BS Info München

Objektorientierte Programmierung

String, RegEx, Exceptions, IO

Bildquelle: http://sapblog.rmtiwari.com/2009/09/abap-regex-meets-csv-and-said-hello.html

Stand: **03.05.2019**
Autoren: **B. Daum, M. Niedermair**

1. Die Klasse String und ihre Methoden

Zur Speicherung und Bearbeitung von Zeichenketten sieht Java die Klasse *String* vor. Theoretisch könnte man stattdessen auch mit *char*-Arrays arbeiten. In der Praxis ist dies aber zu umständlich.

Anders als der elementare Datentyp *char* ist *String* eine Java-Klasse, die verschiedenste Methoden zur Verfügung stellt (siehe unten). Sie ist im Paket *java.lang* der Klassenbibliothek definiert und steht allen Java-Programmen ohne *import* zur Verfügung. Im Vergleich zu anderen Klassen weisen *Strings* einige Besonderheiten auf:

- Neue Objekte der Klasse *String* werden einfach durch eine Zuweisung (als literal) erzeugt, also ohne *new*.
- Mehrere Zeichenketten können mit den Operatoren + bzw. += verbunden werden (allerdings: schlechte Performance; die Klasse StringBuilder bietet bessere Performance).
- Zeichenketten sind Objekte. Nicht initialisierte Zeichenketten enthalten den Zustand *null*, keine leere Zeichenkette ""! Zeichenketten lösen daher eine *NullPointerException* aus, wenn die zu verarbeitenden *String*-Variablen nicht initialisiert wurden (wie alle anderen Objekte auch).
- Zeichenketten sind unveränderlich (immutable). Es ist daher nicht möglich, eine Zeichenkette zu verändern. Wenn man s = s + "xx" ausführen möchte, wird stattdessen ein neues Zeichenkettenobjekt erzeugt.[1]

Ausgewählte Methoden der Klasse String (alphabetisch sortiert)	**Beispiel-Code**	**Erläuterung**
charAt(index)	charAt(4)	Liefert den character am gewünschten Index zurück, im Beispiel an der 5. Stelle. Achtung, der Index startet immer an der Stelle 0!
compareTo()	str1.compareTo(str2)	Dient dem Vergleich von String1 (str1) mit String2 (str2). Die Methode liefert folgende Rückgabewerte: - *Negativer Wert*, wenn str1 größer als str2 ist, str1 also vor str2 einsortiert werden würde ("a" wird vor "b" einsortiert). - *Positiver Wert*, wenn str1 kleiner als str2 ist, str2 also vor str1 einsortiert werden würde. - *0*, wenn str1 und str2 identisch sind.
contains()	str3.contains("hello")	Überprüft, ob eine bestimmte Zeichenkette innerhalb eines Strings vorhanden ist. Die Rückgabewerte sind dabei entweder *true* oder *false*. Im Beispiel wird überprüft, ob die Variable str3 die Zeichenkette "hello" enthält.

[1] Vgl. zu diesem einführenden Abschnitt **Java der Grundkurs** von Michael Kofler, S. 131/132

equals()	str4.equals(str5)	Dient dem Vergleich des Inhalts von Strings. Sind die Inhalte von str4 und str5 identisch, ist der Rückgabewert *true*, ansonsten *false*. ACHTUNG: Bei einfachen Datentypen dient zum Vergleich das doppelte Istgleich (==). Dies führt im Falle von Strings zwar nicht zu einem Programmabbruch, liefert aber auch kein richtiges Ergebnis, da nur überprüft wird, ob die beiden Objekte str4 und str5 auf die gleiche Referenz verweisen.
equalsIgnoreCase()	str5. equalsIgnoreCase(str6)	Wie *equals()*, allerdings werden Groß- und Kleinschreibung ignoriert.
indexOf()	str7.indexOf("a")	Liefert den Index des ersten "a" zurück, das in der Zeichenkette von str7 enthalten ist. Achtung, auch hier beginnt der Index bei 0!
length()	str8.length()	Gibt die Gesamtlänge der in str8 gespeicherten Zeichenkette zurück.
matches()		Siehe Kapitel "Reguläre Ausdrücke"
replace()	str10.replace(char1, char2) str10.replace('b', 'B') str10.replace('b', 'D')	Durchsucht die Zeichenkette, die in str10 gespeichert ist, nach dem char1. Alle Treffer werden durch char2 ersetzt.
replaceAll()	str11.replaceAll(String regex, String Ersetzung)	Durchsucht Zeichenketten mit Hilfe regulärer Ausdrücke und nimmt Ersetzungen vor (siehe Kapitel "Reguläre Ausdrücke")
split()		Siehe Kapitel "Reguläre Ausdrücke"
startsWith(),endsWith()	str13.startsWith("hello") str13.endsWith("world")	Überprüft, ob die gespeicherte Zeichenkette mit "hello" beginnt bzw. mit "world" endet. Rückgabewerte sind *true* und *false*.
substring(start, ende)	str14.substring(4) str14.substring(5,10)	Gibt einen Teil der Zeichenkette zurück, die in str14 gespeichert ist. Im ersten Beispiel wird die komplette Zeichenkette ab der 5. Stelle (Index 4) zurückgegeben. Im zweiten Fall wird die Zeichenkette von der 6. Stelle (Index 5) bis zur 10. Stelle (Index 9) zurückgegeben. Die Indexangabe nach dem Komma (hier Index 10) wird bei der Rückgabe also nicht mehr berücksichtigt! Bsp.: String str = "HelloWorlds"; System.out.println(str.substring(5,10)); // Ausgabe: World
toLowerCase()	str15.toLowerCase()	Der komplette Inhalt von str15 wird in Kleinbuchstaben zurückgegeben.
toUpperCase()	str16.toUpperCase()	Der komplette Inhalt von str16 wird in Großbuchstaben zurückgegeben.
trim()	str17.trim()	Whitespace (Leerzeichen) zu Beginn und am Ende der Zeichenkette, die in str17 gespeichert ist, wird entfernt.

Formatieren und Ausgeben mit format()[2]

Die Klasse String stellt mit der statischen Methode format() eine Möglichkeit bereit, Zeichenketten nach einer Vorgabe zu formatieren:

```
String s = String.format( "Hallo %s. Es gab einen Anruf von %s.", "Chris", "Joy" );
System.out.println( s );    // Hallo Chris. Es gab einen Anruf von Joy.
```

Der erste übergebene String nennt sich Format-String. Er enthält neben auszugebenden Zeichen weitere so genannte Format-Spezifizierer, die dem Formatierer darüber Auskunft geben, wie er das Argument formatieren soll. %s steht für eine unformatierte Ausgabe eines Strings. Nach dem Format-String folgt ein Array mit variabler Länge (oder alternativ das Feld direkt) mit den Werten, auf die sich die Format-Spezifizierer beziehen.

Die wichtigsten Format-Spezifizierer im Überblick:

Spezifizierer	Steht für ...	Spezifizierer	Steht für ...
%n	neue Zeile	%b	Boolean
%%	Prozentzeichen	%s	String
%c	Unicode-Zeichen	%d	Dezimalzahl
%x	Hexadezimalschreibweise	%t	Datum und Zeit
%f	Fließkommazahl	%e	wissenschaftliche Notation

Die **Formatierung von Zahlen** funktioniert nach folgendem Schema[3]:

```
PrintStream o = System.out;

int i = 123;
o.printf( "|%d|%d|%n" ,        i, -i );      // |123|-123|
o.printf( "|%5d|%5d|%n" ,      i, -i );      // |  123| -123|
o.printf( "|%-5d|%-5d|%n" ,    i, -i );      // |123  |-123 |
o.printf( "|%+-5d|%+-5d|%n" , i, -i );      // |+123 |-123 |
o.printf( "|%05d|%05d|%n%n",  i, -i );      // |00123|-0123|

double d = 12345.678;
o.printf( "|%f|%f|%n" ,          d, -d );    // |12345,678000| |-12345,678000|
o.printf( "|%+f|%+f|%n" ,        d, -d );    // |+12345,678000| |-12345,678000|
o.printf( "|% f|% f|%n" ,        d, -d );    // | 12345,678000| |-12345,678000|
o.printf( "|%.2f|%.2f|%n" ,      d, -d );    // |12345,68| |-12345,68|
o.printf( "|%,.2f|%,.2f|%n" ,    d, -d );    // |12.345,68| |-12.345,68|
o.printf( "|%.2f|%(.2f|%n",      d, -d );    // |12345,68| |(12345,68)|
o.printf( "|%10.2f|%10.2f|%n" , d, -d );    // |  12345,68| | -12345,68|
o.printf( "|%010.2f|%010.2f|%n",d, -d );    // |0012345,68| |-012345,68|
```

Die **Formatierung von Datumsangaben** funktioniert nach folgendem Schema:

```
Date t = new Date();
o.printf( "%tT%n", t );                     // 11:01:39
o.printf( "%tD%n", t );                     // 04/18/08
o.printf( "%1$te. %1$tb%n", t );            // 18. Apr
```

Für weitere Format-Spezifizierer für Datumsangaben vgl. unten genannte Quellen.

[2] Vgl. zu diesem Abschnitt **Java ist auch eine Insel** von Christian Ullenboom, http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_04_010.htm, Abrufdatum 01.09.17

[3] Vgl. **Java ist auch eine Insel** von Christian Ullenboom, http://openbook.rheinwerk-verlag.de/javainsel/javainsel_04_011.html, Abrufdatum: 30.09.17

Konvertieren unterschiedlicher Typen in String-Repräsentationen mittels valueOf()[4]

Die statischen überladenen String.valueOf()-Methoden liefern die String-Repräsentation eines primitiven Werts oder eines Objekts. Bei Objekten wird dazu die toString()-Methode aufgerufen.

Beispiel:
Konvertierungen einiger Datentypen in Strings:

```
String s1 = String.valueOf( 10 );          // 10
String s2 = String.valueOf( Math.PI );     // 3.141592653589793
String s3 = String.valueOf( 1 < 2 );       // true
```

Die valueOf()-Methode ist überladen, und insgesamt gibt es für jeden primitiven Datentyp eine Implementierung:

- static String valueOf(boolean b)
- static String valueOf(char c)
- static String valueOf(double d)
- static String valueOf(float f)
- static String valueOf(int i)
- static String valueOf(long l)

  Liefert die String-Repräsentation der primitiven Elemente.
- static String valueOf(char[] data)
- static String valueOf(char[] data, int offset, int count)

  Liefert vom char-Feld oder einem Ausschnitt des char-Feldes ein String-Objekt.

[4] Vgl. zu diesem Abschnitt **Java ist auch eine Insel** von Christian Ullenboom, http://openbook.rheinwerk-verlag.de/javainsel/javainsel_04_005.html, Abrufdatum: 02.09.17

Programmieraufträge[5]

Auftrag 1: Speak like a Bavarian

Erzeugen Sie in einer Testklasse eine Klassenmethode void speakLikeABavarian(String s), die einen übergebenen String auf dem Bildschirm bringt, aber jedes vorkommende „Hallo“ durch „Servus“ ersetzt.

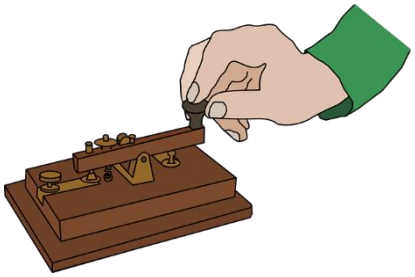
Hinweis: Nutzen Sie bei dieser Aufgabe die fett gedruckten Methoden der Seiten 3 und 4.

Auftrag 2: Ausgabe im Morse-Code

Ein Morse-Funkspruch besteht aus kurzen und langen Symbolen, die mit . und - angedeutet sind. Jeder Buchstabe ist durch ein Leerzeichen getrennt.

Setzen Sie die Zeichenkette „Eat sleep code repeat“ in das Morse-Alphabet um.

```
// A .-      N -.      0 -----
// B -...    O ---     1 .----
// C -.-.    P .--.    2 ..---
// D -..     Q --.-    3 ...--
// E .       R .-.     4 ....-
// F ..-.    S ...     5 .....
// G --.     T -       6 -....
// H ....    U ..-     7 --...
// I ..      V ...-    8 ---..
// J .---    W .--     9 ----.
// K -.-     X -..-
// L .-..    Y -.--
// M --      Z --..
```



Bildquelle: https://pixabay.com/de/vectors/code-kommunikation-hand-geschichte-1295854/

Nicht bekannte Zeichen sollen übersprungen werden.

Auftrag 3: Formatieren

Gegeben seien

```
double d = 1234567.1234;
Calendar c = Calendar.getInstance();
```

Erzeugen Sie die folgenden Ausgaben

```
1234567,12
1.234.567,12
    1.234.567,12  (rechtsbündig auf der Breite von 20 Stellen)
```

und geben Sie zusätzlich das aktuelle Datum in der Form Tag/Monat/Jahr (z.B. 30/04/19) aus.

Auftrag 4: Eigene Übungen

Suchen Sie sich zwei noch nicht genutzte Methoden der Klasse String aus und erstellen Sie ein eigenes Übungsbeispiel, welches diese beiden Methoden sinnvoll einsetzt.
Versehen Sie Ihr Programm mit Kommentaren, um es anschließend im Plenum präsentieren zu können.

[5] Vgl. http://www.tutego.de/javabuch/aufgaben/string.html, Abrufdatum: 04.05.18

2. Reguläre Ausdrücke[6]

Ein regulärer Ausdruck (engl. regular expression, kurz regex) ist die Beschreibung eines Musters (engl. *pattern*). Reguläre Ausdrücke werden bei der Zeichenkettenverarbeitung beim Suchen und Ersetzen eingesetzt. Für folgende Szenarien bietet die Java-Bibliothek entsprechende Unterstützung an:

- Frage nach einer **kompletten Übereinstimmung**: Passt eine Zeichenfolge komplett auf ein Muster? Man nennt das „match“. Die Rückgabe einer solchen Anfrage ist entweder *true* oder *false*.
- **Finde Teilstrings**: Das Pattern beschreibt einen Teilstring, und gesucht sind alle Vorkommen dieses Musters in einem Suchstring.
- **Ersetze Teilfolgen**: Das Pattern beschreibt Wörter, die durch andere Wörter ersetzt werden.
- **Zerlegen einer Zeichenfolge**: Das Muster steht für Trennzeichnen, so dass nach dem Zerlegen eine Sammlung von Zeichenfolgen entsteht.

Ein Pattern-Matcher ist die „Engine“, die reguläre Ausdrücke verarbeitet. Zugriff auf diese Muster-Engine bietet die Klasse **Matcher**. Dazu kommt die Klasse **Pattern**, die die regulären Ausdrücke in einem vorcompilierten Format repräsentiert. Beide Klassen befinden sich im Paket java.util.regex.

Pattern.matches() bzw. String.matches()

Die statische Methode java.util.regex.Pattern.matches() und die Objektmethode matches() der Klasse String testen, ob ein regulärer Ausdruck eine Zeichenfolge komplett beschreibt.

In folgendem Beispiel wird getestet, ob eine Zeichenfolge in einfache Hochkommata eingeschlossen ist:

Ausdruck	Ergebnis
Pattern.matches("'.*'", "'Hallo Welt'")	true
"'Hallo Welt'".matches("'.*'")	true
Pattern.matches("'.*'", "''")	true
Pattern.matches("'.*'", "Hallo Welt")	false
Pattern.matches("'.*'", "'Hallo Welt")	false

Der Punkt im regulären Ausdruck steht für ein beliebiges Zeichen, und der folgende Stern ist ein Quantifizierer, der beliebig viele Zeichen erlaubt.

Regeln für reguläre Ausdrücke

Für reguläre Ausdrücke existieren eine ganze Menge von Regeln. Während die meisten Zeichen aus dem Alphabet erlaubt sind, besitzen Zeichen wie der Punkt, die Klammer, das Sternchen und einige weitere Zeichen Sonderfunktionen. So maskiert auch ein vorangestelltes »\« das folgende Sonderzeichen aus, was bei besonderen Zeichen wie ».« oder »\« wichtig ist.

[6] Vgl. zu diesem Kapitel **Java ist auch eine Insel** von Christian Ullenboom, http://openbook.rheinwerk-verlag.de/javainsel/javainsel_04_008.html, Abrufdatum: 23.09.17

Zunächst gilt es, die Anzahl an Wiederholungen zu bestimmen. Dazu dient ein Quantifizierer (auch Wiederholungsfaktor genannt). Drei wichtige gibt es:

Quantifizierer	Anzahl an Wiederholungen
X?	X kommt einmal oder keinmal vor.
X*	X kommt keinmal oder beliebig oft vor.
X+	X kommt einmal oder beliebig oft vor.

Folgende Beispiele veranschaulichen die Anwendung des Quantifizierers Sternchen (*****):

Ausdruck	Ergebnis
Pattern.matches("0", "0")	true
Pattern.matches("0", "1")	false
Pattern.matches("0", "00")	false
Pattern.matches("0*", "0000")	true
Pattern.matches("0*", "01")	false
Pattern.matches("0*", "01")	false
Pattern.matches("0*", "0*")	true

Zeichenklassen

Oftmals muss in regulären Ausdrücken ein Bereich von Zeichen abgedeckt werden, z.B. "alle Buchstaben". Aus diesem Grund gibt es die Möglichkeit, Zeichenklassen zu definieren. Beispiele für Zeichenklassen sind:

Zeichenklasse	Enthalten sind:
[aeiuo]	Zeichen a, e, i, o oder u
[^aeiuo]	**nicht** die Zeichen a, e, i, o, u
[0-9a-fA-F]	Zeichen 0, 1, 2, ..., 9 oder Groß-/Klein-Buchstaben a, b, c, d, e, f

Daneben gibt es vordefinierte Zeichenklassen, die in erster Linie Schreibarbeit ersparen.
Die wichtigsten sind:

Zeichenklasse	Enthalten ist / sind:
.	jedes Zeichen
\d	Ziffern: `[0-9]`
\D	keine Ziffern: `[^0-9]` beziehungsweise `[^\d]`
\s	whitespace: `[ \t\n\x0B\f\r]`
\S	kein whitespace: `[^\s]`
\w	Wortzeichen: `[a-zA-Z_0-9]`
\W	keine Wortzeichen: `[^\w]`

Zerlegen von Zeichenketten mit split()[7]

Die Objektmethode split() eines String-Objekts zerlegt die eigene Zeichenkette in Teilzeichenketten. Die split()-Methode delegiert dabei wie auch match() an das Pattern-Objekt. Die Trenner sind völlig frei wählbar und als regulärer Ausdruck beschrieben. Die Rückgabe ist ein Feld der Teilzeichenketten.

Beispiel: Zerlege einen Domain-Namen in seine Bestandteile:

```
String path = "www.tutego.com";
String[] segs = path.split( Pattern.quote( "." ) );
System.out.println( Arrays.toString(segs) ); // [www, tutego, com]
```

Erläuterung: Da der Punkt als Trennzeichen ein Sonderzeichen für reguläre Ausdrücke ist, muss er passend mit dem Backslash auskommentiert werden. Das erledigt die statische Methode quote(). Anderfalls würde split(".") auf jedem String ein Feld der Länge 0 liefern.

Ein häufiger Trenner ist \s, also whitespace.

Beispiel: Zähle die Anzahl der Wörter in einem Satz (der Trenner ist entweder whitespace oder ein Satzzeichen):

```
String string = "Es kann jeden treffen. Auch dich!";
int nrOfWords = string.split( "(\\s|\\p{Punct})+" ).length;
System.out.println( nrOfWords );   // 6
```

Weitere Quellen zu Regulären Ausdrücken

Interessiert finden unter https://www.kompf.de/java/regex.html

- Kurz-Übersichten zu den wichtigsten Regular Expressions,
- einen interaktiven Regex-Tester.

[7] Vgl. zu diesem Kapitel **Java ist auch eine Insel** von Christian Ullenboom, http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_04_008.htm, Abrufdatum: 04.05.18

Übungsaufgaben[8]:

Aufgabe 1: Titel in HTML-Seiten suchen

Webseiten werden textuell beschrieben. Eine einfache Webseite könnte sein:

<h1>Dies ist eine Überschrift ersten Grades</h1>

<h2>Dies ist eine Überschrift zweiten Grades</h2>

Dies ist einfach nur ein ganz normaler Abschnitt.

<h1>Dies ist noch eine Überschrift ersten Grades</h1>

Die Befehle in spitzen Klammern geben dabei den Browsern an, welche Bedeutung welcher Abschnitt einer Webseite hat.

a) Formulieren Sie einen regulären Ausdruck, der Überschriften ersten Grades sucht.
b) Formulieren Sie einen regulären Ausdruck und zusätzlich einen ersetzenden Ausdruck, um unterstrichene Texte (<u>… </u>) durch kursive Texte (…) zu ersetzen.

Aufgabe 2: Postleitzahlen erkennen

Postleitzahlen in der Schweiz sind vierstellig im Bereich von 1000..9999.

Nennen Sie einen regulären Ausdruck, der Schweizer Postleitzahlen erkennt.

Aufgabe 3: Datumserkennung anpassen

In einem Programm, das Webseiten auf Daten hin durchsucht, finden Sie folgenden regulären Ausdruck:
\d\d\.\d\d\.\d\d

a) Beschreiben Sie in Worten, welche Daten dieser Ausdruck findet und welche Daten er nicht findet.
b) Geben Sie **Beispiele** für alle Arten von Daten an, die gemäß folgender Beschreibung zulässig sind:
   - Sowohl zweistellige als auch vierstellige Jahreszahlen sind zulässig;
   - Sowohl einstellige als auch zweistellige Tag- und Monatsangaben sind zulässig.
c) **Erweitern** Sie den Ausdruck **\d\d\.\d\d\.\d\d** so, dass er Daten gemäß folgender Beschreibung erkennt:
   - Sowohl zweistellige als auch vierstellige Jahreszahlen sind zulässig;
   - Sowohl einstellige als auch zweistellige Tag- und Monatsangaben sind zulässig.

Aufgabe 4: Für schnelle Lerner

Testen Sie Ihre RegEx-Kenntnisse in einem Spiel:
- https://alf.nu/RegexGolf
- https://regexcrossword.com/

[8] Vgl. http://www.swisseduc.ch/informatik/120-lektionen/principles/computation/kara/docs/aufgaben-regulaere-ausdruecke.pdf, Abrufdatum: 24.09.17

3. Ausnahmebehandlung

Bei der Ausführung von Programmen können immer wieder Fehler auftreten, mit denen der Ersteller gerechnet hat oder auch nicht. Zum Beispiel könnte ein Nutzer anstelle eines Nettopreises (Double) versehentlich einen Text (String) eingeben. Im Programmverlauf tritt nun eine Ausnahme / ein Fehler auf, die sogenannte *InputMismatchException*.

Java bietet dem Programmierer mit dem Dreisprung aus try{}, catch{} und finally{} die Möglichkeit, diese und weitere Ausnahmen abzufangen und nach Belieben zu behandeln.

Programmierung einer Fehlerbehandlung

Dem Programmierer stehen zur Fehlerbehandlung drei Anweisungen zur Verfügung, von denen die dritte optional ist.

1. Der try-Block
   Dem regulären Programmcode, der normalerweise innerhalb der main-Methode steht, wird das Schlüsselwort **try** vorgesetzt. Es folgt ein geschweiftes Klammerpaar, innerhalb dessen der Code steht, der ausgeführt werden soll (das reguläre Programm).
2. Der catch-Block
   Treten innerhalb des try-Blocks Fehler auf, wird die Verarbeitung dort abgebrochen und die Fehler werden durch den catch-Block abgefangen. Dabei können diverse Fehler unterschieden und nacheinander behandelt werden (vgl. Abschnitt "Arten von Ausnahmen").
   Durch das Abfangen kann der Programmierer nun zum Beispiel angepasste, verständliche Fehlermeldungen ausgeben, das Programm neu starten o.ä.
3. Der finally-Block (optional)
   Der im finally-Block enthaltene Code auf jeden Fall ausgeführt, auch im Nicht-Fehlerfall.

Beispiel für eine Ausnahmebehandlung mit allen 3 Blöcken:

```
public static void main (String[] args)
{
    try
    {   byte zahl1, zahl2;
        Scanner eingabe = new Scanner(System.in);

        zahl1 = eingabe.nextByte();
        zahl2 = eingabe.nextByte();

        System.out.println(zahl1 / zahl2);
    }

    catch (Exception e)
    {
        System.out.println("Folgender Fehler ist aufgetreten:" + e);
    }

    finally
    {
        System.out.println("Vielen Dank für die Nutzung unserer Software!");
    }
}
```

Arten von Ausnahmen

Im obigen Beispiel gibt es nur einen catch-Block, der alle bekannten Fehler (Exceptions) abfängt. Es ist jedoch zulässig und sinnvoll, mehrere catch-Blöcke nacheinander zu formulieren, um gezielt mit unterschiedlichen Fehlern umgehen zu können. Es empfiehlt sich dabei, zunächst die speziellen und anschließend die allgemeinen Ausnahmen zu behandeln, da die catch-Blöcke sequentiell abgearbeitet werden.

Einige ausgewählte Fehler (Exceptions) sind:

Exception	Beschreibung
ArithmeticException	Es wurde eine unzulässige Rechenoperation durchgeführt, z.B. das Teilen durch 0.
BufferOverflowException	Es wurde versucht, mehr zu speichern als die Kapazität des Buffers zuläßt.
ClassCastException	Es wurde versucht, auf ein Objekt ein Cast-Konstrukt anzuwenden, dessen Typ inkompatibel zu dem Objekt ist.
IllegalArgumentException	Es wurde versucht, einer Methode einen ungültigen Parameterwert zu übergeben.
IndexOutOfBoundsException	Ein Array wurde mit einem ungültigen Index angesprochen.
InputMismatchException	Vgl. Einführungsbeispiel

Übungsaufgaben:

Aufgabe 1: Koordinaten

Schreiben Sie ein Programm zur Erfassung von Koordinaten (Längen- und Breitengrade). Diese werden in einem Array gespeichert und zum Abruf bereitgehalten.

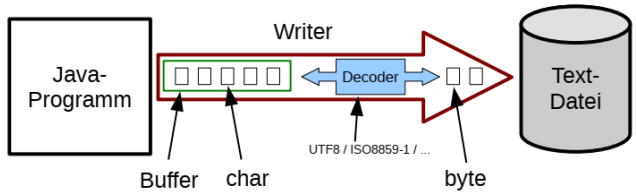
Implementieren Sie eine Ausnahmebehandlung für *InputMismatchExceptions* und *IndexOutOfBoundsExceptions*. Weitere Ausnahmen, die Sie nicht vorhersehen, sollen auch abgefangen werden.

4. Lesen und Schreiben von Textdateien (IO)

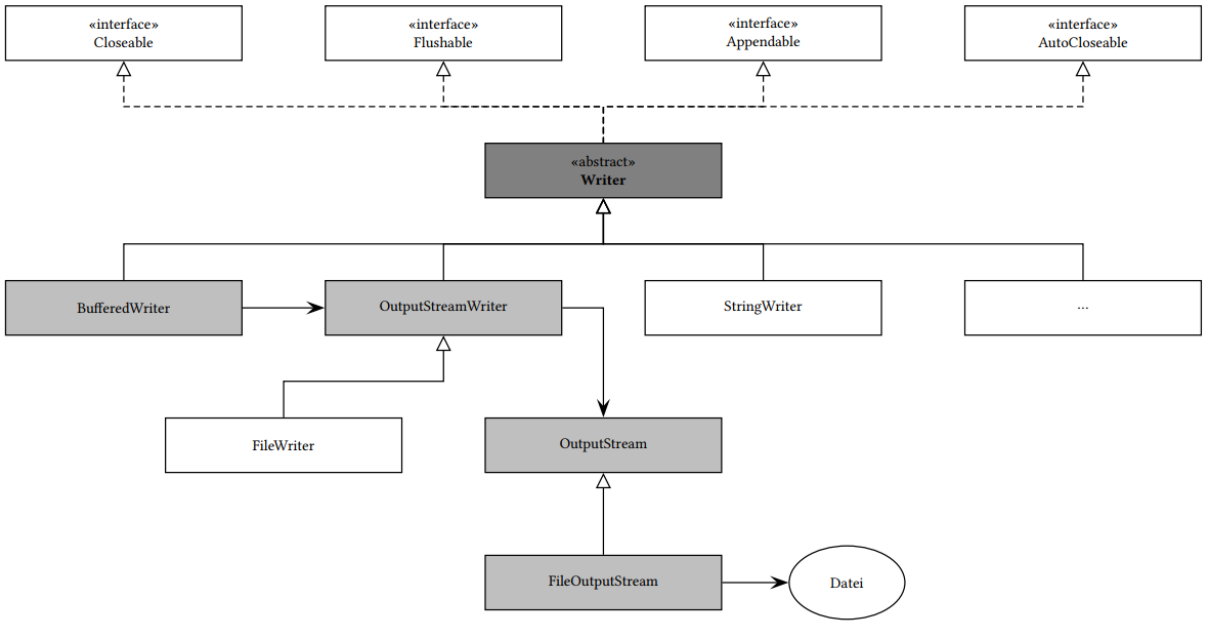
Für Textdateien stehen die Klassen *Reader* zum Lesen und *Writer* zum Schreiben zur Verfügung.

4.1 Textdateien schreiben

Für das Schreiben in eine Textdatei wird die abstrakte Klasse *Writer* bzw. deren Ableitungen verwendet.



Damit das Schreiben effektiver funktioniert, wird zusätzlich ein Buffer verwendet, der die geschriebenen Zeichen blockweise weiter gibt. Standardmäßig hat der Puffer (Buffer) eine Größe von 8192 Bytes, kann aber beliebig angepasst werden. Die Zeichen (char – 16 bit – 2 bytes) werden dabei vom *Writer* an den Decoder (hier die Klasse *OutputStreamWriter*) weitergeleitet, der die Zeichen entsprechend der Dekodiertabelle in Bytes umwandelt und diese wiederum an einen *OutputStream* weiterleitet, damit die Daten in einer Datei landen.



Warum ist dies so kompliziert?

Das liegt daran, dass Text mit unterschiedlicher Kodierung, auch Encoding genannt (z. B. UTF-8, ISO8859-15, ASCII, …) erstellt werden kann.1 Ein normaler String in Java wird mit UTF-16 kodiert (16-Bit), daher hat auch ein char die „Breite“ von 16-Bit. Ein nor- males Dateisystem verwendet aber für die Speicherung immer Bytes, also 8-Bit. Somit muss die entsprechende Kodierung in ein 8-Bit Schema umgewandelt werden, welches die Klasse OutputStreamWriter mit einem entsprechenden Kodierer erledigt.

Code-Beispiel

```
try {
   File f = new File("datei.txt");
   BufferedWriter out = new BufferedWriter(
         new OutputStreamWriter(
               new FileOutputStream(f), "UTF8"));
   String text = "Dies ist ein Text mit Umlauten: öäüß ÖÄÜ!";
   out.write(text);
   out.newLine();
   out.close();
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

Erläuterung: Über die Verbindung *BufferedWriter* zu *OutputStreamWriter* und dann zu *FileOutputStream* wird jedes Zeichen (char mit 16-bit) in entsprechende Bytes umgewandelt. Wenn der Puffer voll ist (oder flush() aus dem Interface *Flushable* bzw. *close*() aus dem Interface *AutoCloseable* bzw. *Closeable* aufgerufen wird), werden die Zeichen in die Datei geschrieben. Die beiden Methoden leiten die Aktion dabei an den nächsten *Writer*, hier *OutputStreamWriter* weiter, usw.

Ausgabe der Datei 'datei.txt' mit dem Programm hexdump:

```
hexdump -C datei.txt
```

```
00000000  44 69 65 73 20 69 73 74  20 65 69 6e 20 54 65 78  |Dies ist ein Tex|
00000010  74 20 6d 69 74 20 55 6d  6c 61 75 74 65 6e 3a 20  |t mit Umlauten: |
00000020  c3 b6 c3 a4 c3 bc c3 9f  20 c3 96 c3 84 c3 9c 21  |........ ......!|
00000030  0a                                                |.|
00000031
```

Bei der Kodierung UTF-8 werden Zahlen, normale Buchstaben etc. mit 8-Bit gespeichert, Umlaute beispielsweise mit 16-Bit, also zwei Bytes (ö=c3 b6, ä=c3 a4, …).

Methoden von „Writer" (Auszug)

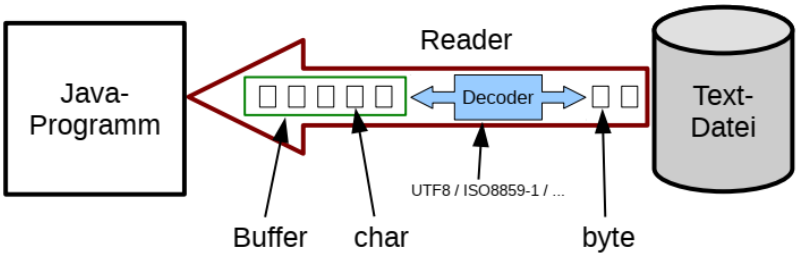
- *append()* Hängt Zeichen, Zeichenketten, Arrays etc. an.
- *close()* Schließt den Stream (zuvor wird flush() aufgerufen).
- *flush()* Leert einen Puffer.
- *write()* Schreibt Zeichen, Zeichenketten, Arrays etc. in den Stream.

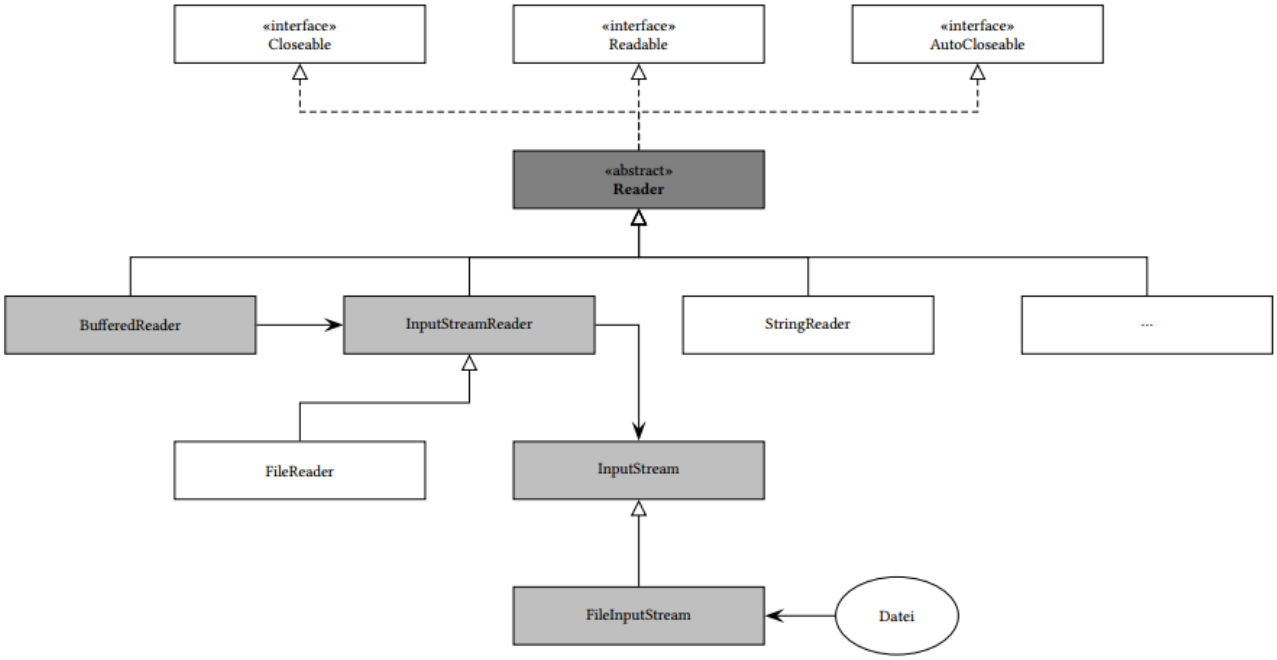
Methoden von „BufferedWriter" (zusätzlich)

- *newLine()* Schreibt die Zeilenendemarkierung
  Windows: carriage return (CR) + newline (NL) (0x0d + 0x0a) (\r + \n)
  Linux: NL (0x0x) (\n)
  Mac: CR (0x0d) (\r)

4.2 Textdateien lesen

Das Lesen aus einer Datei funktioniert sehr ähnlich wie das Schreiben. Wichtig dabei ist, dass man dieselbe Kodierung verwendet wie beim Speichern.





Code-Beispiel

```
try {
   File f = new File("datei.txt");
   BufferedReader in = new BufferedReader(
      new InputStreamReader(
         new FileInputStream(f), "UTF8"));
   String line;
   while ((line = in.readLine()) != null) {
      System.out.print(line);
   }
   in.close();
} catch (IOException e) {
   System.err.println(e.getMessage());
   System.exit(1);
}
```

```
Dies ist ein Text mit Umlauten: öääüß ÖÄÜ!
```

Methoden von „Reader“ (Auszug)

- *read()* Liest ein Zeichen, ein Array etc. ein.
- *close()* Schließt den Stream.
- *reset()* Das Lesen beginnt wieder von vorne.
- *skip()* Es werden n-Zeichen übersprungen.

Methoden von „BufferedReader“ (zusätzlich)

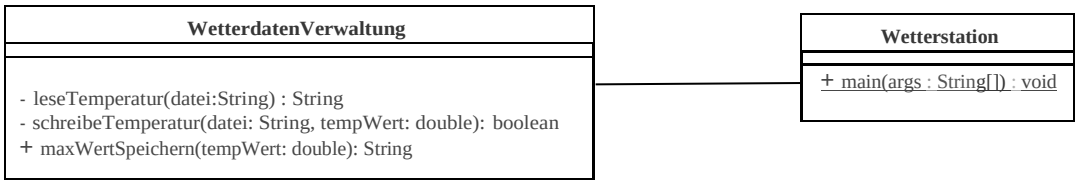
- *readLine()* Liest eine komplette Zeile ein. Die Zeilenendemarkierung wird dabei automatisch erkannt.

Übungsaufgaben:

Aufgabe 1: Wetterstation[9]

Eine Wetterstation möchte die täglich aufgenommenen Wetterdaten programmiertechnisch erfassen und verwalten.

In der ersten Testphase, in der die eingelesene Höchsttemperatur gespeichert werden soll, ist folgendes Programm geplant:



public class WetterdatenVerwaltung

```
- leseTemperatur(datei: String) : String
```

Methode für den lesenden Zugriff auf eine Datei.
Die Methode greift auf die durch den Übergabeparameter festgelegte Datei zu und liest den darin vorhandenen Wert aus. Dieser Wert wird an die aufrufende Stelle zurückgegeben.
Parameter:

- Dateiname (mit Dateierweiterung) als String

Rückgabewerte:

- Temperaturwert (aus der angegebenen Datei) als String
- "*Fehler beim Dateizugriff! – Lesefehler*!", wenn Lesezugriff IOException verursacht
- "*Fehler beim Dateizugriff! – angegebene Datei nicht vorhanden*!", wenn Datei nicht gefunden wurde (FileNotFoundException)

```
- schreibeTemperatur(datei:String, tempWert:double) : boolean
```

Methode für den schreibenden Zugriff auf eine Datei.
Die Methode greift auf die durch den Übergabeparameter festgelegte Datei zu und schreibt den als zweiten Parameter übergebenen Wert in die Datei. Ein bestehender alter Wert wird dabei überschrieben. Die Methode liefert einen boolean-Wert als Bestätigung zurück.
Parameter:

- Dateiname (mit Dateierweiterung) als String
- Temperaturwert als double

Rückgabewerte:

- *true,* wenn Speichervorgang erfolgreich abgeschlossen wurde
- *false,* bei IOException

[9] © Zm

`+ maxWertSpeichern(tempWert:double) : String`

Methode zum Abspeichern des maximalen Temperaturwertes. Der übergebene Temperaturwert wird mit dem in der Datei "**maxtemp.txt**" gespeicherten Wert verglichen. Ist der neue Temperaturwert höher, soll dieser in der Datei gespeichert werden. Zum Lesen des Wertes aus der Datei und zum Abspeichern des neuen Wertes werden die Methoden leseTemperatur(...) und schreibeTemperatur(...) der Klasse verwendet. Der aktuell gespeicherte Maximalwert (bzw. ein Fehlerwert) wird zurückgegeben.
Parameter:

- Temperaturwert als double

Rückgabewerte:

- aktuell gespeicherter Temperaturwert (maximale Temperatur) als String
- "*Fehler beim Dateizugriff! – Lesefehler*!", wenn Lesezugriff IOException verursacht
- "*Fehler beim Dateizugriff! – angegebene Datei nicht vorhanden*!", wenn Datei nicht gefunden wurde (FileNotFoundException)
- *"Fehler beim Dateizugriff – Schreibfehler!*", wenn Schreibzugriff IOException verursacht

public class Wetterstation

Ausführbare Klasse zum Starten des Wetterdatenprogramms.
Innerhalb der main-Methode soll ein neuer Temperaturwert mittels Scanner eingelesen werden. Anschließend soll der neue Wert durch Aufruf der Methode maxWertSpeichern(…) mit dem bereits in der Datei "**maxtemp.txt**" gespeicherten Wert verglichen werden.

Durch entsprechende Bildschirmanzeigen sollen dem Benutzer Informationen über die Verarbeitung gegeben werden (aktuelle gespeicherte Maximaltemperatur, Fehlermeldung).

Beispiele für die Bildschirmanzeige:

- Fehlerlose Verarbeitung

```
Temperaturwert eingeben
12.5
------------------------------------------------------------
maximale Temperatur in °C:
25
```

- Fehler beim Lesen der Datei

```
Temperaturwert eingeben
12.5
------------------------------------------------------------
maximale Temperatur in °C:
Fehler beim Dateizugriff! - Lesefehler
```

```
Temperaturwert eingeben
12.5
------------------------------------------------------------
maximale Temperatur in °C:
Fehler beim Dateizugriff - angegebene Datei nicht vorhanden!
```

- Fehler beim Schreiben der Datei

```
Temperaturwert eingeben
12.5
------------------------------------------------------------
maximale Temperatur in °C:
Fehler beim Dateizugriff! - Schreibfehler
```

Arbeitsauftrag

Erstellen Sie entsprechend der gegebenen Beschreibungen das Programm für die Wetterdatenverwaltung in zwei Schritten.

1. Schreiben Sie die Klasse WetterdatenVerwaltung mit den Methoden leseTemperatur( ) und schreibeTemperatur( ) und setzen Sie diese zunächst auf public. Testen Sie die beiden Methoden mithilfe einer eigenen ausführbaren Testdatei.
2. Verändern bzw. erweitern Sie die Klasse WetterdatenVerwaltung so, dass sie den gegebenen Beschreibungen entspricht. Legen Sie die Klasse Wetterstation an. Testen Sie ihr Programm.