Informatik B

Vorlesung 5

Files, IO

Rückblick

- Assertions
- **`Throwable`**
  - **`Exceptions`**
  - **`Error`**
  - **`RuntimeException`**
- Stack Trace

Die Arbeit mit Dateien

- Um an die Information einer Datei zu gelangen ist es notwendig folgende Operationen durchführen zu können:
  - Dateien anzulegen
  - Dateien löschen
  - Dateien umbenennen
  - Verzeichnisse verwalten
  - Auslesen von Dateien
  - Schreiben von Dateien

`File`

- Das **`File`**-Objekt wurde eingeführt, um Dateioperationen plattformunabhängig durchzuführen
- Probleme gibt es bei der Rechteverwaltung da die Betriebssysteme unterschiedliche Ansätze zur Rechteverwaltung besitzen
- Ein **`File`**-Objekt repräsentiert einen Datei- oder Verzeichnisnamen im Dateisystem
- Der Verweis wird durch einen absoluten oder relativen Pfadnamen spezifiziert

File

Erzeugen eines **File**-Objektes:

- **File(String pathname)** Erzeugt ein **File**-Objekt aus einem Dateinamen.
- **File(String parent, String child)**
- **File(File parent, String child)** Setzt ein neues **File**-Objekt aus einem Basisverzeichnis und einem weiteren Teil zusammen, der auch wieder ein Verzeichnis oder ein Dateiname sein kann.
- **File(URI uri)** Fragt von **uri** den Pfadnamen (**uri.getPath()**) und erzeugt ein neues **File**-Objekt
- Der Pfad ist plattformabhängig
  - Unter Windows trennt ein Backslash „\“ die Pfade
  - Unter Unix trennt ein Slash „/ “ die Pfade
  - Die Klasse File speichert in einer Klassenvariable **separatorChar** den Pfadtrenner
- Die Darstellung des Wurzelverzeichnisses ist ebenfalls unterschiedlich
  - Unter Unix ist dies ein einzelner Slash „/ “
  - Unter Windows ist die Angabe des Laufwerks vor den Doppelpunkt und das Backslash-Zeichen gestellt („Z:\“)
  - Mit der Klassenmethode **File.listRoots()** können die Wurzelverzeichnisse ausgegeben werden

File

- Das **File**-Objekt muss nicht unbedingt eine existierende Datei oder ein existierendes Verzeichnis repräsentieren
- **exists()** testet, ob die Datei oder das Verzeichnis tatsächlich vorhanden ist
- Da ein **File**-Objekt Dateien sowie Verzeichnisse gleichzeitig repräsentiert, ermöglichen **isDirectory()** und **isFile()** eine genauere Aussage über den **File**-Typ
- Es kann passieren, dass für ein **File**-Objekt weder **isDirectory()** noch **isFile()** die Rückgabe **true** liefern (In Java können nur normale Dateien erzeugt werden.)
- Beispiel: **filetest1**

File

Datei/Verzeichnisattribute

- Eine Datei oder ein Verzeichnis besitzt zahlreiche Eigenschaften, die sich mit Anfragemethoden auslesen und teilweise auch ändern lassen:
  - **`boolean canExecute()`** (Java6), **`canRead(), canWrite()`** liefern **`true`**, wenn die Ausführungsrechte/Leserechte/Schreibrechte gesetzt sind
  - **`long length()`** Gibt die Länge der Datei in Byte zurück oder 0L, wenn die Datei nicht existiert oder es sich um ein Verzeichnis handelt
  - **`long lastModified()`** Liefert den Zeitpunkt, zu dem die Datei zum letzten Mal geändert wurde. Die Zeit wird in Millisekunden ab dem 1. Januar 1970, 00:00:00 UTC, gemessen. Die Methode liefert 0, wenn die Datei nicht existiert oder ein Ein-/Ausgabefehler auftritt
  - **`boolean setLastModified(long time)`** Setzt die Zeit (wann die Datei zuletzt geändert wurde). Die Zeit ist wiederum in Millisekunden seit dem 1. Januar 1970 angegeben. Ist das Argument negativ, dann wird eine **`IllegalArgumentException`** ausgelöst

File

- Dateiattribute:
  - **`boolean setReadOnly()`** Setzt die Datei auf *readonly*. Liefert **`true`** wenn die Änderung möglich war
- Java6
  - **`boolean setExecutable(boolean executable)`**
  - **`boolean setExecutable(boolean executable, boolean ownerOnly)`**
  - **`boolean setReadable(boolean readable)`**
  - **`boolean setReadable(boolean readable, boolean ownerOnly)`**
  - **`boolean setWritable(boolean writable)`**
  - **`boolean setWritable(boolean writable, boolean ownerOnly)`**

File

Freier Plattenspeicher

- In Java 6 sind Methoden zum Ermitteln des freien Plattenspeichers hinzugekommen:
  - **`getFreeSpace()`**
  - **`getUsableSpace()`**
  - **`getTotalSpace()`**
  - Beispiel: **`filetest2`**

File

Umbenennen und Verzeichnisse anlegen

- **`boolean mkdir()`** Legt das Unterverzeichnis an
- **`boolean mkdirs()`** Legt das Unterverzeichnis inklusive weiterer ev. benötigter Verzeichnisse an
- **`boolean renameTo( File d )`** Benennt die Datei in den Namen um, der durch das **`File`**-Objekt **`d`** gegeben ist
- Achtung:
  - **`File`**-Objekte sind *immutable*, stehen also immer nur für genau eine Datei
  - Ändert sich der Dateiname, ist das **`File`**-Objekt ungültig und es ist kein Zugriff mehr über dieses **`File`**-Objekt erlaubt
  - Auch wenn eine Laufzeitumgebung keine Exception auslöst, sind alle folgenden Ergebnisse von Anfragen unsinnig.
- Zum Teil kann **`renameTo()`** auch zum Verschieben von Dateien dienen. Oft gilt aber die Einschränkung, dass dies nur auf demselben Datenträger möglich ist (also etwa von Laufwerk C nach C:/, aber nicht von C nach D:/)
- Beispiel: **`filetest3`**

File

Verzeichnisse listen

- Ein Verzeichnis kann reine Dateien oder auch wieder Unterverzeichnisse besitzen
- Die **`list()`**- und **`listFiles()`**-Methoden der Klasse File geben ein Feld von Zeichenketten mit Dateien und Verzeichnissen beziehungsweise ein Feld von **`File`**-Objekten mit den beinhalteten Elementen zurück
- Die Methode **`list()`** liefert dabei nur relative Pfade, also einfach den Dateinamen oder den Verzeichnisnamen, der absolute Name zu einer Dateiquelle muss erst zusammengesetzt werden
- Die Methode **`listFiles()`** liefert komplette **`File`**-Objekte, die ihre ganze Pfadangabe schon kennen

File

Dateien anlegen

- **`boolean createNewFile() throws IOException`**
  - Legt eine neue, leere Datei mit dem im **`File`**-Objekt gespeicherten Namen an, insofern eine Datei mit diesem Namen noch nicht existiert.
- **`static File createTempFile(String prefix, String suffix) throws IOException`**
  - Legt eine neue Datei im temporären Verzeichnis an
  - Der Dateiname setzt sich aus einem benutzerdefinierten Präfix, einer Zufallsfolge und einem Suffix zusammen
- **`static File createTempFile( String prefix, String suffix, File directory ) throws IOException`**
  - Legt eine neue Datei im gewünschten Verzeichnis an
  - Der Dateiname setzt sich aus einem benutzerdefinierten Präfix, einer Zufallsfolge und einem Suffix zusammen

File

Dateien und Verzeichnisse löschen

- **`boolean delete()`**
  - Löscht die Datei oder das **leere** Verzeichnis
- **`void deleteOnExit()`**
  - Löscht die Datei/das Verzeichnis, wenn die virtuelle Maschine korrekt beendet wird
  - Einmal vorgeschlagen, kann das Löschen nicht mehr rückgängig gemacht werden
  - Falls die JVM fehlerhaft beendet wird (z.B. Stromausfall), kann die Datei möglicherweise noch vorhanden sein, was insbesondere für temporäre Dateien lästig ist
- Beispiel: **`filetest4`**

Datenströme

- In Java erfolg der Zugriff auf Dateiinhalte mit Datenströmen (engl. *stream*)
- Mittels Datenströmen können Daten sehr elegant bewegt werden
- Eingabeströme (engl. *input streams*) sind zum Beispiel Daten der Tastatur oder vom Netzwerk
- Ausgabeströme (engl. *output streams*) fließen in ein Ausgabemedium, beispielsweise in den Drucker oder in eine Datei
- Streams sind immer unidirektional
- In Java sind über dreißig Klassen zur Verarbeitung der Datenströme vorgesehen
- Da die Datenströme an kein spezielles Ein- oder Ausgabeobjekt gebunden sind, können sie beliebig miteinander gemischt werden
- Die Schachtelung von Streams erlaubt die Konstruktion von **Filtern**, die bei der Ein- oder Ausgabe bestimmte Zusatzfunktionen übernehmen können

Datenströme

- Klassen zum Lesen und Schreiben von Binär- und Zeichendaten sind im Package **`java.io zu finden`**
- Für die byte-orientierte Verarbeitung, etwa von PDF- oder MP3-Dateien, gibt es andere Klassen als für Textdokumente
- Die zeichenorientierten Klassen nennen sich **`Reader`**, **`Writer`** und die byte-orientierten Klassen **`InputStream`** und **`OutputStream`**
- Die Unterscheidung ist notwendig, da Java auf Unicode basiert und die Datentypen byte und char einen unterschiedlichen Wertebereich besitzen (**`char`**: 16 Bit, **`byte:`** 8 Bit)
- Die Basisklassen geben abstrakte **`read()`**- oder **`write()`**-Methoden vor
- Die Unterklassen überschreiben die abstrakten Methoden, da nur sie wissen, wie etwas tatsächlich gelesen oder geschrieben wird

IO-Basisklassen

- Konkrete Eingabe/Ausgabe-Klassen wie **`FileInputStream`**, **`FileOutputStream`**, **`FileWriter`** oder **`BufferedWriter`** erweitern abstrakte Oberklassen
- Im Allgemeinen können vier Kategorien gebildet werden:
  - Klassen zur Ein-/Ausgabe von Bytes (oder Byte-Arrays)
  - Klassen zur Ein-/Ausgabe von Unicode-Zeichen (Arrays oder Strings)

Basisklasse für	Bytes (oder Byte-Arrays)	Zeichen (oder Zeichen-Arrays)
Eingabe	**`InputStream`**	**`Reader`**
Ausgabe	**`OutputStream`**	**`Writer`**

- Für die Umwandlung zwischen Byte- und Zeichenstreams gibt es so genannte Brückenklassen:
  - **`InputStreamReader (byte`** nach **`char)`**
  - **`OutputStreamWriter (char`** nach **`byte)`**

`Closeable` und `Flushable`

- **`Closeable`** wird von allen lesenden und schreibenden Datenstrom-Klassen implementiert, die geschlossen werden können
- Das sind alle **`Reader/Writer-`** und **`InputStream/OutputStream`**-Klassen
- **`void close() throws IOException`** schließt den Datenstrom. Einen geschlossenen Strom noch einmal zu schließen hat keine Konsequenz
- **`Flushable`** findet wird nur bei schreibenden Klassen implementiert
- **`Flushable`** ist insbesondere bei Klassen wichtig, die Daten puffern
- **`void flush() throws IOException`** schreibt gepufferte Daten in den Strom

Datenströme

Die Kommandozeile

- Die Kommandozeile stellt Eingabe- und Ausgabeschnittstellen zur Verfügung
  - *stdin* Standardeingabe
  - *stdout* Standardausgabe
  - *stderr* Fehlerausgabekanal
- Beim Laden der Klasse **`System`** werden automatisch Objekte erzeugt, die die Schnittstellen ansprechen können
  - **`System.in`** für die Eingabe
  - **`System.out`** für die Ausgabe
  - **`System.err`** für die Fehlerausgabe
- Mit diesen Strömen können Daten eingelesen oder ausgegeben werden
- Beispiel: **`iotest1`**

Datenströme

Dateien mit Texten

- Mit dem **`FileWriter`** Texte in Dateien schreiben
  - Der **`FileWriter`** erlaubt zeichenbasierte Ausgaben in eine Datei
  - Methoden aus Oberklassen werden nicht überschrieben bzw. implementiert
  - Die Klasse fügt nur Konstruktoren hinzu, damit eine Datei geöffnet werden kann
  - Alle Methoden sind von der Klasse **`OutputStreamWriter`** geerbt
- Zeichen mit der Klasse **`FileReader`** lesen
  - Der FileReader liest aus Dateien einzelne Zeichen, Strings oder Zeichenfelder
  - Wie beim **`Writer`** deklariert die Klasse Konstruktoren zur Annahme des Dateinamens
  - Die Methoden zum Lesen stammen aus den Oberklassen **`InputStreamReader`** und **`Reader`**
- Beispiel: **`iotest2`**

Datenströme

Dateien kopieren

- Will man eine Datei kopieren sollte man keinen Reader verwenden, sondern muss byteweise arbeiten
- Byteströme für Dateien stellen die Klassen **`FileOutputStream`** und **`FileInputStream`** zur Verfügung
- Das Kopieren einer Datei Byte für Byte dauert relativ lang, da jedes Byte einzeln ausgelesen und geschrieben werden muss
- Dateien sind zudem in Blöcken angeordnet, so dass ein blockweises lesen und schreiben auch auf Betriebssystemebene schneller ist
- Beispiel: **`iotest3`**

SequenceInputStream

Daten nacheinander aus mehreren Strömen lesen

- Ein **SequenceInputStream**-Filter hängt mehrere Eingabeströme zu einem großen Eingabestrom zusammen
- Nützlich ist dies, wenn wir aus Strömen lesen wollen und es uns egal ist, was für ein Strom es ist, wo er startet und wo er aufhört.
- Der **SequenceInputStream** lässt sich erzeugen, indem im Konstruktor zwei **InputStream**-Objekte mitgegeben werden
- Soll aus zwei Dateien ein zusammengesetzter Datenstrom gebildet werden, benutzen wir folgende Programmzeilen:

```
InputStream s1 = new FileInputStream(„datei1.txt");
InputStream s2 = new FileInputStream(„datei2.txt");
InputStream s = new SequenceInputStream(s1, s2);
```

- Ein Aufruf irgendeiner **read()**-Methode liest nun erst Daten aus **s1**
- Liefert **s1** keine Daten mehr, kommen die Daten aus **s2**
- Kommen keine Daten mehr von **s2**, aber wieder von **s1** können diese nicht mehr verarbeitet werden

Filter

- **`FilterInputStream`** und **`FilterOutputStream`** filtern Bytes der Ein- und Ausgabe
- Ein Filter besitzt Methoden zum Lesen, bzw. Schreiben von Daten
- Die Lese- und Schreibaufrufe werden nur an gekapselte Streams weitergereicht
- Der Filter selbst ist für eine Veränderung der ausgelesenen Daten zuständig
- **`FilterInputStream`** und **`FilterOutputStream`** haben selbst keinen Filter implementiert, dafür sind deren Subklassen zuständig
  - **`BufferedInputStream`** und **`BufferedOutputStream`** sind Filter, die blockweise lesen/schreiben und steigern dadurch die IO-Geschwindigkeit
  - **`GZIPInputStream`** liest komprimierte Daten aus, **`GZIPOutputStream`** schreibt Daten im ZIP Format
- Filter sind ideal geeignet, um eigene Datenmanipulationen zu implementieren

Rückblick

- File
- Datenströme
  - Basisklassen
  - **`System.in, System.out, System.err`**
  - Textdateien schreiben/auslesen
  - Dateien kopieren
  - **`SequenzInputStream`**
  - Filter

Ausblick

- Filter: WordCount
- Die Klasse **`String`**
- **…**