

Aufgabe 10.1 (P) Programmierung eines Netzwerk-Servers

In dieser Aufgabe soll ein einfacher Dienst programmiert werden, der über eine Netzwerkverbindung erreichbar ist. Im Netzwerk hat jedes Gerät eine sog. *IP-Adresse*, mit der es von anderen Geräten aus erreichbar ist. Eine IP-Adresse ist ein 32-Bit-Integer, der byteweise durch Punkt getrennt geschrieben wird, z.B. 192.168.0.1. Jedes Gerät hat außerdem die feste *Loopback-Adresse* 127.0.0.1, unter der man es von sich selbst aus erreichen kann – dies eignet sich gut zum Testen von Netzwerkprogrammen. Da auf einem Computer mehrere Programme gleichzeitig das Netzwerk benutzen möchten, braucht jedes Programm außerdem eine zusätzliche Adresse innerhalb eines Gerätes, den sog. *Port*. Ein Port ist ein 16-Bit-Integer ohne Vorzeichen, der üblich einfach als Zahl angegeben wird.

Zur Kommunikation über das Netzwerk gibt es nun stets die folgenden beiden Rollen:

- Der *Server* öffnet einen Port, gibt beim Betriebssystem also an, an diesem Port auf eingehende Verbindungen zu warten.
- Der *Client* kann sich nun durch Angabe der IP-Adresse und des Ports zu dem Server-Programm verbinden.

In Java lässt sich ein Port (hier: 8000) durch die Klasse `java.net.ServerSocket` wie folgt öffnen:

```
ServerSocket serverSocket = new ServerSocket(8000);
```

Anschließend können Verbindungen über die `accept()`-Methode angenommen werden:

```
java.net.Socket client = serverSocket.accept();
```

Die Methode `accept()` liefert einen sog. *Socket* zurück, der die Verbindung mit dem gegebenen Client repräsentiert. Ein Socket erlaubt es nun, Daten mit dem Client auszutauschen. Dazu bietet die Klasse `Socket` die Methoden

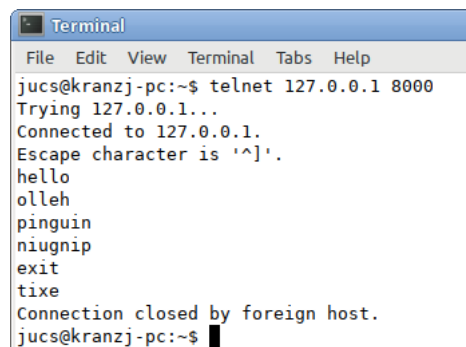
```
InputStream java.net.Socket.getInputStream() throws IOException
```

und

```
OutputStream java.net.Socket.getOutputStream() throws IOException
```

an. Nutzen Sie die beschriebenen Klassen und Methoden, um einen Service zu implementieren, der je eine Zeile Text vom Client empfängt und diese buchstabenweise umgedreht wieder zurückschickt. Sendet der Client das Wort `exit`, soll die Verbindung geschlossen werden. Bricht der Client die Verbindung ab oder sendet das Wort `exit`, so soll sich anschließend wieder ein neuer Client mit dem Service verbinden können.

Zur Kommunikation mit dem Server können Sie den Befehl `telnet` verwenden¹. Die folgende Abbildung zeigt eine typische Session bei der Kommunikation mit dem in dieser Aufgabe erstellten Dienst:



```
Terminal
File Edit View Terminal Tabs Help
jucs@kranzj-pc:~$ telnet 127.0.0.1 8000
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
hello
olleh
penguin
niugnip
exit
tixe
Connection closed by foreign host.
jucs@kranzj-pc:~$
```

Hinweis: Nutzen Sie die Klasse `BufferedReader` und `InputStreamReader` zum Verarbeiten des `InputStream` sowie `OutputStreamWriter` und `PrintWriter` zum Verarbeiten des `OutputStream` des Sockets.

Aufgabe 10.2 (P) Programmierung eines Netzwerk-Clients

In der letzten Aufgabe haben Sie einen einfachen Service programmiert und mit diesem anschließend über ein vorhandenes Konsolenprogramm kommuniziert. In dieser Aufgabe soll nun ein Client geschrieben werden, der den Service nutzt, um ein vom Nutzer eingegebenes Wort buchstabenweise zu spiegeln.

Sie können eine Verbindung zu einem Service aufnehmen, indem Sie eine Instanz der Klasse `java.net.Socket` erstellen und dieser den Host (hier: "127.0.0.1") und Port (hier: 8000) im Konstruktor übergeben. Senden Sie anschließend den vom Nutzer eingegebenen String als Zeile als den Service; geben Sie die Antwort des Servers auf der Konsole aus.

Hinweis: Verwenden Sie `flush()`-Methode der Klasse `PrintWriter`, um sicherzustellen, dass die Anfrage tatsächlich an den Server verschickt wird.

Aufgabe 10.3 (P) Download einer CSV-Datei

In der vorherigen Aufgabe haben wir einen einfachen Netzwerk-Client erstellt, der mit unserem eigenem Server interagiert. Nun wollen wir mit einem bereits vorhandenen Server kommunizieren, in diesem Fall einem Webserver. Das Ziel ist der Download der CSV-Datei, die wir auf Blatt 9 verwendet haben. Diese ist unter <http://www2.in.tum.de/~kranzj/temperaturesEurope1963Till2013ByCity.csv> online verfügbar.

Wie bereits an der URL ersichtlich, steht diese Datei über das sog. *HTTP-Protokoll* zur Verfügung. HTTP ist ein textbasiertes Protokoll, bei dem zunächst der Client einen

¹Unter Windows steht stattdessen das Programm PuTTY unter <https://www.putty.org/> zur Verfügung.

Request an den Server schickt. Dieser wiederum antwortet mittels einer *Response* – anschließend wird die Verbindung geschlossen. Sowohl Request als auch Response bestehen jeweils aus zwei Teilen – dem sog. *Header* und dem *Body*, wobei Header und Body auch jeweils leer sein können.

Besuchen Sie z.B. die Webseite <https://www.google.de> mit Ihrem Browser², sendet dieser z.B. folgenden Request:

```
1 GET / HTTP/1.1
2 Host: www.google.de
3 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:64.0)
   ↪ Gecko/20100101 Firefox/64.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
5 Accept-Language: de,en;q=0.7,en-US;q=0.3
6 Accept-Encoding: gzip, deflate, br
7 DNT: 1
8 Connection: keep-alive
9 Upgrade-Insecure-Requests: 1
```

Der Request beginnt mit dem Request-Header. In der ersten Zeile findet sich Typ des Requests (in dieser Aufgabe stets **GET**) sowie der Pfad, auf die wir zugreifen möchten. Es folgen darauf eine Reihe von Attributen, z.B. die Version des Browsers und Betriebssystems in der zweiten Zeile (alle Attribute sind optional, das **Host**-Attribut wird allerdings üblich benötigt, damit der Server die Anfrage bearbeiten kann). Wichtig ist, dass die einzelnen Zeilen zur Trennung ein *Carriage Return*- und *Line Feed*-Symbol verwenden, also **\r\n**. Das Ende des Headers wird durch eine leere Zeile markiert. Darauf folgt der Body des Requests, der in diesem Fall leer ist.

Im Beispiel sieht die Antwort des Google-Servers verkürzt wie folgt aus:

```
1 HTTP/2.0 200 OK
2 date: Fri, 21 Dec 2018 14:51:07 GMT
3 expires: -1
4 cache-control: private, max-age=0
5 content-type: text/html; charset=UTF-8
6
7 <!doctype html><html itemscope="" itemtype="http://schema.org/WebPage"
   ↪ lang="de">
8 ...
9 </html>
```

In ersten Zeile findet sich die Protokoll-Version des Servers sowie ein Fehlercode (in diesem Fall 200, was meint, dass kein Fehler aufgetreten ist). Es folgt der Response-Header, der wiederum Attribute enthält. Durch eine leere Zeile getrennt wird anschließend der Body des Response gesendet – dies ist in unserem Fall die Webseite von Google.

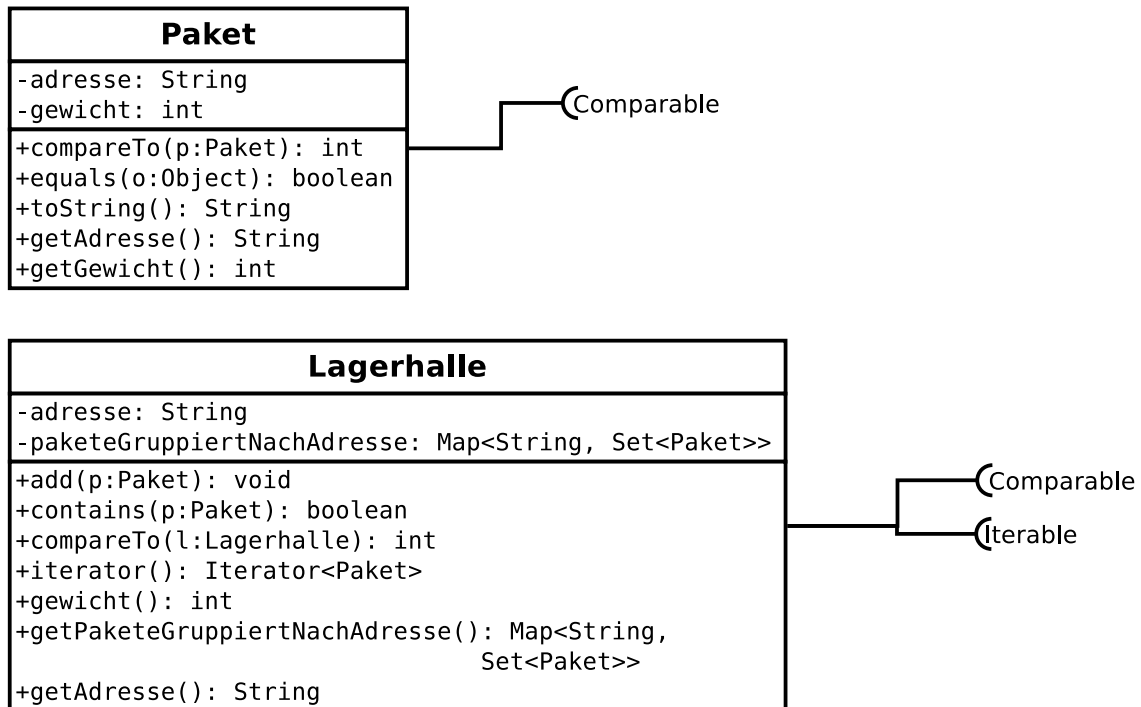
Implementieren Sie nun einen Client, der die angegebene CSV-Datei auf ihren Computer herunterlädt. Sie müssen im Request-Header lediglich die ersten zwei Zeilen des Beispiels übernehmen und anpassen, weitere Attribute werden keine benötigt. Nutzen Sie in dieser Aufgaben keine Schleifen der Rekursion, verwenden Sie stattdessen ausschließlich die *Streams*-API von Java.

²Hier wird, wie heutzutage üblich, die verschlüsselte Version von HTTP, nämlich HTTPS, verwendet.

Aufgabe 10.4 (P) Iteratoren

In dieser Aufgabe implementieren wir ein Beispiel für Iteratoren, welches sich an dem aus der Vorlesung bekannten Beispiel der Fussballmannschaft orientiert.

1. Implementieren Sie die Interfaces `Comparable` und `Iterable` für `Paket` und `Lagerhalle` im Sinne des folgenden UML-Diagramms, welches durch die Verbindung mit dem offenen Halbkreis die **implements**-Beziehung symbolisiert:



2. Verwenden Sie die Methode `Lagerhalle.iterator()`, um die Methode `gewicht()` zu implementieren. Rufen Sie dafür den `Paket`-Iterator ab und iterieren Sie mittels dem über `iterator()` gewonnenen Iterator über alle in der Lagerhalle enthaltenen Pakete, um deren Gewicht aufzuaddieren. Dabei sollen natürlich Duplikate vermieden werden. Verwenden Sie außerdem beim Iterieren das `foreach`-Konstrukt.
3. Legen Sie zum Testen 10 verschiedene Pakete in der Lagerhalle über eine entsprechende `main()`-Methode ab. Achten Sie darauf, dass an manche Adressen mehrere Pakete zugestellt werden sollen. Testen Sie den Iterator auf Vollständigkeit und das Gewicht auf Korrektheit.
4. Wandeln Sie das `Iterable` Lagerhaus mit Hilfe der Java-Bibliotheksfunktion `StreamSupport.stream()` in einen `Paket`-Stream um. Diskutieren Sie, was diese Hilfsfunktion macht, insbesondere vor dem Hintergrund des `Iterable`-Interfaces und den `default`-Modifiern. Benutzen Sie dann die Java Streams API, um die Pakete im Lagerhaus nach Gewicht zu sortieren. Konkatenieren Sie alle Adressen der resultierenden Paketliste zu einem String mittels `reduce()`. Der Trenner zwischen den Adressen soll das Newline-Zeichen sein. Geben Sie das Ergebnis auf der Kommandozeile aus. Benutzen Sie die Stream API außerdem, um pro Adresse Anzahl und Gesamtgewicht zu berechnen und in geeigneter Formatierung auf der Kommandozeile auszugeben.

Aufgabe 10.5 (P) ArraySet

In dieser Aufgabe geht es darum, eine einfache Mengen-Datenstruktur zu programmieren. Die Datenstruktur soll unveränderlich sein; schreibende Operationen auf ihr erzeugen also jeweils eine neue Instanz. Gegeben sei dafür das folgende Gerüst:

```
1 public class ArraySet<T> {
2     private T[] elements;
3
4     public ArraySet() {
5         // Hinweis: Erstellen eines generischen Arrays wie
6         // hier gezeigt möglich!
7         this.elements = (T[]) (new Object[0]);
8     }
9
10    private ArraySet(T[] elements) {
11        this.elements = elements;
12    }
13
14    @Override
15    public String toString() {
16        return Arrays.toString(elements);
17    }
18 }
```

Implementieren Sie zunächst die Methode

```
public ArraySet<T> add(T element),
```

die der Menge ein Element hinzufügt. Achten Sie darauf, dass Elemente nicht mehrfach in einer Menge vorkommen dürfen. Implementieren Sie nun die folgenden Methoden, die Sie bereits aus der Stream-API kennen:

1. Implementieren Sie die Methode

```
public ArraySet<T> filter(Predicate<T> pred),
```

die eine neue Menge von Objekten erzeugt, welche nur diejenigen Elemente der gegebenen Menge enthalten, die das Prädikat `pred` erfüllen. Beachten Sie, dass das Prädikat pro Element nur einmal ausgewertet werden darf.

Hinweis: Nutzen Sie die Methode `boolean test(T t)` des Interfaces `Predicate<T>`, um zu testen, ob das Prädikat für ein bestimmtes Objekt `t` erfüllt ist.

2. Implementieren Sie die Methode

```
public <U> ArraySet<U> map(Function<T, U> mapFunc),
```

die eine Menge von Objekten auf eine andere Menge von Objekte durch eine Mapping-Funktion abbildet.

Hinweis: Nutzen Sie die Methode `U apply(T t)` des Interfaces `Function<T, U>`, um die Funktion `mapFunc` auf ein Objekt `t` anzuwenden.

3. Implementieren Sie die Methode

```
public <U> U reduce(U init, BiFunction<T, U, U> func),
```

welche die Elemente einer Menge angefangen mit dem Initialwert `init` ab Index 0 wiederholt mittels der Funktion `func` verknüpft und das Endergebnis zurückliefert.

Hinweis: Nutzen Sie die Methode `U apply(T t, U u)` des Interfaces `BiFunction<T, U, U>`, um die Funktion `func` auf ein Objekt `t` und ein Objekt `u` anzuwenden.

Testen Sie Ihre Implementierung mit und ohne Benutzung von Lambda-Ausdrücken:

1. Schreiben Sie eine Klasse `EvenStringsPred` **implements** `Predicate<String>`, welche die Methode `public boolean test(String arg0)` implementiert³. Nutzen Sie eine Instanz dieser Klasse als Parameter für die Methode `filter()` einer Menge von Strings. Mithilfe des Prädikats sollen sich aus einer Menge diejenigen Strings herausfiltern lassen, die eine gerade Länge haben. Testen Sie analog dazu auch die Methoden der beiden anderen Teilaufgaben.
2. Testen Sie die Methoden mit passenden Lambda-Funktionen. An einer Instanz von `ArraySet<Integer>` mit dem Namen `arraySet` können sie beispielsweise die Methode `arraySet.filter(x -> x % 2 == 0)` aufrufen, um eine neue Instanz von `ArraySet<Integer>` zu erhalten, welche nur gerade Zahlen beinhaltet.

Hinweis: Nutzen Sie zur Umsetzung lediglich Klassen und Methoden der Java-Klasse `java.util.Arrays` sowie des Packages `java.util.function`. Nutzen Sie insbesondere nicht die Streams-API.

Die Hausaufgabenabgabe erfolgt über Moodle. Bitte geben Sie Ihren Code als UTF8-kodierte (ohne BOM) Textdatei(en) mit der Dateiendung `.java` ab. Geben Sie **keine** Projektdateien Ihrer Entwicklungsumgebung ab. Geben Sie **keinen** kompilierten Code ab (`.class`-Dateien). Sie dürfen Ihren Code als Archivdatei abgeben, dabei allerdings **ausschließlich** das ZIP-Format nutzen. Achten Sie darauf, dass Ihr Code kompiliert. Hausaufgaben, die sich nicht im vorgegebenen Format befinden, werden nur mit Punktabzug oder gar nicht bewertet.

Aufgabe 10.6 (H) Der TestIt-Server

[4 Punkte]

In dieser Aufgabe geht es darum, die Funktionalität der auf der Blatt 8 erstellten `TestIt`-Klasse über das Netzwerk anzubieten. Erstellen Sie dazu eine Klasse `TestItServer`, die einen Netzwerkdienst an Port 8000 anbietet und exakt die gleichen Befehle erwartet, wie von Blatt 8 und vorher bekannt.

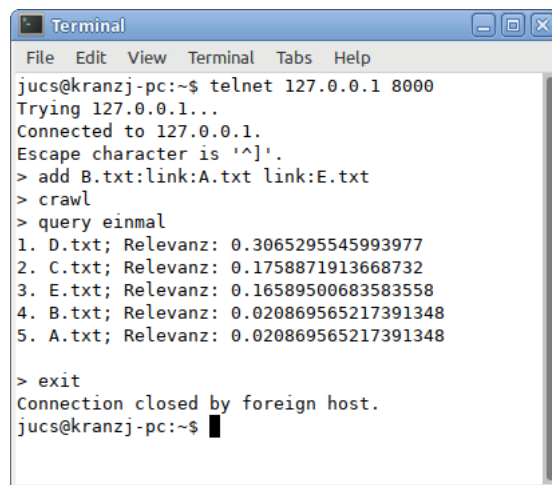
Achten Sie bei der Implementierung auf folgende Dinge:

- Wie bereits von Blatt 8 bekannt, soll die Eingabe des Befehls `exit` dazu führen, dass die Verbindung zum Client geschlossen wird; es können sich dann neue Clients mit dem Server verbinden.

³Die anderen Methoden des Interfaces sind als sog. *default-Methoden* bereits implementiert und müssen nicht von Ihnen geschrieben werden (siehe <https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html>).

- Der Client kann die Verbindung auch unerwartet jederzeit schließen. Dies soll nicht dazu führen, dass der Server abstürzt, sondern dass dieser auf neue Verbindungen wartet.
- Nachdem ein Client sich vom Server getrennt hat, kann sich ein weiterer Client mit diesem verbinden. Die Clients sollen dabei von einander getrennt sein, sie sollen sich also keine `LinkedDocumentCollection` teilen.
- Die Dokumente, die im Dateisystem erwartet und mittels `crawl()` hinzugefügt werden, befinden sich auf dem Server. Diese müssen also nicht vom Client hochgeladen werden.

Testen Sie Ihre Implementierung wie im Präsenz-Teil besprochen. Eine Session könnte z.B. wie folgt ablaufen.



```

Terminal
File Edit View Terminal Tabs Help
jucs@kranzj-pc:~$ telnet 127.0.0.1 8000
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
> add B.txt:link:A.txt link:E.txt
> crawl
> query einmal
1. D.txt; Relevanz: 0.3065295545993977
2. C.txt; Relevanz: 0.1758871913668732
3. E.txt; Relevanz: 0.16589500683583558
4. B.txt; Relevanz: 0.020869565217391348
5. A.txt; Relevanz: 0.020869565217391348

> exit
Connection closed by foreign host.
jucs@kranzj-pc:~$

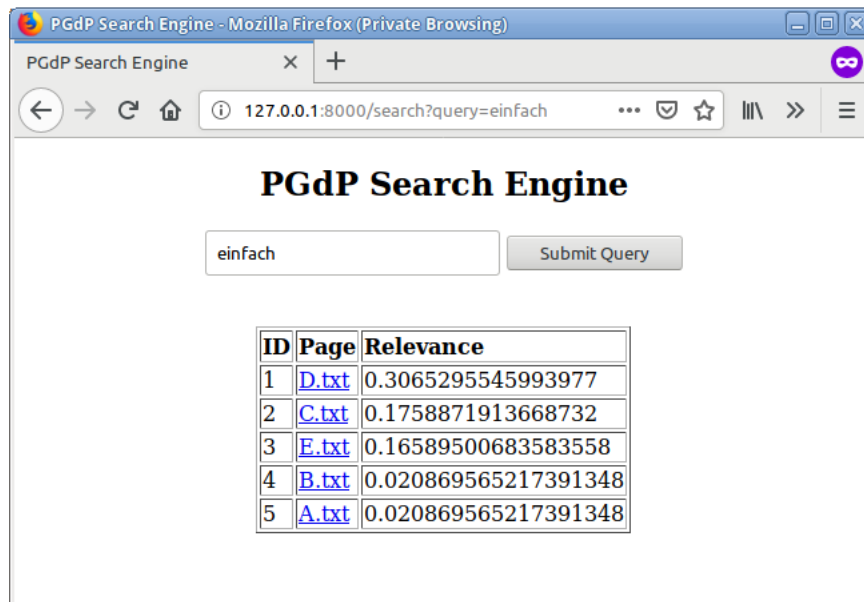
```

Hinweis: Achten Sie darauf, mittels der `flush()`-Methode Daten tatsächlich über das Netzwerk zu verschicken.

Können zu Ihrem Server mehrere Clients gleichzeitig verbinden? Und wenn ja, können diese ihn auch gleichzeitig benutzen? Begründen Sie Ihre Antwort in einem kurzen Kommentar.

Aufgabe 10.7 (H) Webserver für die Suchmaschine [11 Punkte + 3 Bonuspunkte]

In dieser Aufgabe geht es darum, einen Webserver für die Suchmaschine zu programmieren, den ein Nutzer (der Client) mit seinem Webbrowser ansprechen kann. Wir wollen es also ermöglichen, im Browserfenster eine Webseite aufzurufen, auf der man eine Anfrage in ein einfaches Formular eingeben und durch einen Klick auf einen Button an den Server abschicken kann. Der Server berechnet daraufhin die Liste der relevanten Dokumente und schickt sie an den Client-Browser zurück, in dem dann wiederum auf einzelne Dokumente in der Resultatliste geklickt werden kann – so wie Sie das von Google Search kennen. Die folgende Abbildung zeigt die vom Client-Browser angezeigte Antwort des Servers auf die Suchanfrage nach dem Wort *einfach*.



Die Kommunikation zwischen dem Client und dem Server läuft im Web allgemein wie folgt ab.

1. Zunächst schickt der Client einen *Request* an den Server, er nutzt dazu einen Browser (z.B. Mozilla Firefox oder Google Chrome). Der Server kann durch eine sog. URL (z.B. <http://127.0.0.1:8000/>) adressiert werden. Die URL enthält hier zunächst das verwendete Protokoll, nämlich HTTP. Es folgen die Adresse des Servers (hier die Loopback-IP-Adresse 127.0.0.1), der Port (hier 8000) sowie der Pfad (hier /). Es können außerdem nach einem Fragezeichen (?) Parameter übergeben werden, die durch ein &-Symbol getrennt werden. Die URL <http://127.0.0.1:8000/search?query=einfach> für den Pfad `/search` übergibt dem Server zusätzlich einen Parameter `query` mit dem Wert `einfach`.
2. Der Server berechnet die *Response* und schickt so die Webseite an den Client, die anschließend von dessen Browser angezeigt wird.

Die Aufgabe des Browsers dabei ist also einerseits, einen Request des Nutzers, z.B. Formulardaten, in einen passenden HTTP-Request zu übersetzen, der über das Netzwerk versandt wird und den der Server interpretieren kann. Andererseits zeigt der Client-Browser die vom Server mittels dem HTTP-Protokoll gesandte Antwort an. Die Antwort des Servers enthält üblicherweise ein Dokument im sog. HTML-Format, welches zur Darstellung von Webseiten verwendet wird. Ein solches HTML-Dokument enthält alle nötigen Anweisungen, um die zunächst rein textuellen Elemente der Response des Servers graphisch ansprechend aufzubereiten.

Unser Server muss folgende Anfragen bearbeiten können.

- **Hauptseite:** Gibt der Benutzer im Browser die URL des Servers mit dem Pfad / und ohne Parameter ein, antwortet dieser mit der Hauptseite. Bei uns ist dies die Suchmaschine mit einer leeren Eingabemaske und ohne Tabelle mit Suchergebnissen.
- **Suche:** Wird auf dem Server der Pfad `/search` mit dem Parameter `query=text` angefordert (siehe die Adresszeile in der Abbildung), so soll dem Browser die Suchergebnisse für den Begriff `text` zurückliefern werden, die dieser anschließend anzeigt. Dieses Szenario zeigt obige Abbildung, in der statt `text` die Anfrage `einfach` geschickt wurde.

- **Datei:** Wird ein anderer Pfad, z.B. `/C.txt`, angefordert, wird nach der gleichnamigen Datei (ohne `/`) gesucht und der Inhalt ggf. zurückgeliefert. So soll es möglich sein, auf ein Suchergebnis zu klicken, um sich den Inhalt des Dokuments anzeigen zu lassen.

Ein typischer Ablauf einer Interaktion zwischen einem Client und dem Server könnte z.B. wie folgt aussehen:

1. Der Benutzer (Client) gibt im Browser die Adresse der Hauptseite der Suchmaschine ein, z.B. `http://127.0.0.1:8000/`.
2. Der Server liefert die Hauptseite zurück, auf der der Nutzer nun den Suchbegriff, z.B. *einfach*, in die Suchmaske eintippt. Anschließend klickt der Nutzer auf den *Submit Query*-Button. Dies veranlasst den Browser dazu, die URL `http://127.0.0.1:8000/search?query=einfach` an den Server zu senden.
3. Der Server erhält nun also die zweite Anfrage, nämlich die Suchanfrage. Er liefert deswegen die oben beschriebene Seite zurück, die eine Tabelle der Resultate enthält. Der Browser zeigt nun die neue Seite mit den Resultaten an (vgl. Abbildung).
4. Der Nutzer entscheidet sich, das oberste Dokument anzusehen. Er klickt deswegen in der Tabelle auf `D.txt` (im Beispiel der Abbildung). Dies veranlasst den Browser dazu, die URL `http://127.0.0.1:8000/D.txt` aufzurufen.
5. Der Server erhält so nun die dritte Anfrage und liefert die Datei `D.txt` an den Client zurück.

Beachten Sie, dass Sie insgesamt 3 Bonuspunkte für die Aufgabe erhalten können, sofern Sie keine Schleifen oder Rekursion, sondern ausschließlich die Streams-API verwenden. Unabhängig vom Bonus dürfen aus der Java-Standardbibliothek lediglich die Klassen `java.net.Socket` und `java.net.ServerSocket` sowie die Pakete `java.util` und `java.io` verwenden.

Wie bereits erwähnt, nutzt die Kommunikation das HTTP-Protokoll, welches schon aus den P-Aufgaben bekannt ist. Wir wollen im ersten Schritt Klassen erstellen, die einen HTTP-Request bzw. eine HTTP-Response repräsentieren. Dazu seien zunächst die folgenden Enumerationen gegeben, die Sie für Ihre Lösung verwenden können:

```

1 public enum HttpStatus {
2     Ok(200, "Ok"),
3     BadRequest(400, "Bad Request"),
4     Forbidden(403, "Forbidden"),
5     NotFound(404, "Not Found"),
6     MethodNotAllowed(405, "Method Not Allowed");
7
8     private int code;
9
10    public int getCode() {
11        return code;
12    }
13

```

```

14     private String text;
15
16     public String getText() {
17         return text;
18     }
19
20     private HttpStatus(int code, String text) {
21         this.code = code;
22         this.text = text;
23     }
24 }
25
26 public enum HttpMethod {
27     GET, POST
28 }

```

Beachten Sie, dass die verschiedenen Aufzählungskonstanten hier je zwei Parameter erhalten⁴. Diese Parameter werden im Konstruktor jeweils innerhalb des mit dieser Konstanten verknüpften Objekts gespeichert. An den Konstanten lassen sich anschließend die definierten Getter aufrufen; zum Beispiel kann man den `code` von `HttpStatus.Ok` durch `HttpStatus.Ok.getCode()` erhalten.

Nutzen Sie diese Enumerationen, um die Klassen `HttpRequest` und `HttpResponse` zu implementieren. Die Klasse `HttpRequest` erhält dabei die Request-Zeile (erste Zeile des Headers) als Parameter im Konstruktor und soll daraus die Attribute `HttpMethod method`, `String path` und `Map<String, String> parameters` berechnen (da der *Body* eines Requests für unseren Anwendungsfall nicht relevant ist, ignorieren wir diesen hier). Aus dem Request `GET /search?query=einfach HTTP/1.1` wird also die HTTP-Methode `GET`, der Pfad `/search` und der Parameter `query` \mapsto `einfach` extrahiert. Am Ende des Request steht jeweils nur die HTTP-Protokollversion (`HTTP/1.1`), die wir hier ignorieren. Lassen sich Methode, Pfad oder Parameter nicht aus der Request-Zeile extrahieren, so soll eine Exception vom Typ `InvalidRequestException` geworfen werden, die von der Klasse `RuntimeException` erbt. Die Klasse `HttpResponse` verfügt über die Attribute `HttpStatus status` und `String body`, welche sie im Konstruktor erwartet. Sie soll außerdem die Methode `String toString()` überschreiben, in welcher sie die String-Darstellung der Antwort derart formatiert erzeugt, dass diese anschließend an den Browser geschickt werden kann.

Der Client-Browser erwartet von uns einen Body innerhalb der HTTP-Response des Servers, der sich im HTML-Format befindet. Betrachten Sie zunächst den HTML-Code der Webseite, die Sie in obiger Abbildung sehen:

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title>PGdP Search Engine</title>
5  </head>
6

```

⁴<https://docs.oracle.com/javase/tutorial/java/java00/enum.html>

```

7 <body>
8 <center>
9 <h2>PGdP Search Engine</h2>
10 <form action="/search">
11   <input type="text" name="query" value="einfach">
12   <input type="submit" value="Submit Query">
13 </form>
14 </center>
15
16 <br><br>
17
18 <center>
19 <table border="1px solid black">
20 <tr><td><b>ID</b></td><td><b>Page</b></td><td><b>Relevance</b></td></tr>
21 <tr><td>1</td><td><a href="D.txt">D.txt</a></td>
22   <td>0.3065295545993977</td></tr>
23 <tr><td>2</td><td><a href="C.txt">C.txt</a></td>
24   <td>0.1758871913668732</td></tr>
25 <tr><td>3</td><td><a href="E.txt">E.txt</a></td>
26   <td>0.16589500683583558</td></tr>
27 <tr><td>4</td><td><a href="B.txt">B.txt</a></td>
28   <td>0.020869565217391348</td></tr>
29 <tr><td>5</td><td><a href="A.txt">A.txt</a></td>
30   <td>0.020869565217391348</td></tr>
31 </table>
32 </center>
33
34 </body>
35 </html>

```

HTML ist dabei eine spezielle Form von XML⁵. Machen Sie sich zunächst mit den Grundlagen von XML vertraut, sodass Sie den Aufbau des Dokuments verstehen. Unsere Webseite besteht aus folgenden Teilen.

- **Header:** Der Titel der Webseite wird im Header (Zeile 3 bis 5) festgelegt.
- **Body:** Auf den Header folgt der Body der Webseite (Zeile 7 bis 34), der wiederum aus mehreren Teilen besteht.
 - **Überschrift:** Die Überschrift befindet sich in Zeile 9.
 - **Eingabezeile mit Suchknopf:** Die Zeilen 10 bis 13 enthalten den Code für die Eingabe des Suchbegriffs und dem entsprechenden Button, die Suche abzuschicken. Dies ist als sog. **Form** (Formular) umgesetzt; nähere Informationen dazu finden Sie unter https://www.w3schools.com/html/html_forms.asp. Zeile 11 setzt das query-Attribut auf den Wert *einfach*, womit das Wort *einfach* im Suchfeld im Browser angezeigt wird. Das liegt daran, dass das gezeigte HTML-Dokument die Antwort auf eine vorher durch den Benutzer eingegebene Suchanfrage *einfach* repräsentiert und diese Suchanfrage dem Benutzer

⁵https://de.wikipedia.org/wiki/Extensible_Markup_Language

auch in der Antwortseite angezeigt werden soll. Für die Eingabe der nächsten Suchanfrage kann der Inhalt des Suchfelds durch den Benutzer überschrieben werden; darum müssen Sie sich nicht kümmern.

- **Tabelle mit Suchbegriffen:** Die Zeilen 19 bis 31 enthalten die Suchergebnisse. Jede Zeile wird dabei von `<tr>` und `</tr>` umschlossen, jede Zelle innerhalb einer Zeile von den Tags `<td>` und `</td>`. Eine genaue Beschreibung von Tabellen in HTML finden Sie unter https://www.w3schools.com/html/html_tables.asp. Jede Zeile der Tabelle verlinkt auf das jeweilige Dokument, sodass der Nutzer es per Klick abrufen kann. Ein relativer Link auf das Dokument `A.txt` mit dem Link-Text `A.txt` lässt sich in HTML mittels `A.txt` repräsentieren. Sie finden mehr Informationen über Links in HTML unter https://www.w3schools.com/html/html_links.asp.

Für die vom Server an den Client gesendeten Antworten wollen wir ein sog. *Template* als Grundlage verwenden. Das Template ist eine HTML-Datei, die die Struktur der Antwortseite beinhaltet, in die aber noch anfragespezifische Antwortteile eingesetzt werden müssen. Das Template besteht also aus Überschrift, dem Suchfeld, einem Platzhalter für einen ggf. voreingetragenen Wert im Suchfeld (s.o.) und einem Platzhalter für die Tabelle mit Relevanz und Links. Die Platzhalter, die üblicherweise mit einem vorangestellten %-Zeichen repräsentiert sind, werden dann durch konkrete Antworten ersetzt. Das Template muss nicht verwendet werden, wenn der Client eine Datei-Anfrage (s.o.) sendet, nachdem der Nutzer auf einen Link in der Antworttabelle geklickt hat. Erstellen Sie eine Klasse `TemplateProcessor`, die den Dateinamen der Template-Datei im Konstruktor erwartet. Sie soll außerdem die Methode

```
public String replace(java.util.Map<String, String> variableAssignments)
```

anbieten, die im Inhalt der Datei alle Vorkommen eines Schlüssels der übergebenen `Map` durch den zugeordneten Wert ersetzt und das Ergebnis zurückliefert. Nehmen Sie als Beispiel an, dass die Datei `foo.txt` den String `Lieber %first_name %last_name, wie geht es dir?` enthält. In diesem Fall erzeugt das Programm

```
1 TemplateProcessor tp = new TemplateProcessor("foo.txt");
2 java.util.TreeMap<String, String> varass = new java.util.TreeMap<>();
3 varass.put("%last_name", "Guin");
4 varass.put("%first_name", "Pin");
5 String text = tp.replace(varass);
6 System.out.println(text);
```

die Ausgabe: `Lieber Pin Guin, wie geht es dir?`

Erstellen Sie eine Datei `search.html`, die ein passendes Template für den Webserver enthält. Geben Sie diese Datei auch auf Moodle ab.

Wir können nun den Webserver selbst implementieren! Der Webserver führt die folgenden Schritte durch:

1. Zunächst wartet der Webserver darauf, dass sich ein Client mit ihm verbindet.

2. Ist ein Client verbunden, sendet dieser den HTTP-Request an den Server. Anhand der ersten Zeile des Headers des Requests erstellt der Server ein `HttpRequest`-Objekt und entscheidet anhand des Pfades, welche Art von Anfrage vorliegt. Erinnern Sie sich, dass dieser Pfad auch `/` sein kann; in diesem Fall wird die Startseite angezeigt.
3. Mithilfe dieser Daten kann der Webserver nun die Antwort generieren. Dazu werden Sie gleich unten verschiedene Methoden implementieren, die jeweils für einen `HttpRequest` eine Antwort in Form eines `HttpResponse`-Objektes errechnen.
4. Der Webserver nutzt nun die `toString()`-Methode der Klasse `HttpResponse` und sendet den resultierenden String als Antwort zurück an den Client.
5. Schließlich kehrt der Webserver zu Schritt 1 zurück und wartet auf die nächste Verbindung eines Clients.

Implementieren Sie in der Klasse `Webserver` zunächst die Methode

```
private static HttpResponse handleMainPage(),
```

die eine Antwort auf die Anfrage nach der Hauptseite, also für eine Anfrage nach dem Pfad `/` und ohne Suchwort, generiert. Die Hauptseite wird dazu als HTML-Dokument in die *Body* des HTTP-Response abgelegt. Wie oben erwähnt, entspricht die Hauptseite genau dem gezeigten HTML-Code, jedoch mit einer leeren Tabelle (Sie müssen hier einfach keine Zeilen in Ihr Template einsetzen) von Suchergebnissen sowie mit leerer Eingabezeile.

Implementieren Sie nun die Methode

```
private static HttpResponse handleSearchRequest(String query),
```

die eine Anfrage `query` gegen eine `LinkedDocumentCollection` matcht. Generieren Sie die `LinkedDocumentCollection` mit dem Namen `ldc` durch folgenden Code:

```

1  double pageRankDampingFactor = 0.85;
2  double weightingFactor = 0.6;
3  LinkedDocumentCollection ldc;
4  {
5      LinkedDocumentCollection temp = new LinkedDocumentCollection();
6      temp.appendDocument(new LinkedDocument("B.txt", "", "", null, null,
7          ↪ "link:A.txt link:E.txt", "B.txt"));
8      ldc = temp.crawl();
9  }
```

Erzeugen Sie eine Tabelle, die der Tabelle in der Abbildung am Anfang der Aufgabe entspricht, indem Sie für jedes Dokument der `LinkedDocumentCollection` eine passend formatierte Zeile an einen String anhängen. Dieser String ist anschließend der Wert des Platzhalters im Template der Antwortseite. Beachten Sie, dass auf die Dokumente geklickt werden können soll, um den Inhalt der Dokumente zu erhalten; dazu haben wir oben das ``-Tag kennengelernt. Bauen Sie aus der Tabelle und dem oben gezeigten HTML-Code ein HTML-Dokument, welches Sie als *Body* Ihrer HTTP-Response verwenden können.

Implementieren Sie außerdem die Methode

```
private static HttpServletResponse handleFileRequest(String fileName),
```

die eine Antwort auf eine Anfrage nach einem Dokument behandelt. Der Inhalt des Dokuments (dieser befindet sich, wie auf Blatt 7 beschrieben, in der zweiten Zeile der Datei) soll dabei einfach als Body der `HttpServletResponse` verwendet werden.

Implementieren Sie schließlich die Logik des Webserver selbst, der wiederholt auf Anfragen von Clients wartet und diese passend beantwortet. Beachten Sie, dass Ihr Server bei fehlerhaften Anfragen nicht abstürzen soll; stattdessen soll er mit einem passenden HTTP-Fehler reagieren, indem er den Fehler an den Client sendet, bei welchem der Fehler dann im Browser angezeigt wird. Alle Fehler, die Sie hier verwenden sollen, finden Sie in der oben gegebenen Enumeration `HttpStatus`.

Aufgabe 10.8 (H) Noch mehr Streams

[5 Punkte]

In Aufgabe 9.1 (P) haben Sie mit Hilfe von Lambda-Funktionen und der Java Streams API die Temperaturen der letzten ca. 10 Jahre an einem einzigen Ort analysiert. Für diese Aufgabe stellen wir Ihnen einen Temperaturdatensatz von der Kaggle-Plattform⁶ zur Verfügung, welcher jährliche Temperaturmessungen von vielen Städten weltweit seit Mitte des 18. Jahrhunderts enthält. Das Ziel hierbei ist es, den weltweiten Temperaturtrend über eine längere Zeit anzuschauen und Merkmale herauszuarbeiten, welche diesen kompakt und objektiv zusammenfassen. Wir haben den Datensatz für diese Aufgabe jedoch auf Temperaturen von europäischen Städten beschränkt, welche in den letzten 50 Jahren gemessen wurden.

1. Kreieren Sie eine Klasse `Temperature` mit dem folgenden Konstruktor:

```
1 public Temperature(  
2     Date date,  
3     double averageTemperature,  
4     double averageTemperatureUncertainty,  
5     String city,  
6     String country,  
7     double latitude,  
8     double longitude  
9 )
```

Legen Sie die Attribute als unveränderliche `final`-Attribute ab und erstellen Sie alle getter-Methoden.

2. Mit diesem Hausaufgabenblatt erhalten Sie eine CSV-Datei ähnlich zu der, wie Sie sie bereits in Aufgabe 9.1 (P) kennengelernt haben. In jener Tutorialsaufgabe ist bereits Quelltext gegeben, welcher CSV-Dateien parsen kann. Parsen bedeutet in diesem Fall, dass ein Algorithmus die CSV-Datei, d.h. eine Komma-separierte Textdatei, öffnet, die Kopfzeile überspringt und Zeile für Zeile an den Kommas trennt und die String-Werte zwischen den Kommas in den entsprechenden Datentyp für die Attribute von `Temperature` umwandelt.

Außerdem wurde in Aufgabe 10.3 gezeigt, wie eine CSV-Datei von einem Webserver geladen werden kann. Davon werden wir später Gebrauch machen, um diese Datei herunterzuladen und direkt weiterzuverarbeiten.

⁶ <http://www2.in.tum.de/~kranzj/temperaturesEurope1963Till2013ByCity.csv>

Die abstrakte Klasse `Temperatures` soll nun nicht nur ein, sondern beliebig viele Temperaturobjekte verwalten. Implementieren Sie für diese Klasse das Interface `Iterable<Temperature>` mit einer Datenstruktur Ihrer Wahl. Die Klasse soll drei Konstruktoren besitzen:

- (a) `public Temperatures(Iterable<Temperature> temperatures)`,
welcher sich mit dem übergebenen `Iterable` von Temperaturobjekten initialisiert.
- (b) `public Temperatures(File csv)`,
welcher die übergebene CSV-Datei parst und in ein `Iterable` von `Temperature`-Objekten umwandelt.
Bei IO-Problemen, zum Beispiel Nicht-Existenz der Datei, soll das `Iterable` leer bleiben. Wenn in einer Zeile der CSV-Datei ein Attribut leer oder ungültig ist, so soll die Zeile übersprungen und keine Instanz von `Temperatur` angelegt werden.
- (c) `public Temperatures(java.net.URL url)`,
welcher die übergebene URL, d.h. eine Webseitenadresse, mit Hilfe des in Aufgabe 10.1 erstellten Netzwerk-Clients als IO-Stream öffnet und diesen während des Auslesens in eine Reihe von `Temperature`-Objekten umwandeln soll. Beachten Sie, dass dabei nichts auf der Festplatte zwischengespeichert werden soll. Auch hier gilt in ähnlicher Weise wie davor, dass bei Verbindungsproblemen das gesamte zu instanziiierende `Iterable` bei Problemen leer bleiben soll.

Aus Aufgabe 10.4 (P) kennen Sie zudem eine Lösung, um einen Stream mittels

```
public Stream<Temperature> stream()
```

für Objekte vom Typ `Temperature` anbieten zu können.

3. Erstellen Sie die folgenden abstrakten Methoden für die Klasse `Temperatures`, welche später durch Unterklassen zu implementieren sein werden:

- `public abstract long size()`
für die Anzahl der gemessenen Temperaturen,
- `public abstract List<Date> dates()`
für die Liste aller aufsteigend sortierten Tage ohne Duplikate von allen gemessenen Temperaturen,
- `public abstract Set<String> cities()`
für die Menge aller Städte von allen gemessenen Temperaturen,
- `public abstract Set<String> countries()`
für die Menge aller Länder von allen gemessenen Temperaturen,
- `public abstract Map<String, Temperatures> temperaturesByCountry()`
für die Menge aller gemessenen Temperaturen gruppiert nach Ländernamen,
- `public abstract String coldestCountryAbs()`
für das Land mit der kältesten gemessenen Temperatur,
- `public abstract String hottestCountryAbs()`
für das Land mit der heißesten gemessenen Temperatur,

- `public abstract String coldestCountryAvg()`
für das Land mit der kältesten durchschnittlichen Temperatur,
- `public abstract String hottestCountryAvg()`
für das Land mit der heißesten durchschnittlichen Temperatur,
- `public abstract Map<String, Double> countriesAvgTemperature()`
für Ländernamen plus deren dazugehörigen Durchschnittstemperatur über alle Zeiten und beinhaltenden Städte.

4. Programmieren Sie zwei Unterklassen von der abstrakten Klasse `Temperatures`, d.h. `IteratorTemperatures` und `StreamTemperatures`, welche jeweils die abstrakten Methoden implementieren sollen. Dabei sollen die Algorithmen in den Methoden der ersteren Klasse Iteratoren und `foreach`-Schleifen verwenden. Für die letztere Klasse soll durchgängig durch Aneinanderreihung von Stream-Operationen mit entsprechenden Lambda-Ausdrücken die Java Streams API verwendet werden.

Testen Sie jede dieser Methoden, indem Sie mit einer Methode

```
public void printSummary()
```

der abstrakten Oberklasse `Temperatures` die Ergebnisse aller Methoden außer `temperaturesByCountry()` und `dates()` in aussagekräftiger Art und Weise auf der Kommandozeile ausgeben. Zum Testen rufen Sie `printSummary()` aus einer für jede Unterklasse zu implementierenden `main`-Methode heraus auf.

Achten Sie bei der Implementierung der `main`-Methode zusätzlich bitte darauf, dass es einen Kommandozeilenparameter gibt. Dieser kann entweder die URL, d.h. die Webadresse, oder einen Dateipfad zu der zu parsenden CSV-Datei enthalten. Die `main`-Methode soll erkennen, ob eine gültige URL vorliegt, und, falls dem so ist, den URL-Konstruktor der `Temperatures`-Klasse aufrufen – oder den File-Konstruktor andernfalls. Dafür können Sie beispielsweise auf Exception-Handling beim Instanzieren von URL zurückgreifen.

5. Verwenden Sie die Streams API, um für jedes Land das mittlere jährliche Temperaturdelta seit 1900 bis zum letzten gemessenen Datum auszurechnen. Das Temperaturdelta zwischen zwei Jahren ist nichts anderes als die Temperaturdifferenz von der jährlichen Durchschnittstemperatur. Beispielsweise würde man die Durchschnittstemperatur im Jahr 1900 von der Durchschnittstemperatur im Jahr 1901 abziehen, um das Delta von 1900 auf 1901 für eine bestimmte Stadt zu erhalten. Der Durchschnitt all dieser Deltas ist das mittlere jährliche Temperaturdelta für eine Stadt. Der Durchschnitt aller Städte eines Landes ist der Durchschnitt eines Landes.

Implementieren Sie in der Klasse `StreamTemperatures` die Methode

```
public Map<String, Double> avgTemperatureDeltaPerYearPerCountry(),
```

welche eine Map von Ländernamen als String auf Deltas als Fließkommawerte zurückgibt. Diese Map soll zudem das globale Temperaturdelta als Durchschnitt über alle Länder im Key "Globally" enthalten.

Geben Sie in der `main`-Methode von `StreamTemperatures` diese Map gut leserlich auf der Kommandozeile aus.