener Genauigkeit nicht stimmt, da es sich um ein statistisches Verfahren handelt. Hinweise zum Algorithmus finden Sie unter: **`http://www.zum.de/Faecher/Inf/RP/Java/java_1.htm`** oder **`http://pigen.de.pn/`**
Cusanus	Ermittlung der Zahl **`Pi`**	Anstelle der arctan-Iteration soll Pi mithilfe der Methode von Cusanus ermittelt werden. Hinweise zum Algorithmus finden Sie unter: **`http://www.herder-oberschule.de/madincea/aufg0009/cusanus.pdf`**
Euler	Ermittlung der eulerschen Zahl **`e`**	Mithilfe einer unendlichen Reihe soll die Eulersche Zahl ermittelt werden mit definierter Genauigkeit ermittelt werden. Hinweise zum Algorithmus und der zu verwendenden Reihe finden Sie unter: **`http://de.wikipedia.org/wiki/Eulersche_Zahl`**
Heron-Verfahren	Ermittlung des Ergebnisses für $\sqrt{2}$ (ohne Verwendung der Funktionen **`Math.sqrt()`** od. **`Math.pow()`**)	Mithilfe des Verfahrens von Heron soll die Wurzel aus 2 ermittelt werden. Hinweise zum Algorithmus und der zu verwendenden Reihe finden Sie unter: **`http://www.math.uni-leipzig.de/pool/tuts/Delphi/dpue080.htm`** **`http://de.wikipedia.org/wiki/Heron-Verfahren`**