Textfiles einlesen & strukturieren

Lernziele:

- Sie können in einem Programm aus einem File Daten lesen
- Strukturierte Textzeilen zerlegen und diese in geeignete Datenobjekte abfüllen
- Ausgaben in Textfiles durch Formatierung strukturieren

1 BufferedReader

Wir werden uns hier zunächst nur mit Text Files beschäftigen, welche eine Zeilenstruktur aufweisen, in denen also Informationseinheiten durch einen Zeilenumbruch voneinander getrennt sind.

Die Klasse aus der Java-Library, welche dafür verwendet werden kann, ist *BufferedReader*. Diese Klasse enthält im Wesentlichen die Methode

```
public String readLine()
```

Diese Methode liefert uns immer jeweils die nächste ungelesene Zeile von der aktuellen Quelle des Readers, oder null, wenn keine Zeilen mehr vorhanden sind.

Nehmen Sie an, Sie haben in der Variablen `br` ein BufferedReader-Objekt. Wie könnte eine Schleife aussehen, welche alle Zeilen einliest und auf der Konsole ausgibt?

Erstellen eines BufferedReader

Um einen *BufferedReader* zu erstellen, müssen wir ihm angeben, von welcher Quelle er lesen soll. In unserem Fall, wo wir Text-Files lesen wollen, ist die Quelle ein *FileReader*. Um diesen zu erstellen, brauchen wir dann den Pfad des zu lesenden Files.

```
String fileName = ....

FileReader fr = new FileReader(fileName);
```

Sie werden beim Verwenden des FileReaders auch eine entsprechende *Exception* benötigen. Setzen Sie ihren Code in einen *try-catch Block* (Exceptions werden ausführlich im Modul 226 behandelt).

Wenn wir den *FileReader* haben, können wir ihn dem zu erstellenden BufferedReader als Quelle angeben:

```
BufferedReader br = new BufferedReader(fr);
```

2 Buffered Reader oder Scanner?

Nebst dem *BufferedReader* kann man auch die *Scanner* Klasse verwenden. Die Effizienz beim Einlesen von einem File ist ungefähr dieselbe. Und die *Scanner* Klasse hat auch mehr Methoden, um das Parsen (das „Strukturieren" eines Input-Streams) besser zu bewältigen. Jedoch ist *Scanner* nicht „thread safe", d.h. es muss von aussen „synchronised" werden (z.Bsp. um zu verhindern, dass ein geändertes File das Lesen stört).

Ein *Scanner* ist sicher die einfachere Variante, um einen User-Input via Konsole zu handhaben. Siehe auch die *InputReader* Klasse.

Sie können gerne beide Varianten für das Einlesen von Files ausprobieren.

Hier sind zwei Beispiele:

Mit Scanner

```
try {
  String pathName = “”; //set path to a file!
  File file = new File(pathName);
  Scanner input = new Scanner(file);
  while (input.hasNextLine()) {
    //zeilenweise einlesen
    String line = input.nextLine();
    System.out.println(line);
  }
  input.close();
}

catch (Exception ex) {
  ex.printStackTrace();
}
```

Mit BufferedReader

```
BufferedReader br = null //initialise
try {
String inputLine;
String filepath = ""; //set path to a file!!
FileReader fr = new FileReader(filepath);
BufferedReader br = new BufferedReader(fr);

while ((inputLine = br.readLine()) != null)
{
//Achtung: nicht nochmals einlesen

System.out.println(inputLine);
}

} catch (IOException e) {
   e.printStackTrace();

} finally {
   try {
  if (br != null) {
br.close();
```

	}
	
	} catch (IOException ex) {
	
	ex.printStackTrace();
	}
	}

Mit dem *Scanner* können Sie direkt ein *File*-Objekt verwenden, der *BufferedReader* verwendet einen *FileReader*. Bei beiden Varianten wird zeilenweise das File eingelesen als String Objekt. In beiden Varianten wird das Lesen am Schluss beendet mit der Methode `close`().

3 Strukturierte Textfiles

Oft werden Daten in strukturierten Text Files abgelegt. So können zum Beispiel Excel-Tabellen im CSV-Format gespeichert werden, so dass jede Zeile aus Excel eine Zeile im Text File ergibt. Innerhalb der Zeilen sind die Spalten durch ein spezielles Trennzeichen getrennt (CSV steht für Comma Separated Values, wobei das Komma eher selten als Trennzeichen verwendet wird).

Ein strukturiertes File mit Personendaten könnte also so aussehen:

```
Huber;Peter;1955
Meier;Irene;1960
Muster;Anna;1974
```

Um nun, nachdem eine Zeile gelesen wurde, auf die einzelnen Felder zugreifen zu können, muss der String, welcher eine Zeile enthält, zerlegt werden. Das ist sehr ähnlich wie Sie es schon im Modul 404 gesehen haben:

```
String line = ...                    // contains one line which  must be processed
String[] lineParts = line.split(";"); // Split line at occurences of semicolon

for ( String p: lineParts ) {         // loop over all parts of the line     // Do
something with this part (Variable p) }
```

Nummerische Felder

Wenn ein Feld eine Zahl enthält (im Beispiel das Geburtsjahr), dann wird das natürlich zunächst einmal als String eingelesen, wie alle anderen Felder auch. Solange wir damit nicht rechnen wollen, spielt das auch keine Rolle.

Meistens ist aber nötig, diese Felder in richtige Zahlen umzuwandeln. Dazu können die beiden folgenden Aufrufe verwendet werden:

```
String intStr;     // contains the String representation of an int

String doubleStr;  // contains the String representation of a double int intVal =
Integer.parseInt(intStr);             // Converts a String to an int value double
doubleVal = Double.parseDouble(doubleStr);  // Converts a String to a double value
```

4 Übungen Textfiles

Wir werden ein Textfile mit folgender Struktur verwenden:

```
Huber;Peter;1955

Meier;Irene;1960

Muster;Anna;1974
```

Übung

Erstellen Sie ein Programm, welches dieses File liest und Zeile für Zeile wieder auf der Konsole ausgibt, ergänzt um das Alter der Person, bezogen auf das aktuelle Jahr welches fest einprogrammiert sein darf.

Erweiterung

Erweitern Sie das Programm so, dass es den Benutzer nach einem Vornamen fragt und dann von allen Personen mit diesem Vornamen den ganzen Namen und das Alter auf der Konsole ausgibt.

Beachten Sie dabei, dass zum Vergleich von zwei String-Variablen nicht der Operator ==verwendet werden kann. Stattdessen muss die Methode `equals` der Klasse String verwendet werden.

5 Daten strukturiert in ein File schreiben

Die Klasse, welche am geeignetsten ist, um Text Files zu schreiben, ist die *PrintWriter*. Ein *PrintWriter* kann erzeugt werden, indem man ihm beim Erstellen einen Filenamen übergibt.

Die Methoden sind dabei sehr ähnlich wie beim *PrintStream* (den Sie z.Bsp. bei `System.out.println` verwenden):

```
print
println
format
```

Natürlich können Sie auch die Klasse *FileWriter* verwenden. Die Unterschiede sind nicht sehr gross, jedoch hat *PrintWriter* mehr Methoden für die Formatierung zur Verfügung.

Übung

Schreiben Sie eine Klasse *MyWriter*, welcher einen User-Input umgekehrt in ein File schreibt.

Quelle: BBW Unterlagen, St.Dütsch. Aktualisiert August 2020.