Next Up Previous
Next: JAVA Container **Up:** JAVA Standard-Bibliotheken **Previous:** Exceptions

I/O in JAVA

In JAVA wird jeglicher I/O, egal ob über Tastatur, Files oder Netzwerk, in Form von *Datenströmen* (streams) durchgeführt. JAVA stellt verschiedene Klassen und Objekte für Datenströme zur Verfügung, die je nach der Anwendung entsprechend kombiniert werden müssen.

- Einfache, grundlegende Stream-Klassen definieren auf einer niedrigen Ebene die Ein- und Ausgabe in Dateien (Files) oder auch in anderen Datenstreams wie z.B. Pipes, Byte-Arrays oder Netzverbindungen (Sockets, URLs).
- Spezielle, darauf aufbauende Stream-Klassen definieren zusätzliche Eigenschaften wie z.B. Pufferung, zeilenorientierte Text-Ausgabe, etc.
- Streams können `geschachtelt' werden
  `TextStream(BufferedStream(RawStream("file.txt")))`

Zwei Gruppen von Datenstreams

- InputStream, OutputStream und RandomAccessFile für byte-orientierte Datenfiles, d.h. I/O im JAVA spezifischen Binärformat, i.a. not human-readable
- Reader und Writer für zeichen- und zeilen-orientierte Textfiles.

Ein Sonderfall sind die streams `System.out, System.err, System.in` die statisch in JAVA definiert sind und immer zur Verfügung stehen, ansonsten muss das I/O package geladen werden:

```
import java.io.*;
```

Ausser bei `System.out, System.err` wird generell ein **IOException** handling mittels **try-catch** erzwungen.

Im folgenden Beschränkung auf die Benutzung der Reader und Writer Klassen für zeichen-orientierte Textfiles.

Einlesen - Reader-Methoden

- Input Stream anlegen (File öffnen) z.B. mit
  `FileReader in = new FileReader ("filename.txt");`
  oder besser
  `BufferedReader in = new BufferedReader ( new FileReader ("filename.txt") );`

- `int ch = in.read()` liefert ein Zeichen
- `int nch = in.read(char[])` liefert soviele Zeichen wie möglich in einen char Array. In beiden Fällen wird `-1` zurückgegeben wenn das Ende des Datenstroms (End-of-File) erreicht wurde.

- `String s = in.readLine()` liest ganze Zeile in String, i.a. beste Methode zum Einlesen von Textfiles, benötigt `BufferedReader()`

- `in.close()` schliesst den Eingabestrom bzw. das File.

- Alles muss innerhalb von `try { ... }` Blöcken ablaufen mit anschliessenden `catch (IOException e){ ... }`.
- Keine direkte Möglichkeit zum Einlesen anderer Datentypen (int, double, ...)
  ⇒ Konvertierung aus String muss selbst gemacht werden mittels
  `Integer.parseInt(), Double.parseDouble(), ...`

Zeichenweise Lesen

```
int ch;
 try {
     BufferedReader in = new BufferedReader (
                             new FileReader ("filename.txt") );
     try {
         while( (ch = in.read()) != -1 ) {
             // do something ...
         }
         in.close();
     } catch (IOException e) {
         System.out.println("Read error " + e);
     }
 }
 catch (IOException e) {
     System.out.println("Open error " + e);
 }
```

Zeilenweise Lesen

```
String thisLine;
try {
    BufferedReader in = new BufferedReader (
                    new FileReader ("filename.txt") );
    try {
        while( (thisLine = in.readLine()) != null ) {
            // do something ...
        }
        in.close();
    } catch (IOException e) {
        System.out.println("Read error " + e);
    }
}
catch (IOException e) {
    System.out.println("Open error " + e);
}
```

Ausgeben - Writer-Methoden

Im Prinzip analog zum Reader:

- Output Stream anlegen (File öffnen) z.B. mit
  `FileWriter out = new FileWriter ("filename.txt");`
  oder besser
  `BufferedWriter bufout = new BufferedWriter ( new FileWriter ("filename.txt") );`

- `out.write(int)` schreibt ein Zeichen
- `out.write(char[])` schreibt alle Zeichen in einem char Array
- `out.newLine()` schreibt ein Zeilenende-Zeichen (Anmerkung: plattformabhängig `"\n"` oder `"\r\n"`)

- `out.close()` schliesst den Ausgabestrom bzw. das File.

- Alles muss innerhalb von `try { ... }` Blöcken ablaufen mit anschliessenden `catch (Exception e){ ... }`.

---

oder etwas komfortabler die PrintWriter-Methoden

- Output Stream anlegen (File öffnen) z.B. mit
  `PrintWriter out = new PrintWriter(new FileWriter("filename.txt"));`
- Ausgabe dann mit `print(Typ)` oder `println(Typ)`
  wobei `Typ` alle gängigen Datentypen sind sowie alle Objekte mit `toString()`
  ⇒ bekannt aus `System.out.println()`

Formatierte Gleitkommazahlen kann man mit der DecimalFormat Klasse aus java.text erreichen:

```
     import java.text.*;
     DecimalFormat dec1 = new DecimalFormat ("####0.00");
     DecimalFormat dec2 = new DecimalFormat ("#0.00000");
     DecimalFormat dec3 = new DecimalFormat ("0.00E000");
     double x = 599.1094876;
//     dec1.format(x) -> "599.11"
//     dec2.format(x) -> "599.09488"
//     dec3.format(x) -> "5.99E002"
```

Zeichenweise Schreiben

```
try {
     BufferedWriter out = new BufferedWriter (
                  new FileWriter ("filename.txt") );
     try {
          while ( ) {
               out.write(...);
               out.newLine();
          }
          out.close();
     } catch (IOException e) {
          System.out.println("Write error " + e);
     }
```

```
} catch (IOException e) {
    System.out.println("Open error " + e);
}
```

Zeilenweise Schreiben

```
try {
    PrintWriter out = new PrintWriter (
                new FileWriter ("filename.txt") );
    try {
        while ( ) {
            out.println(...);
        }
        out.close();
    } catch (IOException e) {
        System.out.println("Write error " + e);
    }
} catch (IOException e) {
    System.out.println("Open error " + e);
}
```

I/O in Java 5

Wie schon in Kapitel 1 diskutiert wurden in JAVA 5 wesentlich einfacher zu handhabende I/O Funktionen eingeführt.

Input aus Datei mittels Scanner Klasse (aus java.util)

- Scanner-Objekt erzeugen, mit File-Objekt als Argument: `Scanner s = new Scanner(new File("somefile.dat");`
  Für diese Operation Exception-handling nötig, d.h. **try-catch** Blöcke
- Dann Daten einlesen mit passenden `s.nextXXX` Methoden bzw. Testen mit `s.hasNextXXX`

```
import java.util.*;
import java.io.*;
public class ReadIn {
    public static void main(String[] args)
    {
    Scanner s;
    try {
         s = new Scanner(new File("some.txt"));
    }
    catch (IOException e) {
        System.out.println("Open error " + e);
        return;
    }
    while ( s.hasNextLine() ) {
        String str = s.nextLine();
        System.out.println( str );
    }
    s.close();
    }
}
```

Ausgabe in Datei: Wie oben diskutiert mit dem `PrintWriter` und `printf` Funktion für einfache formatierte Ausgabe.

Oder mit der **Formatter** Klasse, dem Gegenstück der Scanner-Klasse zur Ausgabe:

```
import java.util.*;
import java.io.*;
public class ReadWrite {
    public static void main(String[] args)
    {
    String infile = "some.txt", outfile = "newsome.txt";
        if ( args.length > 0 ) { // input file 1st arg
        infile = args[0];
    }
        if ( args.length > 1 ) { // output file 2nd arg
        outfile = args[1];
    }
    try {
        Scanner s = new Scanner(new File(infile)); // open inp
        Formatter f = new Formatter(new File(outfile)); // open outp
        int linenum = 0;
        while ( s.hasNextLine() ) {
        linenum ++;
        String str = s.nextLine();
        f.format("%4d %s %n", linenum, str);  // output, line-number 1st
        }
        s.close();
        f.close();
    }
    catch (IOException e) {
        System.out.println("Open error " + e);
        return;
    }
    }
}
```

I/O übers Netz

Im Package java.net gibt es eine Vielzahl von Klassen zur Kommunikation über das weltweite Internet oder interne Intranets und Extranets mit den Internet-Protokollen TCP und UDP.

Leider keine Zeit für genauere Behandlung, nur ein kleines Beispiel.

Mit der Klasse **URL** kann man Verbindungen zu den Standard-Internet Services machen, z.B.
`URL("http://www.spiegel.de")` oder
`URL("ftp://ftp.uu.net/")`
Anwenden der Methode `URL.openstream()` öffnet die Verbindung für einen byte-Stream.

Byte-streams leider nicht behandelt, aber es genügt die Konverter-Klasse `InputStreamReader` zu kennen um es mit einem `BufferedReader` und `readLine()` zu lesen.

D.h. mit wenigen Zeilen kann man jede beliebige WWW page lesen:

```
import java.io.*;
import java.net.*;
 ...
 try {
     BufferedReader in = new BufferedReader
         (new InputStreamReader
             ( ( new URL(args[0]) ).openStream()) );
```

Exception handling

Jede I/O Operation kann im Prinzip IOExceptions auslösen und **muss** in **try-catch** Block gemacht werden.

Mehrere Möglichkeiten:

Geschachtelte **try-catch** Blöcke:

```
try {
     open ...
     try {
         while ( read ... ) {}
         close ...
     } catch (IOException e) {
        ...
     }
 } catch (IOException e) {
     ...
 }
```

Alles in einem Block:

```
try {
     open ...
     while ( read ... ) {}
     close ...
 } catch (IOException e) {
     ...
 }
```

Sequentielle **try-catch** Blöcke:

```
try {
     open ...
 } catch (IOException e) {
     ...
 }
 try {
```

```
    while ( read ... ) {}
} catch (IOException e) {
    ...
}
try {
    close ...
} catch (IOException e) {
    ...
}
```

Oder schliesslich noch die exception einfach weitergeben:

Dazu muss Funktion, in der der I/O ausgeführt wird, mit `throws IOException` definiert werden

---

```
public void readMyFile throws IOException  {
    open ...
    while ( read ... ) {...}
    close ...
}
```

Dann keine **try-catch** Blöcke nötig, Verantwortung wird an das *rufende* Programm abgeschoben
⇒ i.a. nicht zu empfehlen.

Next Up Previous
Next: JAVA Container **Up:** JAVA Standard-Bibliotheken **Previous:** Exceptions
GDuckeck 2016-03-11