Informatik B - Objektorientierte Programmierung in Java

Vorlesung 14:

Sequentielle Ein-/Ausgabe, Streams, Dateiorganisation

Inhalt

- Sequentielle Ein-/Ausgabe, Streams, Dateiorganisation
  - Einführung
  - Datenhaltung und Persistenz
  - Standarddaten-Streams
  - Eingabe-Streams
  - Ausgabe-Streams
  - Dateiorganisation
  - Direktzugriffsspeicher (RandomAccessFile)
  - Beispiele

Einführung

- Für die Ein- und Ausgabeprogrammierung stehen im Paket `java.io` unterschiedliche Klassen für Ein- bzw. Ausgabe-Streams sowie eine Reihe von Hilfsklassen zur Verfügung.
- Der Grundgedanke der Java-Datenströme ist die einheitliche Modellierung der Datenübermittlung zwischen Datenquelle und Datenspeicher (Datensenke), unabhängig davon, von welcher Art die Quelle bzw. der Speicherplatz sind (Lesen von Datei, Schreiben in einen Speicherbereich im Hauptspeicher, Lesen von Daten aus einer Quelle aus dem Internet etc.).

Einführung

- Der Entwickler muss jeweils ein Objekt von der von ihm benötigten Stream-Klasse erzeugen und kann für die eigentliche Ein- oder Ausgabe die vordefinierten Operationen der Streams anwenden.
- Das Schachteln von Streams erlaubt die Konstruktion von Filtern, die bei der Ein- oder Ausgabe bestimmte Zusatzfunktionen ausführen.

Einführung

- Die Streams im Paket `java.io` lassen sich nach den folgenden Hauptmerkmalen einteilen:
  - Ein- und Ausgabe
    - Die wichtigste Unterscheidung betrifft den Unterschied zwischen Eingabe- und Ausgabe-Streams, d.h. zwischen Streams, die zum Einlesen von Daten aus einer Datenquelle verwendet werden und solchen, die der Ausgabe in einen Datenspeicher dienen.

Einführung

  - Binär- bzw. Zeichen-Streams
    - Je nachdem, ob es sich um Datenströme vom Typ `byte` oder um Streams auf der Basis von Zeichen (`char`) handelt, unterscheidet man Byte-Streams (Basisklassen `InputStream` und `OutputStream`) und Zeichen-Streams (Basisklassen `Reader` bzw. `Writer`).
    - Die Unterscheidung zwischen Byte- und Zeichen-Streams ist deshalb bedeutsam und notwendig, da Java auf UNICODE basiert und die Datentypen `char` und `byte` einen unterschiedlichen Darstellungsbereich haben (`char` ist ein 16 Bit-Datentyp und daher zwei Byte breit).
    - Für die Umwandlung zwischen Byte- und Zeichen-Streams stehen sog. Brückenklassen zur Verfügung (Klassen `InputStreamReader` und `OutputStreamWriter`).

Datenhaltung und Streams

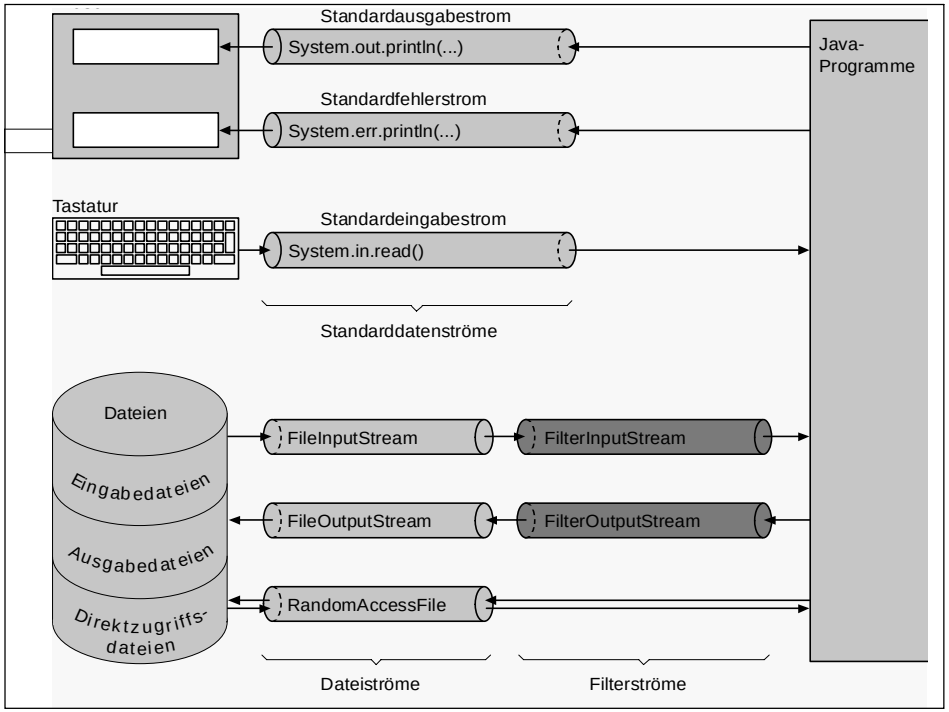
- Datenhaltung in Java
  - Ein-/Ausgabe mit Hilfe von Streams im Java-Paket `java.io`
    - Stream = Schnittstelle eines Programms nach außen
    - Vergleichbar mit einer Pipeline
      - Auf der einen Seite Füllen der Pipeline mit Daten
      - Auf der anderen Seite Entnehmen der Daten
      - Verhalten ist analog zu einer Warteschlange
    - Streams sind immer unidirektional
      - Ein Eingabestream kann *nicht* zur Ausgabe benutzt werden und umgekehrt
    - Klasse `RandomAccessFile`
      - Ermöglicht das Lesen *und* Schreiben einer Datei

Anwendung: Persistenz und Datenhaltung

- Externe Speicher
  - Aufbewahren der Zustände der Objekte über den aktuellen Programmlauf
  - Persistenz
    - Aus langfristig gespeicherten Daten kann wieder ein analoger Arbeitsspeicherzustand wie vor der Speicherung hergestellt werden
  - Drei verschiedene Möglichkeiten, Daten langfristig aufzubewahren
    - Dateien
    - objektorientierte Datenbanken
    - relationale Datenbanken.

Standarddatenströme

- Standarddatenströme
  - Standardeingabe-Stream (`System.in`)
    - liest Bytes von der Standardeingabe ein
    - Beispiel: `System.in.read();`
  - Standardausgabe-Stream (`System.out`)
    - gibt Zeichen auf den Bildschirm aus
    - Beispiel: `System.out.println(...);`
    - ACHTUNG: System.out ist ein Byte-Stream!
  - Standardfehler-Stream (`System.err`)
    - gibt Fehlermeldungen aus
    - Beispiel: `System.err.println(...);`
  - `java.lang.System` ermöglicht den Zugriff auf die Systemfunktionalität.



Eingabe-Streams

- Eingabe-Streams lesen Daten aus einer Quelle ein, wobei die Quelle ein beliebiger Datenlieferant sein kann, etwa eine Datei im Dateisystem, eine Netzwerkverbindung, die über einen Port und ggf. ein bestimmtes Protokoll angesprochen wird (z. B. HTTP, FTP), ein String (also ein Zeichenkettenobjekt im Speicher) oder ein anderer Eingabe-Stream, der seine Daten an diesen Stream weiterreicht.
- Die Verkettung von Streams ermöglicht es, mehrere Streams, z.B. Dateien, zusammenzufassen und für den Aufrufer als einen einzigen Stream darzustellen.

Eingabe-Streams

- Die beiden Basisklassen für Byte- und Zeicheneingabe-Streams, `InputStream` und `Reader` haben weitgehend identische Operationen für das Einlesen von Daten und die Manipulation der Verbindung zur Datenquelle:
  - `int read()/int read()`
  - `int read(byte[] b)/int read(char[] cbuf)`
  - `int read(byte[] b, int off, int len) / int read(char[] cbuf, int off, int len)`
    - Lesen eines einzelnen Bytes bzw. Zeichens oder eines Felds von Bytes bzw. Zeichen aus der Quelle
    - Ein Rückgabewert von –1 zeigt das Ende des Eingabe-Streams an.

Eingabe-Streams

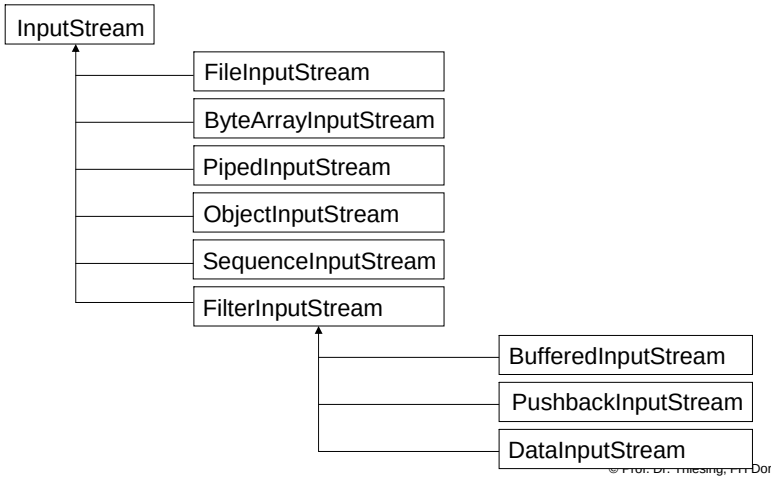
- `long skip(long n)`
  - springt von der aktuellen Position in der Datenquelle um n Bytes oder Zeichen weiter
- `boolean markSupported()`
  - prüft, ob die Datenquelle das Setzen einer Marke unterstützt
- `void mark(int MarkLimit)`
  - setzt eine Marke, an die mit `reset()` zurückgesprungen werden kann, soweit nicht mehr als `MarkLimit` Byte oder Zeichen gelesen wurden.
    Warum sollte man `MarkLimit` angeben? Man kann ja auch aus einem Stream lesen, den man nicht positionieren kann, z.B. `System.in`. Daher wird eine bestimmte Menge an Bytes (`MarkLimit` viele) im Ziwschenspeicher abgelegt.

Eingabe-Streams

- `void reset()`
  - setzt den "Lesekopf' an die Position der Marke in der Datenquelle zurück
- `int available()`
  - liefert die Anzahl an Bytes, die ohne Blockieren mindestens gelesen werden können (nur bei `InputStream`)
- `boolean ready()`
  - liefert `true`, falls der nächste Aufruf von `read()` ohne Blockieren erfolgen kann (nur bei `Reader`)
- `void close()`
  - schließt den Stream

Eingabe-Streams

- Klassenhierarchie Byteeingabe-Streams

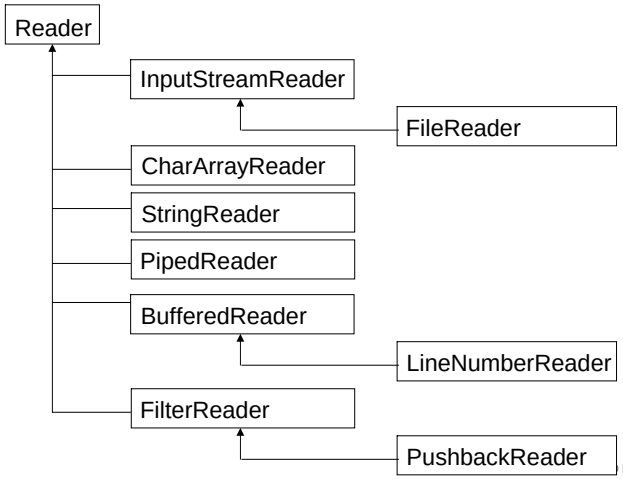


Eingabe-Streams

Klasse	Beschreibung
InputStream	Abstrakte Klasse für die byte-orientierte Eingabe
FileInputStream	Eingabe von Datei
ByteArrayInputStream	Eingabe aus Byte-Array
PipedInputStream	Eingabe von Pipe (Nachrichtenwarteschlange zur Kommunikation zwischen Threads)
ObjectInputStream	Eingabe serialisierter Objekte
SequenceInputStream	Verkettung mehrerer Eingabe-Streams zu einem Stream
FilterInputStream	Abstrakte Klasse für die gefilterte Eingabe
BufferedInputStream	Eingabe mit Pufferung
PushbackInputStream	Eingabe mit der Möglichkeit zur Rückstellung von gelesenen Bytes
DataInputStream	Eingabe von primitiven Datentypen in portabler Weise

Eingabe-Streams

- Klassenhierarchie Zeicheneingabe-Streams



Eingabe-Streams

Klasse	Beschreibung
Reader	Abstrakte Klasse für die zeichen-orientierte Eingabe
InputStreamReader	Klasse für das Interpretieren eines Byte-Streams als Zeichen-Stream
FileReader	Eingabe von Datei
CharArrayReader	Eingabe aus Zeichen-Array
StringReader	Eingabe aus Zeichenkette
PipedReader	Eingabe von Pipe (Nachrichtenwarteschlage zur Kommunikation zwischen Threads)
BufferedReader	Eingabe mit Pufferung und Eingabe ganzer Zeilen
LineNumberReader	Eingabe mit der Fähigkeit, Zeilen zu zählen
FilterReader	Abstrakte Klasse für gefilterte Eingabe
PushbackReader	Eingabe mit der Möglichkeit zur Zurückstellung von Zeichen

Eingabe-Streams

- Zu den Operationen von `InputStream` bzw. `Reader` kommen für einzelne Stream-Klasse spezifische Operationen hinzu, die die besondere Funktionalität dieses Stream-Typs ausmachen.
- Beispielsweise verfügt `BufferedReader` als Unterklasse von `Reader` zusätzlich über eine Operation `readLine()`, die eine ganze Zeile aus einer Datenquelle (z.B. eine Textdatei, die über einen `FileReader` angesprochen wird) einliest.

Eingabe-Streams

- Schachteln von Eingabe-Streams
  - Das Schachteln von Eingabe-Streams bietet die Möglichkeit, bei der Eingabe bestimmte Zusatzfunktionen und Filter zu realisieren.
  - Zum Schachteln von Eingabe-Streams wird dem Konstruktor z.B. der Klasse `BufferedReader` ein Objekt der Klasse `FileReader` übergeben:
    - ```
      BufferedReader r = new BufferedReader(
                  new FileReader("MeinText.txt"));
      ```
  - Eine Leseoperation des `BufferedReader` ruft dann eventuell zunächst die Leseoperation des `FileReader` auf, um den Puffer aufzufüllen und liefert anschließend die Daten als Ergebnis zurück.
      ```

Eingabe-Streams

- Verketten von Eingabe-Streams
  - Mit Hilfe der Klasse `SequenceInputStream` können mehrere Eingabe-Streams zusammengefasst werden, so dass sie wie ein einzelner Stream erscheinen.
  - Die Daten werden nacheinander aus den einzelnen Streams gelesen.
  - Die Streams können z.B. in Form einer `Enumeration` oder - wenn es sich um genau zwei Streams handelt - direkt an den Konstruktor übergeben werden:
    - `public SequenceInputStream(Enumeration e)`
    - `public SequenceInputStream(InputStream s1, InputStream s2)`

Ausgabe-Streams

- Ausgabe-Streams dienen dazu, die an sie übergebenen Daten in einen Datenspeicher zu schreiben.
- Analog zu den Eingabe-Streams können die Datenspeicher unterschiedlicher Art sein (Datei, Zeichenkette im Speicher, Netzwerkverbindung, ein anderer Ausgabe-Stream etc.).

Ausgabe-Streams

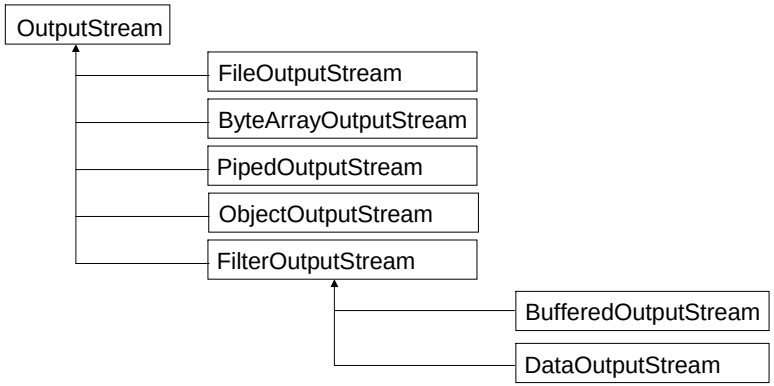
- Die beiden abstrakten Basisklassen für die Datenausgabe, `OutputStream` und `Writer`, haben annähernd dieselben Operationen:
  - `void write(int b)/void write (int c)`
  - `void write(byte[] b)/void write(char[] cbuf)`
  - `void write(byte[] b, int off, int len)/ void write(char[] cbuf, inf off, int len)`
    - Schreiben eines einzelnen Bytes bzw. Zeichens oder eines Feldes von Bytes/Zeichen in den Ausgabe-Stream

Ausgabe-Streams

  - `void flush()`
    - schreibt noch im Puffer des Streams verbliebene Daten in den Datenspeicher (nur bei gepufferten Streams relevant)
  - `void close()`
    - schließt den Ausgabe-Stream
- Die Klasse `Writer` besitzt zusätzlich folgende Schreiboperationen:
  - `void write(String str)`
  - `void write(String str, int off, int len)`

Ausgabe-Streams

- Klassenhierarchie Byteausgabe-Streams

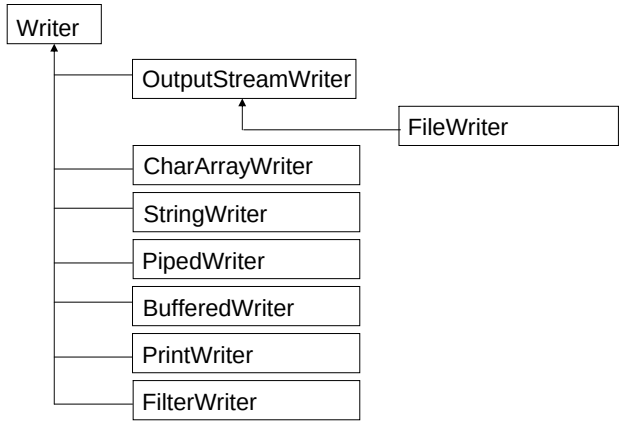


Ausgabe-Streams

Klasse	Beschreibung
OutputStream	Abstrakte Klasse für die byte-orientierte Ausgabe
FileOutputStream	Ausgabe in eine Datei
ByteArrayOutputStream	Ausgabe in ein Byte-Array
PipedOutputStream	Ausgabe in eine Pipe (Nachrichtenwarteschlange zur Kommunikation zwischen Threads)
ObjectOutputStream	Ausgabe serialisierter Objekte
FilterOutputStream	Abstrakte Klasse für die gefilterte Ausgabe
BufferedOutputStream	Ausgabe mit Pufferung
DataOutputStream	Ausgabe von primitiven Datentypen in portabler Weise

Ausgabe-Streams

- Klassenhierarchie Zeichenausgabe-Streams



Ausgabe-Streams

Klasse	Beschreibung
Writer	Abstrakte Klasse für die zeichen-orientierte Ausgabe
OutputStreamWriter	Klasse für die Umformung eines Zeichen-Streams in einen Byte-Stream
FileWriter	Ausgabe in eine Datei
CharArrayWriter	Ausgabe in ein Zeichen-Array
StringWriter	Ausgabe in eine Zeichenkette
PipedWriter	Ausgabe in eine Pipe (Nachrichtenwarteschlage zur Kommunikation zwischen Threads)
BufferedWriter	Ausgabe mit Pufferung
PrintWriter	Ausgabe aller Datentypen im Textformat
FilterWriter	Abstrakte Klasse für die gefilterte Ausgabe

Ausgabe-Streams

- Beim Öffnen einer existierenden Datei zum Schreiben besteht die Möglichkeit, zwischen dem Überschreiben und dem Anhängen an die vorhandenen Daten zu wählen.
- Wie für die Eingabe-Streams gilt, dass die konkreten Unterklassen von `OutputStream` bzw. `Writer` die Operationen der Basisklassen überschreiben bzw. spezifische Operationen hinzufügen.
  - Beispielsweise verfügt `BufferedWriter` über eine Operation `newLine()`, mit der ein Zeilenumbruch in den Ausgabe-Stream geschrieben werden kann.

Beispiele

- DateiEingabe.java
- DateiAusgabe.java

Dateiorganisation

- *random access*
  - Fast alle Programmiersprachen unterstützen eine Direktzugriffspeicherungsform
  - Ermöglicht ohne Durchsuchung der Datei von vorne nach hinten
    - Speicherung von Datensätzen
    - Lesen und erneutes Speichern eines Datensatz
  - Voraussetzung für diese Speicherungsform:
    - Alle Datensätze haben dieselbe Länge
  - Zugriff:
    - Positionierung eines Zeigers auf den Anfang des gewünschten Datensatzes.

Dateiorganisation

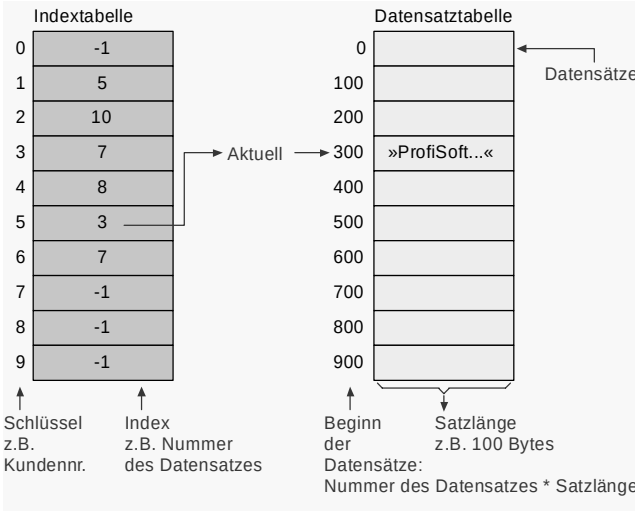
- Endbenutzer weiß nicht, an welcher Position ein Datensatz beginnt
  - Herstellung und Verwaltung einer Zuordnung zwischen einem fachkonzeptorientierten Schlüssel und der Position des Datensatzes
- Im kaufmännischen Bereich:
  - Verwendung von Nummern für die Identifikation
    - Z.B. Kundennummer, Artikelnummer usw.

Dateiorganisation

- Indexverwaltung
  - Verwaltung der Zuordnung zwischen einem solchen Schlüssel und einer zugehörigen Datensatzposition in einer Tabelle
  - Indextabelle beansprucht nur wenig Platz
    - Sie kann komplett in den Arbeitsspeicher geladen werden
    - Sie wird aber selbst ebenfalls in einer Datei gespeichert und bei Änderungen aktualisiert.

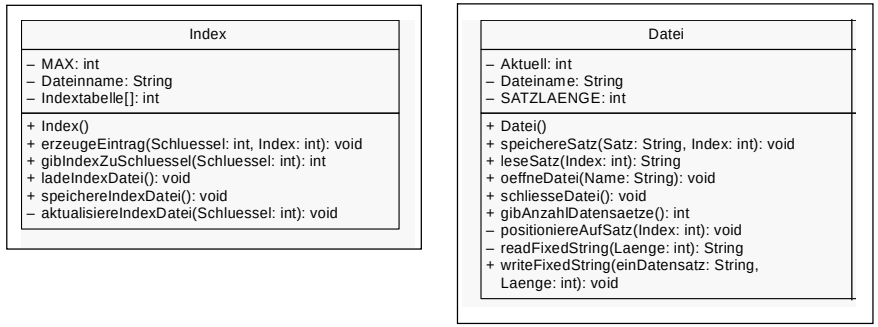
Dateiorganisation

- Indizierte Organisation mit physisch sortiertem Index



Dateiorganisation

- Klassen für eine Indexverwaltung: Beispiel



Dateiorganisation

- Klassen sind unabhängig voneinander
- Indexverwaltung kann ausgetauscht oder erweitert werden, ohne Änderung der Klasse `Datei`
- Beispiel
  - Verwendung des Kundennamens zusätzlich als Schlüssel
    - Eine weitere Indexklasse kann mit Hilfe einer *Hash*-Tabelle eine Zuordnung zwischen Kundenname und Datensatzposition verwalten.

Direktzugriffsspeicher in Java

- `RandomAccessFile`
  - `RandomAccessFile(String name,String mode)`
    - 1. Parameter: systemabhängiger Dateiname
    - 2. Parameter: `“r”` oder `“rw”`
  - `public void writeInt(int)`
    - Schreibt ein `int` in die Datei (jeweils 4 Bytes)
  - `public int readInt()`
    - Liest eine 32-Bit-lange ganze Zahl von der Datei
  - `public void writeChar(int)`
    - Schreibt 1 Zeichen im Unicode (2 Bytes) in die Datei
  - `public char readChar()`
    - Liest ein Unicode-Zeichen von der Datei.

Direktzugriffsspeicher in Java

  - `public void seek(long pos)`
    - Positionszeiger wird auf den zu lesenden Satz gestellt
    - Ab dieser Position wird dann mit den Lese- und Schreiboperationen gelesen bzw. Geschrieben
    - `pos` gibt dabei die *Byte*-Position an, d.h. die Länge der Datensätze wird in *bytes* gezählt
  - `public int skipBytes(int n)`
    - Überspringen von n Bytes
  - `public long length()`
    - Gibt die Länge der Datei zurück
- Beispiel: Klasse `Direktzugriffsspeicher`
  - Klasse `Index` / Klasse `Datei`.