Inhaltsverzeichnis

1. RDP-Aufgabe: Kommentare /* ... */ löschen....1
1.1. RDP-Aufgabe: TestCommentBlock.java....1
1.2. Die Theorie: deterministischer endlicher Automat....2
1.3. Der Lösungsansatz....3
1.3.1. Zustände, Zustandsübergänge (Ereignis/Aktivität)....3
1.3.2. Zustandsdiagramm, Automatengraph....3
1.3.3. Programm-Ablauf....4
1.3.4. Automaten-Tabelle....4
1.3.5. Hinweise zur Lösung....5
1.3.6. Abgabe: TestCommentBlock.java....6
1.4. +Aufgabe: Zeilenkommentare....7

1. RDP-Aufgabe: Kommentare /* ... */ löschen

☑ Quelle: http://sol.cs.hm.edu/dpunkt-java-praktikum/

1.1. RDP-Aufgabe: TestCommentBlock.java

Schreiben Sie das Programm **TestCommentBlock.java**, das
1. einen Java-Quelltext von der einer Datei (***CommentBlock_UE.java***) liest und
2. daraus Kommentare (/* ... */) entfernt und den Rest wieder ausgibt.

Zur Vereinfachung können Sie davon ausgehen, dass im Quelltext
- keine Zeilenkommentare(//),
- keine Textkonstante bzw. Zeichenkonstanten vorkommen.

Beachten Sie, dass Kommentare nicht ersatzlos gelöscht werden dürfen. Sie haben im Quelltext die Funktion von Zwischenraum. D.h. statt des /* ... */ muss ein Leerzeichen eingefügt werden.

Hier die Datei CommentBlock_UE.java:

```
public class CommentBlock_UE {
	public /* comment */static/**/void main(String[] args) {
		System.out.println(5/5);
		System.out.println(5/* comment ***/5);
		System.out.println(5/*/* //** comment */*1);
		System.out.println(5/* comment * */-4);
		System.out.println(5/* comment * /*/-4);
	}
}
```

Ergebnis: Nach dem Löschen des Kommentars:

```
public class CommentBlock_UE {
	public  static void main(String[] args) {
		System.out.println(5/5);
		System.out.println(5 /5);
		System.out.println(5 *1);
		System.out.println(5 -4);
		System.out.println(5 -4);
	}
}
```

Hier das Programm: **TestCommentBlock.java:**

```
// javac  TestCommentBlock.java
// java  TestCommentBlock

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class TestCommentBlock {
	public static void main(String[] args) throws IOException {
		BufferedReader in= new BufferedReader(
				new FileReader("CommentBlock_UE.java"));

		String withComment="";
		String line;
		String withoutComment;

		while((line= in.readLine()) !=null)
			withComment+= line + '\n';
		in.close();

		System.out.println("-- with comment --------------------------");
		System.out.println(withComment);

		withoutComment= delCommentBlock(withComment);
		System.out.println("-- without comment -----------------------");
		System.out.println(withoutComment);
	}

	public static String delCommentBlock(String sIn){
		String sRet="";
		//
		// ENTER CODE HERE
		//
	}
}
```

1.2. Die Theorie: deterministischer endlicher Automat

http://de.wikipedia.org/wiki/Zustandsdiagramm_(UML)
http://www.uml.ac.at/wp-content/uploads/teaching/04_Zustandsdiagramm_Folien.pdf

Zur Lösung der obigen Aufgabe kann ein sog. **deterministischer endlicher Automat** helfen. Solche Automaten sind in der Informatik ein sehr nützliches Hilfsmittel und werden häufig gebraucht.

Zustände und **Zustandsübergänge** sind das Wesensmerkmal solcher Automaten.

Der Ablauf eines Systems/einer Aufgabe kann als eine **Folge von Zuständsübergängen** modelliert werden.

Wir betrachten dabei:

1. **Zustände**
2. **Zustandsübergänge** (Transitionen)
3. **Ereignisse**, die eine Transition auslösen

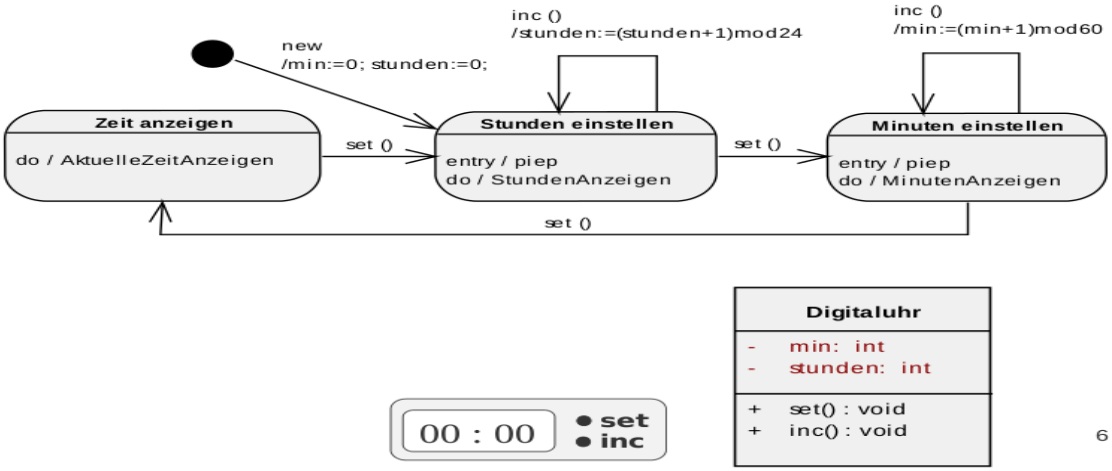
4. **Aktivitäten**, die

   innerhalb eines Zustands ausgeführt werden oder

   beim Zustandsübergang ausgeführt werden.

Beispiel: Digitaluhr

- 3 Zustände: Zeit anzeigen, Stunden einstellen, Minuten einstellen
- Zustandsübergänge: inc() gedrückt, set() gedrückt
- Initialzustand: Stunden einstellen



1.3. Der Lösungsansatz

1.3.1. Zustände, Zustandsübergänge (Ereignis/Aktivität)

In der gestellten Aufgabe kann man folg. definieren:

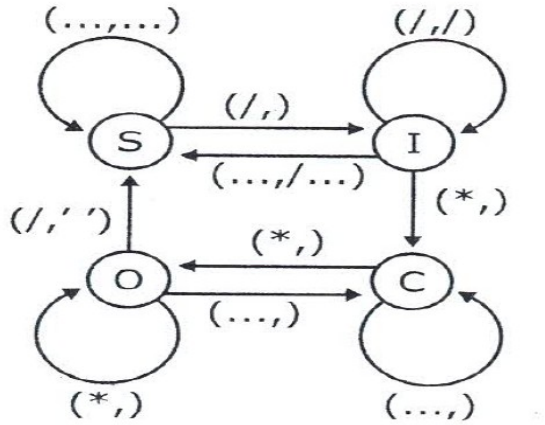
☑ **Zustände**:
- ☐ **Source:** im Quellcode und
- ☐ **Into Comment:** Kommentar-Start-Zeichen (/) erkannt.
- ☐ **Comment**: im Kommentar
- ☐ **OutOf Comment**: Kommentar-End-Zeichen (*) erkannt.

☑ **Ereignisse**
- ☐ Kommentar-Start-Zeichen (/) lesen
- ☐ beliebiges Zeichen (...) lesen
- ☐ Kommentar-End-Zeichen (*) lesen

Nun aber zur Lösung der oben gestellten Aufgabenstellung.

1.3.2. Zustandsdiagramm, Automatengraph

Für eine korrekte und robuste Lösung implementieren Sie einen Automaten, den der folgende **Automatengraph** darstellt:



☑ Die vier **Zustände** sind:
- ☐ **S...** Im Quelltext, außerhalb eines Blockkommentars (source). Ist der Initial-Zustand.
- ☐ **I ...** (evtl.) Anfang eines Kommentars (into)
- ☐ **C...** Innerhalb eines Kommentars (comment)
- ☐ **O...** (evtl.) Ende des Kommentars (outof)

☑ Die **Zustandsübergänge** werden durch **(eingabezeichen, ausgabezeichen)** beschriftet.
- ☐ (/ , ' ') Eingabezeichen ist ein / und Ausgabezeichen ist ein Leerzeichen.

☑ Die **Zeichen** sind:

/	... (evtl.) Bestandteil des Kommentarzeichen
*	... (evtl.) Bestandteil des Kommentarzeichen
...	... alle anderen Zeichen

1.3.3. Programm-Ablauf

- ☐ In jedem Schritt **liest** der Automat ein **Eingabezeichen** und
- ☐ **folgt** dabei der entsprechend beschrifteten **Kante zu einem Folgezustand**.

Dabei wird das angegebene Ausgabezeichen ausgegeben.

Die Rolle von Kommentaren als Zwischenraum lässt sich leicht implementieren, indem am Ende jedes Kommentars ein Leerzeichen ausgegeben wird.

1.3.4. Automaten-Tabelle

Der Automatengraph ist zwar anschaulich, lässt sich aber nicht sehr gut in ein Programm umsetzen.

Besser eignet sich dafür eine **Automaten-Tabelle**, die pro Zeile
den Ausgangszustand, das Ereignis, die Aktivität und den neuen Zustand enthält.

Ausgangs-ZUSTAND	**EREIGNIS** **g**elesenes Zeichen	**AKTIVITÄT**	**NEUER ZUSTAND**

Aus dem obigen Graphen ergibt sich die folgende Tabelle:

AUTOMATEN-TABELLE Ereignis: ein Zeichen lesen Zustände: S,I,C,O			
ZUSTAND	**EREIGNIS** **g**elesenes Zeichen	**AKTIVITÄT**	**NEUER ZUSTAND**
Source	/	nothing	**I**
	*	* ausgeben	**S**
	...	... ausgeben	**S**
Into	/	/ ausgeben	**I**
	*	nothing	**C**
	...	/ ... ausgeben	**S**
Comment	/	nothing	**C**
	*	nothing	**O**
	...	nothing	**C**
Outof	/	' ' ausgeben	**S**
	*	nothing	**O**
	...	nothing	**O**

1.3.5. Hinweise zur Lösung

Die Zustände bestehen aus einer Sammlung von verhältnismäßig wenigen Alternativen, die alle bekannt sind und sich nicht ändern. Dieses Kriterium spricht für eine Repräsentation durch einen Aufzählungstyp:

```
enum State {Source, Into, Comment, Outof};
```

Der Interpreter des Automaten liest die Eingabe Zeichen für Zeichen. In einer Variablen vom Typ State wird der aktuelle Automatenzustand gespeichert.

```
	public static String delCommentBlock(String sIn){
		String sRet="";
		State state = Source;

		for(int i= 0; i < sIn.length(); i++){
			char chr = sIn.charAt(i);
			//
			// ENTER CODE HERE
			//
	}
```

In der Eingabe-Schleife wird die **Automaten-Tabelle** durch eine **switch-Anweisung** realisiert.

```
...
switch(state) {
      case Source:      … break;
      case Into:        … break;
      case Comment:     … break;
      case Outof:       … break;
}
```

Diese Anweisung entspricht einer Selektion der Automaten-Tabellenzeilen.
In jeder Zeile wird nun die passende Spalte selektiert. Auch dafür können if-Kaskaden oder switch-Anweisungen eingesetzt werden. In der folgenden Codeskizze ist der Kürze wegen nur eine der geschachtelten switch-Anweisungen ausgeführt.

```
switch(state) {
      case Source:
            switch(chr) {
                  case '/':   … break;
                  case '*':   … break;
                  default :   … break;
            }
            break;
      case Into:        … break;
      case Comment:     … break;
      case Outof:             … break;
```

☑ Die Aktionen sind weitgehend regelmäßig aufgebaut: Zuerst werden (sofern nötig) Zeichen ausgegeben, dann wird der Folgezustand an die Variable state zugewiesen:

```
switch(state) {
      case Source:
            switch(chr) {
                  case '/':
                        // keine Ausgabe
                        state= Into;
                        break;

                  case '*':
                        sRet += chr;
                        // Zustand bleibt unverändert
                        break;

                  default:
                        sRet += chr;
                        // Zustand bleibt unverändert
                        break;
            }
            break;

      case Into:        … break;
      case Comment:     … break;
      case Outof:       … break;
```

1.3.6. Abgabe: TestCommentBlock.java

Zum Laufen bringen.

1.4. +Aufgabe: Zeilenkommentare

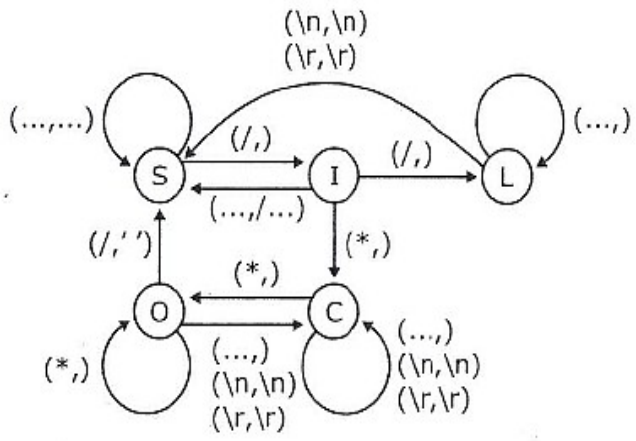
Zeilenkommentare beginnen mit // und enden mit der Zeile. Sie wurden bisher nicht berücksichtigt.

Entwickeln Sie ein Programm CommentBlockLine, das sowohl Block- wie auch Zeilenkommentare entfernt.

Zeilenwechsel werden mit Kontrollzeichen codiert, die sich von System zu System unterscheiden. Als einfache Lösung reicht es aus, die beiden Zeichen '\n' und '\r' gleichermaßen als Zeilenwechsel zu behandeln.

Oft soll die Zeilenstruktur des Quelltextes beibehalten werden. Sorgen Sie dafür, dass Zeilenwechsel auch innerhalb von Blockkommentaren erhalten bleiben. Das bedeutet, dass Blockkommentare durch die entsprechende Anzahl Leerzeilen ersetzt werden.

Hinweis:
Der Automat der obigen Programmversion wird um den Zustand L (line comment, ganz rechts) erweitert:



Näher an einer Implementierung liegt wieder die entsprechende Automatentabelle, die um eine neue Zeile (für L = line comment) und eine neue Spalte (für die NewLine-Zeichen) erweitert ist:

	/	*	**NewLine**	**alle anderen**
S	I	S, ...	S, ...	S, ...
I	L	C	S, / ...	S, / ...
C	C	0	C, ...	C
0	S, ' '	0	C, ...	C
L	L	L	S, ...	L

Der Code zur Umsetzung dieser Tabelle folgt dem gleichen Muster wie die vorhergehende Lösung und wird hier nicht mehr ausgeführt. Wir wollen eine andere Methode zur Abarbeitung von endlichen Automaten kennen lernen.