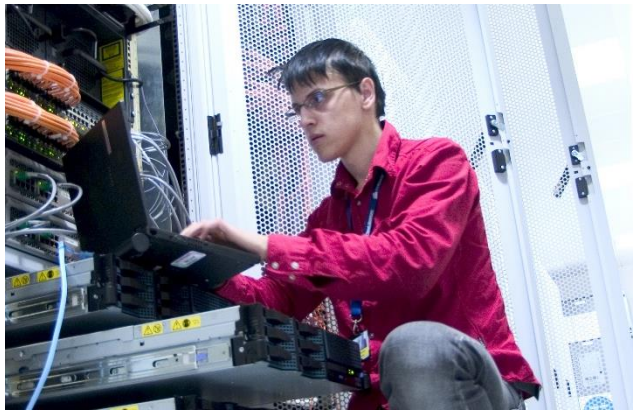
EMIL-VON-BEHRING GYMNASIUM

Client-Server-Programmierung



Allgemeine Hinweise

- Für diese Aufgabe sollte eine professionelle Entwicklungsumgebung (wie IntelliJ oder Eclipse) genutzt werden. Bei BlueJ kann nur eine main()-Methode gleichzeitig gestartet werden. Das heißt, dass hier der Server und der Client nicht aus demselben Programm gestartet werden können, außer wenn diese eine main()-Methode beide Objekte Client und Server erzeugt/startet.
- Zudem werden in diesem Projekt viele Java-Klassen benutzt, die importiert werden müssen. Hier bieten professionelle Entwicklungsumgebungen eine Importfunktion sowie eine bessere Code-Vervollständigung bei der Handhabung dieser Klassen an.
- Zudem können Try-Catch-Blöcke automatisch an allen nötigen Stellen durch die Entwicklungsumgebung gesetzt werden.

Allgemeine Hinweise

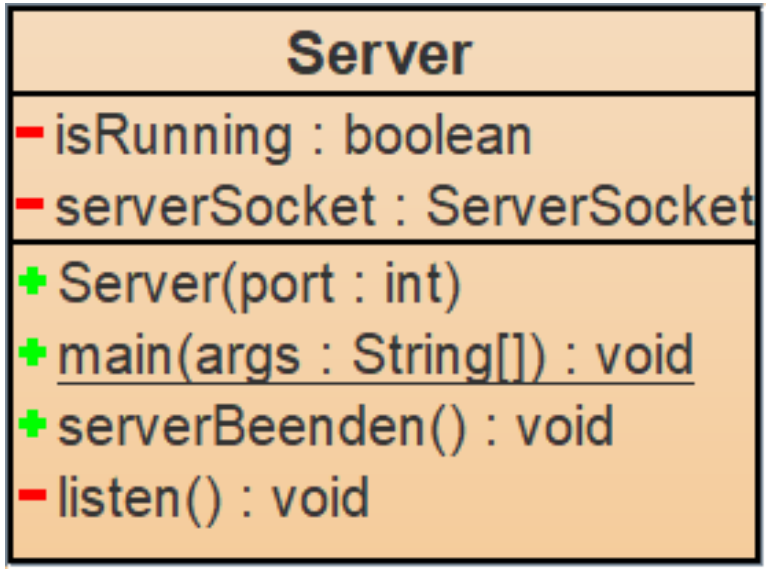
- Für das Programmieren der Client-Server-Kommunikation werden vielen Java-Klassen benötigt. Die Benutzung der Methoden dieser Klassen können in der Java-API nachgelesen werden (https://docs.oracle.com/javase/7/docs/api/). Welche Methoden jeweils benutzt werden sollen, kann dem Aufgabentext entnommen werden.
- Beispiel: Die Verbindungsanfrage soll von der Variable serverSocket akzeptiert werden. -> serverSocket.accept();

Allgemeine Aufgabenstellung

- In dieser Aufgabe soll eine einfache Server-Client-Kommunikation mittels des Transportprotokolls TCP realisiert werden.
- Der Server stellt einen Papagei mit dem Namen Polly dar. Polly ist sehr stur und will immer nur Kekse essen, ganz egal was man ihr auch sagt. Aber wenigstens geht sie auf Kommando („nerv nicht“) schlafen.

Klasse Server – Grundgerüst

- Schreibe eine Klasse `Server` und implementiere das Grundgerüst (alle Attribute und Methodensignaturen) gemäß der Klassenkarte.
- In der main-Methode soll ein neuer Server mit einem beliebigen Port > 1024 (z.B. 7569) erstellt werden.
- Im Konstruktor müssen alle Attribute initialisiert werden (`isRunning` auf **true**) und am Ende die Methode `listen()` aufgerufen werden.
- In der Methode `serverBeenden()` soll `isRunning` auf **false** gesetzt werden.

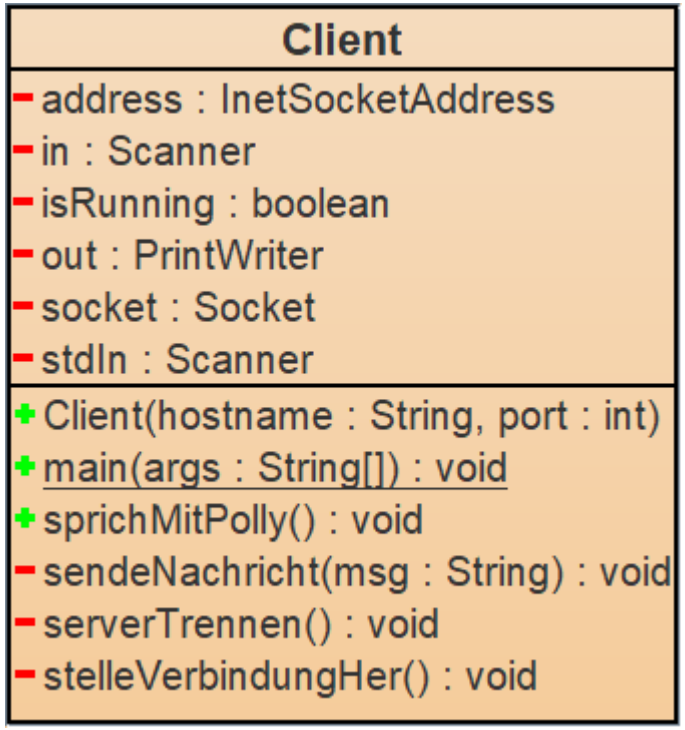


Klasse Server – Methode listen() – Teil 1

- Solange der Server läuft (`isRunning`) soll der Server jede eingehende Verbindung annehmen:
  - Der Server soll in der Methode listen() dauerhaft aktiv sein -> Dauerschleife mittels `isRunning`
  - Falls beim `serverSocket` eine Verbindunganfrage gestellt wird, soll diese akzeptiert werden und in einer lokalen Variable „`client`“ vom Datentyp `Socket` gespeichert werden.
  - Falls der Server nicht mehr laufen soll, muss nach der Dauerschleife der `serverSocket` geschlossen werden.

Klasse Client – Grundgerüst

- Schreibe eine Klasse `Client` und implementiere das Grundgerüst (alle Attribute und Methodensignaturen) gemäß der Klassenkarte.
- In der main-Methode soll ein neuer Client mit der IP-Adresse des gerade programmierten Servers und mit dem gewählten Port des Servers (z.B. 7569) erstellt werden.
- Falls der Server auf demselben Rechner läuft wie der Client, kann die IP-Adresse „127.0.0.1“ oder „localhost“ gewählt werden.



Klasse Client – Konstruktor Client()

- Im Konstruktor sollen alle Attribute des Clients initialisiert werden:
  - `stdIn` soll Konsoleneingaben einlesen und damit muss der Scanner auf `System.in` lauschen
  - Die Internetsocketadresse `address` soll mit dem `hostname` und dem `port` erzeugt werden.
  - Anschließend soll die Methode `stelleVerbindungHer()` aufgerufen werden. (Diese initialisiert dann die Attribute `socket`, `in` und `out`).
  - `isRunning` ist nach der Verbindungsherstellung **true**
  - Danach soll die Methode `sprichMitPolly()` aufgerufen werden. Diese Methode kommuniziert mit dem Server/Polly

Klasse Client – Methode stelleVerbindungHer()

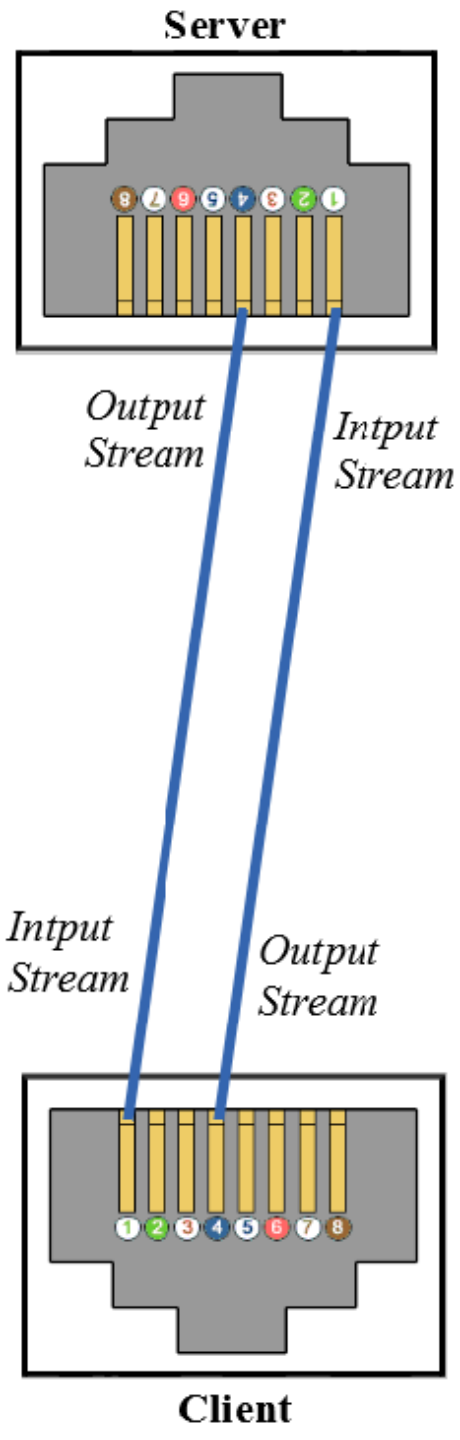
- In der Methode stelleVerbindungHer() werden die restlichen Attribute initialisiert:
  - Das Attribut socket bekommt einen Standard-Socket zugewiesen (Standardkontruktor).
  - Anschließend soll der Socket sich mit der Internetadresse `address` verbinden.
  - Der `Scanner in` und der `PrintWriter out` werden mit dem InputStream und dem OutputStream des Socket folgendermaßen initialisiert:

    - *in = new Scanner(new BufferedReader(new InputStreamReader(socket.getInputStream())))*

    - *out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(socket.getOutputStream())))*

Erklärung zur Initialisierung des In-/Out-Kanals

- Jeder Socket besitzt einen InputStream und einen OutputStream. Wenn sich zwei Socket verbunden haben, dann werden die Daten/Nachrichten über den Output des einen Sockets versendet und landet beim beim anderen Socket im Input und auch umgekehrt. (siehe Bild)
- Der InputStream und OutputStream arbeiten auf Bitebene. Das heißt, es werden beim InputStream Bytes gelesen/empfangen und beim OutputStream Bytes geschrieben/versendet.



Erklärung zur Initialisierung des In-/Out-Kanals

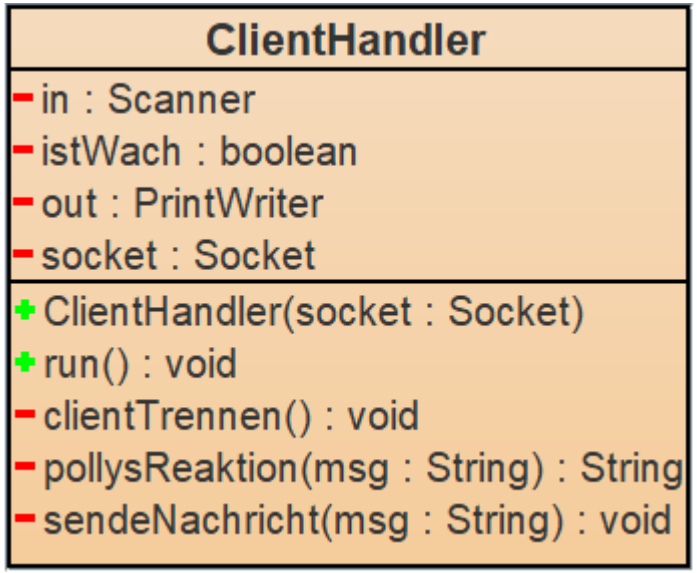
- Die Klassen InputStreamReader und OutputStreamWriter ermöglichen es, dass man die Daten **zeichenweise** (Buchstaben, Zahlen und Sonderzeichen nach einem spezifizierten Zeichensatz) lesen/schreiben kann.
- Die Klasse BufferedReader kann einen zeichenbasierten Stream lesen und puffert/speichert die gelesenen Zeichen zwischen, sodass man anschließend z.B. **zeilenweise** die Daten/Texte lesen kann.
- Die Klasse BufferedWriter puffert Zeichen(-folgen) (also Text), um effizientes Schreiben in einen zeichenbasierten Outputstream zu ermöglichen.

Erklärung zur Initialisierung des In-/Out-Kanals

- Der Scanner kann die eingelesenen Daten/Texte passend „zerteilen“ und für die Weiterverarbeitung umwandeln (z. B. Text in einer Zeile “5“ (Datentyp String) zu einer Zahl 5 (Datentyp int)). (Stichwort: Parser/Parsing)
- Die Klasse PrintWriter bietet die print(…)- bzw. println(…)-Methoden für primitive Datentypen und Strings an. Diese Methoden nutzen dabei automatisch die für das Betriebssystem passenden Zeilenumbruchszeichen (\n vs. \r\n). Zudem wirft die Klasse PrintWriter keine I/O-Exceptions (Input/Output).

Klasse ClientHandler – Grundgerüst

- In der Klasse Server müssen nun die angefragten Verbindungen von den Clients angenommen und abgearbeitet werden.
- Die Abarbeitung der Clientanfragen muss parallelisiert erfolgen, da sonst immer nur ein Client gleichzeitig mit dem Server kommunizieren kann.
- Für die Abarbeitung der Anfragen ist die Klasse ClientHandler zuständig, die als „Hilfsklasse“ in der Klasse vom Server angelegt werden kann. ClientHandler implementiert das Interface Runnable. Implementiere das Grundgerüst!



Klasse ClientHandler – Konstruktor

- Im Konstruktor sollen alle Attribute des ClientHandlers initialisiert werden:
  - Dem Referenzattribut socket wird der Übergabeparameter zugewiesen.
  - `istWach` ist true, da Polly jetzt Kekse haben will
  - `in` und `out` wird genauso wie beim Client initialisiert:

    - *in = new Scanner(new BufferedReader(new InputStreamReader(socket.getInputStream())))*

    - *out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(socket.getOutputStream())))*

Klasse Client/ClientHandler – Methode sendeNachricht()

- Implementiere in **beiden** Klassen Client und ClientHandler die Methode `sendeNachricht(String msg)`:
  - Die Nachricht `msg` soll mit der Methode `println(…)` vom PrintWriter in den Puffer geschrieben werden.
  - Anschließend soll mit der Methode `flush()` vom PrintWriter die Daten in den OutputStream geschrieben und damit „abgeschickt“ werden.

Klasse Client/ClientHandler – Methode trennen()

- Implementiere in **beiden** Klassen Client und ClientHandler die Methoden `serverTrennen()` und `clientTrennen()`. Beide Methoden erledigen dieselben Schritte:
  - Zunächst sollen die beiden Methoden den Scanner und den PrintWriter schließen
  - Anschließend soll der Socket geschlossen werden
- So werden zunächst die In- und Out-Kanäle und anschließend die Verbindung selbst zwischen Client und ClientHandler geschlossen.

Klasse Server – Methode listen() – Teil 2

- Nachdem die Verbindungsanfrage akzeptiert und in der lokalen Variable `client` gespeichert wurde, soll ein neuer ClientHandler mit diesem gespeicherten Socket `client` erzeugt werden.
- Anschließend soll der ClientHandler mithilfe eines Threads parallel gestartet werden.

Klasse ClientHandler – Methode run() – Teil 1

- Zur Erinnerung: der Dienst des Servers (also die Aufgabe des ClientHandlers) ist es einen nervigen Papagei Polly darzustellen, der immer nur Kekse haben möchte.
- Durch das Starten des Threads in der listen()-Methode, wird die run()-Methode des ClientHandlers ausgeführt (und Polly ist wach):
  - Sende als Erstes eine Nachricht an den Client: "Polly ist wach und will einen Keks“
  - Definiere dir für die Antwort an den Client eine lokale Variable `antwort`
  - Solange Polly wach ist, soll der Client mit ihr kommunizieren können -> Dauerschleife (Inhalt der Dauerschleife siehe nächste Folie)
  - Nach der Schleife soll die Verbindung zum Client getrennt werden.

Klasse ClientHandler – Methode run() – Teil 2

- In der Dauerschleife der run()-Methode soll folgendes implementiert werden:
  - Falls der Scanner eine neue Zeile lesen kann – **hasNextLine()**
  - Dann soll die nächste Zeile eingelesen werden – **nextLine()** – und der Methode `pollysReaktion(...)` übergeben werden
  - Die Rückgabe der Reaktionsmethode soll in der lokalen Variable `antwort` gespeichert werden.
  - Anschließend soll die `antwort` versendet werden

Klasse ClientHandler – Methode pollysReaktion()

- Zur Erinnerung: Polly ist sehr stur und will immer nur Kekse essen, ganz egal was man ihr auch sagt. Aber wenigstens geht sie auf Kommando („nerv nicht“) schlafen.
- Falls die übergebene Nachricht an die Methode pollysReaktion() „nerv nicht“ ist:
  - Dann soll Polly nicht mehr wach sein und als Antwort "Okay Polly geht schlafen!“ zurückgeben.
  - Ansonsten antwortet Polly "Polly will noch einen Keks!“

- Tipp: Nutze hierfür die Methode equalsIgnoreCase(…), um die Groß- und Kleinschreibung nicht zu beachten!

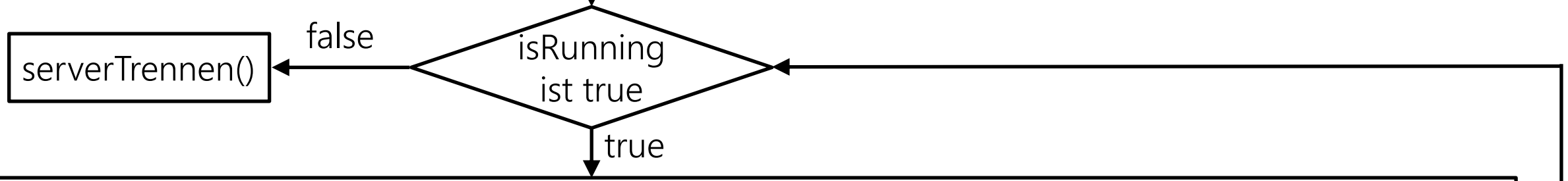
Klasse Client – Methode sprichMitPolly() – Teil 1

- Zuletzt muss noch die Kommunikation vom Client mit Polly fertiggestellt werden. Die Methode `sprichMitPolly()` wird nach der Verbindungsherstellung mit dem Server im Konstruktor aufgerufen.
- Der Client soll ermöglichen, dass du über die Konsole direkt mit Polly sprechen kannst. Also soll der Client die Antworten von Polly auf der Konsole ausgeben, deine Eingaben auf der Konsole einlesen und an den Server verschicken und anschließend wieder die Antwort von Polly wieder empfangen und ausgeben.
- Hierfür werden zwei lokale Variablen `consolenInput` und `pollysAntwort` vom Datentyp String benötigt
- Der Ablauf der Methode ist visuell auf der nächsten Folie dargestellt

Klasse Client – Methode sprichMitPolly() – Teil 2

- Falls man beim InputStream eine neue Zeile lesen kann, dann
  - Zeile einlesen und der Variablen pollysAntwort zuweisen
  - Pollys Antwort ausgeben in der Form: [Server] Pollys Antwort: "+pollysAntwort

Hinweis: Polly sendet sofort – nachdem die Verbindung herstellt ist – die Nachricht „Polly ist wach und möchte einen Keks". Diese Nachricht soll erstmal ausgegeben werden.



- Ausgabe auf Konsole in der Form: [Client] Was willst du Polly sagen?
- Falls man eine neue Zeile von der Konsole einlesen kann (hasNextLine() auf stdIn), dann
  - Dann soll die Variable consolenInput (für die Nachricht an Polly) mit dem Input des Benutzers eingelesen werden (nextLine() auf stdIn)
  - Die gerade eingelesene Nachricht soll versendet werden
- Falls man beim InputStream über den Scanner in eine neue Zeile einlesen kann, dann
  - Dann soll diese Antwort von Polly in der Variable pollysAntwort gespeichert werden
  - Pollys Antwort ausgeben in der Form: „[Client] Pollys Antwort: "+pollysAntwort
  - Falls Pollys Antwort „Okay Polly geht schlafen!" entspricht -> isRunning auf false

Test des Client-Server-Programms

- Teste dein Programm, in dem du zuerst den Server startest (main()-Methode des Servers) und anschließend den Client startest (main()-Methode des Clients), der sich dann mit diesem Server verbindet. Anschließend solltest du mit Polly sprechen können und ein mögliches „Gespräch“ könnte folgendermaßen aussehen:

```
[Server] Pollys Antwort: Polly ist wach und will einen Keks
[Client] Was willst du Polly sagen?
Hallo
[Server] Pollys Antwort: Polly will noch einen Keks!
[Client] Was willst du Polly sagen?
Keks
[Server] Pollys Antwort: Polly will noch einen Keks!
[Client] Was willst du Polly sagen?
Nerv nicht
[Server] Pollys Antwort: Okay Polly geht schlafen!

Process finished with exit code 0
```