Informatik B

Vorlesung 6

WordCount, SystemProperties, Systemvariablen, Strings

Rückblick

- File
- Datenströme
  - **`Reader/Writer`**
  - **`InputStream, OutputStream`**
  - Brückenklassen
  - **`Filter`**

WordCount

- **`wc`** ist ein Unix-Kommando
- Das Kommando zählt bei einer Datei:
  - Zeilen
  - Wörter
  - Zeichen

WordCount in Java

- Ein Stream muss geöffnet werden
  - Es sollen Zeichen gelesen werden
  - Die Datei muss mit einem **`Reader`** ausgelesen werden
- Beim Auslesen der Daten soll gezählt werden
  - Der Datenstrom soll also weiterverarbeitet werden
  - Es sollte ein **`FilterReader`** verwendet werden
  - Die Methoden **`int read()`** und **`int read(char[] buf,int offset, int length)`** müssen implementiert werden

Zeilenumbruch

- Um Zeilen zählen zu können muss klar sein, wann eine Zeile zu Ende ist
- Ein Zeilenumbruch wird durch besondere Zeichen in einer Datei dargestellt
  - Unter Mac `\r`
  - Unter Windows `\r\n`
  - Unter Linux `\n`
- Problem: Das Programm soll plattformunabhängig sein

Systemvariablen

- Die Java-Umgebung verwaltet Systemeigenschaften
  - Pfadtrenner
  - Version der JVM
  - Betriebsystemname
  - und viele weitere
- Diese Eigenschaften werden in einem **`java.util.Properties`**-Objekt verwaltet
- Die statische Funktion **`System.getProperties()`** erfragt diese Systemeigenschaften
- Zum Erfragen einzelner Eigenschaften ist das Properties-Objekt aber nicht unbedingt nötig: **`System.getProperty(String key)`** erfragt direkt eine Eigenschaft
- Abfrage des Zeilentrenners zur Laufzeit: **`System.getProperty("line.separator");`**

Systemvariablen

- **`static String getProperty(String key)`**
  - Gibt die Belegung einer Systemeigenschaft zurück
  - Ist sie nicht vorhanden, ist der Rückgabewert **`null`**
- **`static String getProperty(String key, String def)`**
  - Gibt die Belegung einer Systemeigenschaft zurück
  - Ist sie nicht vorhanden, liefert die Methode die Zeichenkette **`def`**
- **`static String setProperty(String key, String value)`**
  - Belegt eine Systemeigenschaft neu
  - Rückgabe ist die alte Belegung – oder **`null`**, falls es keine alte Belegung gab
- **`static String clearProperty(String key)`**
  - Löscht eine Systemeigenschaft aus der Liste
  - Rückgabe ist die alte Belegung – oder **`null`**, falls es keine alte Belegung gab.
- **`static Properties getProperties()`**
  - Liefert ein mit den aktuellen Systembelegungen gefülltes **`Properties`**-Objekt
- Beispiel: **`systemproperties1`**

Properties setzen

- Eigenschaften lassen sich auch beim Programmstart setzen
- Praktisch für eine Eigenschaft, die beispielsweise das Verhalten des Programms steuert oder das Programm konfiguriert
- In der Kommandozeile werden mit -D der Name der Eigenschaft und ihr Wert angegeben
  `java –DDEBUG=true MyProperty`
- Die so gesetzte Property kann über die Methode **`getProperty(„DEBUG“);`** ausgelesen werden
- Falls ein Wahrheitswert gesetzt wird, kann dieser auch mit **`Boolean.getBoolean("DEBUG")`** abgefragt werden
- Beispiel: **`systemproperties2`**

Systemvariablen

- Fast jedes Betriebssystem nutzt das Konzept der Umgebungsvariablen
- Die von Java mittels **`System.getProperties()`** eingeführten Eigenschaften unterscheiden sich grundlegend von den System-Umgebungsvariablen und tauchen daher in der Liste nicht auf
- Seit Java5 ist es möglich, auf diese System-Umgebungsvariablen zuzugreifen
- Dazu dient die statische Methode **`System.getenv()`**
- Sie liefert eine Menge von **`<String, String>`**-Paaren
- Spezielle Umgebungsvariablen können mit **`getenv(String)`** abgefragt werden
- Beispiel: **`systemvariablen1`**

WordCount

- Implementierung einer Unterklasse von **`FilterReader`**
- Methode **`int read()`**
  - Liest Zeichen für Zeichen
  - Feststellen welches Zeichen und in Abhängigkeit davon Wort-, Zeilen- und Zeichenanzahl hochzählen
- Methode **`int read(char buffer[], int offset, int length)`**
  - Nicht auf die Methode **`int read()`** zurückführen, da sonst der Performancevorteil des blockweisen Lesens nicht gegeben wäre
  - **`char-`**Buffer Zeichen für Zeichen durchgehen und ggf. die einzelnen Zählvariablen hochzählen
- Beispiel: **`wc`**
- Frage: Warum braucht man im Beispiel **`wc`** nicht die Methode **`int read(char[] buf)`** zu überschreiben?

`String`

- Ein String ist eine Sammlung von Zeichen, die im Speicher geordnet abgelegt sind
- Die Zeichen sind einem Zeichensatz entnommen, der in Java dem 16-Bit-Unicode-Standard entspricht
- Wenn ein einzelnes Zeichen in einer Variablen vom Typ **`char`** gespeichert werden kann, können Zeichenfolgen in einem **`char`**-Feld gehalten werden
- Wollten wir Zeichenfolgen selbstständig in Feldern verwalten, ist das mühselig, denn wir müssten uns über viele Dinge selbst kümmern, etwa um die Verschiebung der Zeichen beim Löschen und Einfügen in der Mitte, um die Verwaltung des Puffers und vieles mehr

`String`

- Java sieht drei Klassen vor, die Zeichenfolgen von der Implementierung abstrahieren
- Die Klassen entsprechen der idealen Umsetzung der objektorientierten Idee:
  - Die Daten sind gekapselt, das heißt, die tatsächliche Zeichenkette ist in der Klasse als privates Feld gegen Zugriffe von außen gesichert
  - Die Länge ist ein privates Attribut der Klasse, die nur über eine Methode zugänglich ist
- Die Klasse **`String`** repräsentiert nicht änderbare (*immutable*) Zeichenketten
- Es gibt einige Methoden, die scheinbar Veränderungen an Strings vornehmen, aber sie erzeugen in Wahrheit neue **`String`**-Objekte
- Beim Aneinanderhängen zweier **`String`**-Objekte entsteht beispielweise ein drittes **`String`**-Objekt für die zusammengefügte Zeichenreihe
- Die Klassen **`StringBuffer`** und **`StringBuilder`** repräsentieren im Gegensatz zu **`String`** dynamisch änderbare Zeichenreihen
- Der Unterschied zwischen **`StringBuffer`** und **`StringBuilder`** ist lediglich, dass **`StringBuffer`** vor der gleichzeitigen Veränderungen verschiedener Prozesse geschützt ist

String

- Damit Zeichenketten genutzt werden können, muss ein Objekt der Klasse **`String`** oder **`StringBuffer/StringBuilder`** erzeugt worden sein
- **`String`**-Literale, müssen nicht immer von Hand mit **`new`** erzeugt werden, ein Ausdruck in doppelten Anführungszeichen ist automatisch ein **`String`**-Objekt
- Das bedeutet auch, dass hinter dem **`String`**-Literal gleich ein Punkt für den Methodenaufruf stehen kann: **`„Hallo Welt“.length();`**
- Die JVM erzeugt für jedes Zeichenketten-Literal automatisch ein entsprechendes **`String`**-Objekt
- Das geschieht für jede konstante Zeichenkette höchstens einmal, egal wie oft sie im Programmverlauf benutzt wird
- Dieses **`String`**-Objekt lebt in einem Bereich, der Konstantenpool genannt wird

Primitive Datentypen von Strings

- Will man aus einer Zeichenkette einen primitiven Datentyp gewinnen, stehen einem eine Vielzahl von Methoden zur Verfügung:

```
java.lang.Boolean.parseBoolean(String s)
java.lang.Byte.parseByte(String s)
java.lang.Short.parseShort(String s)
java.lang.Integer.parseInt(String s)
java.lang.Long.parseLong(String s)
java.lang.Double.parseDouble(String s)
java.lang.Float.parseFloat(String s)
```

- Kann eine Methode eine Konvertierung nicht durchführen, weil sich der **`String`** nicht konvertieren lässt, löst sie eine **`NumberFormatException`** aus
- Um Probleme mit führenden oder nachfolgenden Leerzeichen zu vermeiden, können diese vorab entfernt werden:

```
String s = „   123   „;
s = s.trim();
int i = Integer.parseInt(s);
```

Vergleich von String-Objekten

- **`equals()`**
  - Der Inhalt eines **`String`**-Objektes kann mittels **`equals()`** verglichen werden
  - Sind in den **`String`**-Objekten gleiche Zeichenketten enthalten liefert **`equals()`** den Wert **`true`**
- **`==`**
  - Die physikalische Gleichheit kann mittels des **`==`** Operators verglichen werden
  - Implizit erzeugte **`String`**-Literale sind bei gleicher Zeichenkette physikalisch gleich: **`„Hallo“==„Hallo“ ist true`**
  - Explizit erzeugte **`String`**-Literale sind physikalisch nicht gleich: **`new String(„Hallo“) == new String(„Hallo“)`** ergibt **`false`**
- **`compareTo()`**
  - Um Strings lexikografisch zu sortieren implementiert die Klasse **`String`** das Interface **`Comparable`**
  - Mit der Methode **`aString.compareTo(anotherString)`** kann getestet werden, welcher der beiden Strings kleiner, gleich oder größer ist
- Beispiel: **`stringvergleich1`**

String-Verkettung

- Ein Aneinanderhängen von Zeichenketten wird oft benötigt
- Objekte vom Typ **`String`** können durch den Plus-Operator mit anderen Strings und Datentypen verbunden werden
- Falls zusammenhängende Teile nicht alle den Datentyp **`String`** besitzen, werden sie automatisch in einen **`String`** umgewandelt
- Besteht der Ausdruck aus mehreren Teilen, so muss die Auswertungsreihenfolge beachtet werden
- **`„Ich bin " + 30 + 3 + " Jahre alt"`** ergibt **`„ich bin 303 Jahre alt“`**
- Beispiel: **`string1`**
- **`„30 + 3 + "Jahre„`** ergibt **`„33 Jahre“`**

`String`-Verkettung

- Der Compiler startet die Konvertierung in einen **`String`**, wenn der den Ausdruck als **`String`**-Objekt erkannt hat
- Der Plus-Operator für Zeichenketten geht streng von links nach rechts und bereitet mit eingebetteten arithmetischen Ausdrücken mitunter Probleme
- Abhilfe: Klammerung der Ausdrücke verwenden
- Beispiel: **`string2`**

`StringBuffer` und `StringBuilder`

- Zeichenketten haben die Eigenschaft, dass ihr Inhalt nicht mehr verändert werden kann
- Die Exemplare der Klasse **`StringBuffer`** und **`StringBuilder`** sind veränderbar
- Die Veränderungen betreffen das **`StringBuffer/StringBuilder`**-Objekt selbst, und es wird kein neu erzeugtes Objekt als Ergebnis geliefert, wie beim Plus-Operator
- Sonst sind sich aber die Implementierung von **`String`**-Objekten und **`StringBuffer/StringBuffer`**-Objekten ähnlich
- In beiden Fällen nutzen die Klassen ein internes Zeichenfeld
- Zeichenketten können:
  - Aneinandergehängt werden: verschiedene **`append()`**-Methoden
  - Umgekehrt werden: **`reverse()`**
  - Geändert werden: **`setCharAt()`**
  - Ersetzt werden: **`replace()`**
  - Usw.

Problem mit dem Plus-Operator

- Wird das „+“ zur Verkettung von Strings verwendet, wird implizit ein **`StringBuilder`** Objekt erzeugt
- Die beiden durch das „+“ verbundene Strings werden dann mittels **`append()`** aneinandergehängt
- Dies kann zu einem Geschwindigkeitsnachteil innerhalb von Schleifen führen
- Wird innerhalb einer Schleife ein String mittels eines „+“ zusammengebaut, wird immer wieder ein neues **`StringBuilder`**-Objekt angelegt
- Zudem wird immer wieder ein neues **`String`**-Objekt erzeugt
- Dies kostet sehr viel Rechenzeit
- Besser ist hier die direkte Verwendung eines **`StringBuilder`**-Objektes
- Außerhalb von Schleifen wird innerhalb eines Blockes nur ein **`StringBuilder`**-Objekt erzeugt
- Beispiel: **`string3`**

Formatieren von Ausgaben

- Immer wieder müssen Zahlen, Datumsangaben und Text auf verschiedenste Art und Weise formatiert werden
- Die Klasse **`String`** stellt mit der statischen Methode **`format(String format,Object … args)`** eine Möglichkeit bereit, Zeichenketten nach einer Vorgabe zu formatieren
- **`format`** ist ein Format-String
  - Er enthält neben auszugebenden Zeichen so genannte Format-Spezifizierer
  - Diese geben dem Formatierer darüber Auskunft, wie das Argument formatiert werden soll

Format-Spezifierer

- **%n** Neue Zeile
- **%b** Boolean
- **%%** Prozentzeichen
- **%s** String
- **%c** Unicode-Zeichen
- **%d** Dezimalzahl
- **%x** Hexadezimalschreibweise
- **%t** Datum und Zeit
- **%f** Fließkommazahl
- **%e** Wissenschaftliche Notation

Variable Parameterliste

- **`Object … args`** ist eine variable Parameterliste
- Bei vielen Methoden ist es klar, wie viele Argumente sie haben
- Es gibt jedoch Methoden, wo die Anzahl mehr oder weniger frei ist
- Z.B. bei einer Methode **`max()`**, die grundsätzlich auch unterschiedlich viele Parameter haben könnte
- Seit Java5 können Methoden mit variabler Argumentanzahl, auch *Varargs* genannt, definiert werden
- Eine Funktion mit variabler Argumentanzahl nutzt drei Punkte zur Verdeutlichung, dass eine beliebige Anzahl Argumente angegeben werden dürfen
- Bei einer variablen Parameterliste kann immer nur Parameter eines Typs enthalten:

```
int max( int... array ) {  // Maximumbestimmung
}
```

- Die variable Parameterliste wird als Array interpretiert
- Beispiel: **`variableparameterliste1`**

Formatieren von Ausgaben

- Bei der formatierten Ausgabe mit der Methode **`format`** wird also zunächst ein **`String`** mit Platzhaltern und Formatierungen angegeben
- In den weiteren Parametern stehen die zugehörigen Werte:
  `String s = String.format(„Hallo %s %s wie gehts“,vorname, nachname);`
- Die Ausgabe formatierter Strings ist auch mit der Methode **`printf()`** für **`PrintWriter`**-Objekte möglich:
  `System.out.printf(„Hallo %s %s wie gehts“,vorname, nachname);`
- Generell können die Format-Spezifizierer mit zusätzlichen Angaben zur Formatierung versehen werden
- Einen Überblick gibt die Javadoc Seite der Klasse **`java.util.Formatter`**
- Beispiel: **`string4`**

Internationalisierung der Ausgaben

- Verschiedene Länder haben Eigenarten bei der Darstellung
  - Ein Datum wird verschieden dargestellt
  - Gleitkommazahlen werden unterschiedlich ausgegeben
  - Usw.
- Bei Systemstart wird über die JVM festgestellt, welche Landeseinstellungen im Betriebssystem vorgegeben sind
- Nachträglich können diese Einstellung mit der Klasse **`Locale`** geändert werden:
  `Locale.setDefault(Locale.GERMAN);`
- Dadurch werden die Ausgaben entsprechend formatiert
- Beispiel: **`string5`**

Strings teilen

- Oftmals muss man eine gegebene Zeichenkette zerlegen
- Eine einfache Möglichkeit bietet die Klasse **`StringTokenizer`**
- Erzeugt man einen **`StringTokenizer`** wird diesem eine Zeichenkette übergeben und gegebenenfalls noch ein Zeichen an dem der String aufgeteilt werden soll (*Delimiter*)
- Die einzelnen Teile erhält man dann mit der Methode **`nextToken()`**
- Die Methode **`hasMoreTokens()`** überprüft, ob noch weitere Teile vorhanden sind
- Mit **`countTokens()`** kann die Anzahl der noch verbleibenden Teile ermittelt werden
- Beispiel: **`string6`**

Strings teilen

- Ein **`StringTokenizer`** kann lediglich an einem einzelnen Zeichen splitten
- Will man einen String anhand einer bestimmten Zeichenfolge teilen reicht der **`StringTokenizer`** nicht aus
- Mit der Methode **`split()`** können Reguläre Ausdrücke definiert werden, anhand derer eine Zeichenkette zerlegt werden kann
- Beispiel: **`string7`**

Rückblick

- WordCount
  - Filter
  - SystemProperties
  - Systemvariablen
- Strings
  - Erzeugen (implizit/explizit)
  - Stringverkettung
  - Internationalisierung
  - Strings splitten

Ausblick

- Nebenläufige Programmierung: **`Threads`**