3 Strings

Wir haben schon ein paar Mal Strings verwendet, ohne viel darüber nachzudenken. Als kurze Erinnerung: Variablen vom Typ `String` modellieren Zeichenketten und dienen zur Verwaltung von textuellen Informationen. In diesem Kapitel schauen wir uns die Thematik genauer an und beginnen unsere Entdeckungsreise zu Zeichenketten.

3.1 Schnelleinstieg

3.1.1 Gebräuchliche Stringaktionen

Stringkonkatenation

Ein sehr gebräuchlicher Anwendungsfall ist das Zusammenfügen von Strings, auch Konkatenieren genannt. Dazu dient bekanntlich der Operator '+'. Nachfolgend kombinieren wir den Vor- und Nachnamen des Autors mit einem Abstand von einem Leerzeichen:

```
jshell> String firstName = "Michael"
firstName ==> "Michael"

jshell> String lastName = "Inden"
lastName ==> "Inden"

jshell> System.out.println(firstName + " " + lastName)
Michael Inden
```

Für Strings existiert alternativ noch eine Methode namens `concat()`, mit der man funktionell das Gleiche erreicht, allerdings leidet die Lesbarkeit, wie dies nachfolgendes Beispiel deutlich zeigt:

```
jshell> String fullName = firstName.concat(" ").concat(lastName)
fullName ==> "Michael Inden"
```

Besonderheiten Beim Erstellen von Texten aus einzelnen Teilbausteinen ist der Operator '+' ziemlich nützlich. Allerdings sollte man dabei ein paar Besonderheiten kennen. Betrachten wir eine Problematik für einen Text, der Auskunft über die Länge einer Nachricht geben soll. Dazu sollen hier einem textuellen Hinweis die Summe aus zwei Zahlen hinzugefügt werden, um die Gesamtlänge einer Nachricht bereitzustellen.

Das folgende Listing zeigt einen ersten Versuch, die Summe der Zahlen anzufügen:

```
jshell> int payloadLength = 42
payloadLength ==> 42

jshell> "Complete Msg-Length: " + payloadLength + 10
$2 ==> "Complete Msg-Length: 4210"
```

Das Ergebnis könnte vielleicht überraschen: Es wird zunächst das erste '+' ausgeführt. Dabei wird die Zahl in einen String verwandelt und dann zum ursprünglichen String hinzugefügt. Das wiederholt sich ebenso für die zweite Zahl. Dadurch erhalten wir das zunächst verwirrende oder zumindest überraschende Ergebnis `4210`.

Das liegt daran, dass wir hier Strings konkatenieren und keine mathematischen Operationen ausführen. Zur Korrektur, also um die Summe der Zahlen dem String hinzufügen, müssen wir den Ausdruck klammern und Folgendes schreiben:

```
jshell> "Complete Msg-Length: " + (payloadLength + 10)
$9 ==> "Complete Msg-Length: 52"
```

Als Nächstes könnten wir statt der Addition zwei Zahlenwerte subtrahieren und zum String hinzufügen wollen. Das funktioniert nicht: Wie gerade schon erkannt, erfolgt die Abarbeitung von links nach rechts. Zudem ist der Operator '–' für Strings nicht definiert. Versuchen wir es trotzdem:

```
jshell> "Complete Msg-Length: " + payloadLength - 10
|  Error:
|  bad operand types for binary operator '-'
|    first type:  java.lang.String
|    second type: int
|  "Complete Msg-Length: " + payloadLength - 10
|  ^-----------------------------------------^
```

Erneut bringt eine Klammerung das korrekte Ergebnis:

```
jshell> "Complete Msg-Length: " + (42 - 10)
$10 ==> "Complete Msg-Length: 32"
```

Groß- und Kleinschreibung

In der Praxis benötigt man immer mal wieder eine Umwandlung von Groß- in Kleinbuchstaben oder andersherum. Dabei helfen die beiden Methoden `toUpperCase()` und `toLowerCase()`. Zuerst wandeln wir einen Text vollständig in Großbuchstaben um und danach dann in Kleinbuchstaben.

```
jshell> String message = "IMPORTANT: Please consult the doctor"
message ==> "IMPORTANT: Please consult the doctor"

jshell> message.toUpperCase()
$26 ==> "IMPORTANT: PLEASE CONSULT THE DOCTOR"

jshell> message.toLowerCase()
$27 ==> "important: please consult the doctor"
```

Besonderheiten: Keine Modifikation erlaubt Bitte beachten Sie, dass Strings in Java als unveränderliche Objekte realisiert sind. Eine Modifikation ist daher nicht möglich. Als Abhilfe kann man einen neuen String mit verändertem Inhalt erzeugen:

```
jshell> String hint = "Immutable String"
hint ==> "Immutable String"

jshell> String result = hint.toUpperCase()
result ==> "IMMUTABLE STRING"

jshell> hint
hint ==> "Immutable String"
```

Leerzeichen entfernen

Mitunter sollen am Anfang oder Ende oder an beiden Seiten unerwünschte Leerzeichen entfernt werden. Dazu kann man die Methoden `strip()` sowie `stripLeading()` und `stripTrailing()` aufrufen. Mitunter findet man noch die Methode `trim()`, die analog zu `strip()` funktioniert, jedoch einige neuere Unicode-Zeichen nicht korrekt verarbeitet.

```
jshell> String valueWithBlanks = "  blanks at the beginning and the end  "
value_with_blanks ==> "  blanks at the beginning and the end  "

jshell> valueWithBlanks.strip()
$96 ==> "blanks at the beginning and the end"

jshell> valueWithBlanks.stripLeading()
$97 ==> "blanks at the beginning and the end  "

jshell> String result = valueWithBlanks.stripTrailing()
result ==> "  blanks at the beginning and the end"

jshell> valueWithBlanks
valueWithBlanks ==> "  blanks at the beginning and the end  "
```

Beachten Sie bitte auch hier, dass durch die Methodenaufrufe der ursprüngliche String nicht verändert wird, sondern ein neuer String als Ergebnis entsteht. Dies folgt aus den Ausgaben, insbesondere der letzten, die einen unveränderten Inhalt der Variablen `valueWithBlanks` zeigt.

Länge ermitteln

Auch die Gesamtlänge eines Strings ist des Öfteren eine wichtige Information. Die Gesamtzahl der Zeichen liefert ein Aufruf von `length()`:

```
jshell> String content = "This is a short message"
content ==> "This is a short message"

jshell> content.length()
$155 ==> 23
```

Auf leeren String prüfen

Zwar kann man mithilfe der zuvor vorgestellten Methode auch prüfen, ob ein String leer ist:

```
jshell> String noContent = ""
noContent ==> ""

jshell> noContent.length() == 0
$55 ==> true
```

Das ist jedoch nicht so schön lesbar und wird eher als schlechter Stil angesehen. Für Strings gibt es dazu die Methoden `isEmpty()` und `isBlank()`.

```
jshell> noContent.isEmpty()
$56 ==> true

jshell> noContent.isBlank()
$57 ==> true

jshell> " \t".isEmpty()
$58 ==> false

jshell> " \t".isBlank()
$59 ==> true

jshell> " \t".strip().isEmpty()
$60 ==> true
```

Im vorletzten Beispiel sieht man, dass `isBlank()` auch Strings nur mit Whitespace (hier Leerzeichen und Tabulator) als leer ansieht. Die Methode `isEmpty()` prüft dagegen auf einen leeren String, also einen ohne Zeichen. Der letzte Aufruf zeigt folgenden Zusammenhang: Wenn `str.isBlank()` den Wert `true` (oder den Wert `false`) liefert, dann gilt dies auch für `str.strip().isEmpty()`. Mathematisch gilt:

$$str.isBlank() \iff str.strip().isEmpty()$$

Auf einzelne Zeichen zugreifen

Auf die einzelnen Buchstaben eines Strings kann man positionsbasiert zugreifen, wobei der erste Buchstabe den Index 0 besitzt. Nachfolgend lesen wir die ersten drei Zeichen des Strings "This is a short message" aus:

```
jshell> content.charAt(0)
$156 ==> 'T'

jshell> content.charAt(1)
$157 ==> 'h'

jshell> content.charAt(2)
$158 ==> 'i'
```

Keine Modifikation erlaubt Bitte beachten Sie, dass keine korrespondierende Methode `setCharAt()` existiert, um Zeichen in einem String zu modifizieren. Wie schon erwähnt, sind Strings in Java als unveränderliche Objekte realisiert. Eine Variante, um Strings doch zeichenbasiert ändern zu können, zeige ich Ihnen später.

Teilbereiche extrahieren

Für einige Anwendungsfälle muss man auf Bestandteile eines Texts zugreifen, die durch eine Anfangs- und optional eine Endposition bestimmt sind. Dazu bietet sich die Methode `substring()` an. Dieser übergibt man entweder Start- (inklusiv) und Endposition (exklusiv) oder aber nur den Start, wodurch dann der Text ab dieser Position bis zum Ende als neuer String geliefert wird:

```
jshell> String info = "Dies ist ein String. Rest ABC"
info ==> "Dies ist ein String. Rest ABC"

jshell> info.substring(0,4)
$100 ==> "Dies"

jshell> info.substring(9,19)
$103 ==> "ein String"

jshell> info.substring(19)
$104 ==> ". Rest ABC"
```

Strings wiederholen

Manchmal ist es notwendig, einen Text n-mal wiederholen zu können. Das ist zwar problemlos mit eine `for`-Schleife möglich, jedoch existiert die recht praktische Methode `repeat()`, womit die Aufgabe erleichtert wird:

```
jshell> String greeting = "MOIN"
greeting ==> "MOIN"

jshell> greeting.repeat(2)
$109 ==> "MOINMOIN"
```

3.1.2 Suchen und Ersetzen

Suchen und Enthaltensein

Ab und an möchte man feststellen, ob ein gewisser Text oder Buchstabe in einem String enthalten ist. Mithilfe der Methode `indexOf()` erhält man die entsprechende Position bzw. den Wert -1 für »nicht gefunden«. Eine Prüfung vom Ende des Strings ermöglicht die Methode `lastIndexOf()`. Soll lediglich geschaut werden, ob der gesuchte Teilstring enthalten ist, bietet sich die Methode `contains()` an:

```
jshell> String maintext = "This is a secret message. Please do not distribute"
maintext ==> "This is a secret message. Please do not distribute"

jshell> maintext.indexOf("This")
$166 ==> 0

jshell> maintext.indexOf("Please")
$167 ==> 26

jshell> maintext.indexOf('o')
$168 ==> 34

jshell> maintext.lastIndexOf('o')
$169 ==> 37

jshell> maintext.contains("not")
$170 ==> true

jshell> maintext.contains("MICHAEL")
$171 ==> false
```

Im Beispiel sehen wir noch ein Detail, nämlich die unterschiedlichen Anführungszeichen. Mit einfachen Anführungszeichen entsteht immer ein einzelnes Zeichen vom Typ `char`. Benutzen wir doppelte Anführungszeichen, so ist dies immer ein String. Dieser kann aber auch genau ein Zeichen lang sein:

```
jshell> char oneSingleCharacter = 'A'
oneSingleCharacter ==> 'A'

jshell> String stringWithOneChar = "A"
stringWithOneChar ==> "A"
```

Weitersuchen Auf eine praktische Besonderheit möchte ich im Zusammenhang mit `indexOf()` noch eingehen. Zunächst einmal sei angemerkt, dass die Methode `indexOf()` nur nach dem ersten Vorkommen sucht. Auch wiederholtes Aufrufen ändert die Fundstelle nicht und man kann so keine anderen Vorkommen finden.

Praktischerweise gibt es aber eine Variante der Methode, der man eine Startposition übergeben kann. Auf diese Weise lässt sich problemlos die Funktionalität »Suchen und Weitersuchen« realisieren, indem man immer nach der Fundstelle weitersucht. Dazu übergibt man einfach die gelieferte Position + 1 wie folgt:

```
jshell> info = "one second, one hour and one day"
info ==> "one second, one hour and one day"

jshell> info.indexOf("one")
$43 ==> 0

jshell> info.indexOf("one", 1)
$44 ==> 12

jshell> info.indexOf("one", 13)
$45 ==> 25
```

Ersetzen von Inhalten

Neben dem Suchen und dem Test auf Enthaltensein möchte man manchmal auch Teile eines Strings ersetzen. Dabei ist die Methode `replace()` hilfreich. Diese ersetzt eine gewünschte Zeichenfolge durch die als zweiten Parameter übergebene Zeichenfolge:

```
jshell> String greeting = "MOIN MOIN"
greeting ==> "MOIN MOIN"

jshell> greeting.replace("MOIN", "GRÜEZI")
$188 ==> "GRÜEZI GRÜEZI"
```

Weil Strings unveränderlich sind, wird auch hier als Ergebnis wieder ein neuer String mit dem veränderten Inhalt erzeugt.

Komplexeres Ersetzen von Inhalten

Ergänzend gibt es die Methode `replaceAll()`, die als Suchzeichenfolge einen regulären Ausdruck nutzt – um beispielsweise Texte, die mit einem A beginnen, gefolgt von einem beliebigen Buchstaben und einem C oder D, durch einen Leerstring zu ersetzen. Für dieses Buch reicht es, zu wissen, dass der '.' im regulären Ausdruck einen Platzhalter für ein einzelnes, beliebiges Zeichen darstellt und dass man eine Menge von Alternativen in eckigen Klammern angeben kann.[1]

```
jshell> String infoRegEx = "ACC_AEC_MUSIC_ABC_AWARD".replaceAll("A.[CD]", "")
infoRegEx ==> "__MUSIC__AW"
```

Gibt man bei `replaceAll()` lediglich den Text an, so wirkt wieder die exakte Übereinstimmung des Musters – hier um alle Vorkommen von "`ABC`" durch "`--`" zu ersetzen:

```
jshell> String infoPlain = "ABCABC_MUSIC_ABCAWARD".replaceAll("ABC", "--")
infoPlain ==> "----_MUSIC_--AWARD"
```

3.1.3 Informationen extrahieren und formatieren

Immer mal wieder enthält ein Text mehrere Informationsbestandteile, etwa eine Uhrzeitangabe mit Stunden, Minuten und Sekunden jeweils durch '`:`' getrennt:

```
jshell> String timestamp = "11:22:33"
timestamp ==> "11:22:33"
```

Unser Ziel ist es, die Einzelbestandteile als Zahlen zu ermitteln. Wie kann man dazu vorgehen? Um die Informationen aufzuspalten und auszulesen, bietet sich die Methode `split()` an:

```
jshell> String[] parts = timestamp.split(":")
parts ==> String[3] { "11", "22", "33" }
```

[1] Weitere Details finden Sie in meinem Buch »Der Weg zum Java-Profi« [3].

Dieser übergibt man im einfachsten Falle ein Trennzeichen (tatsächlich ist es ein regulärer Ausdruck, wie dies im Praxistipp thematisiert wird) und erhält als Ergebnis eine Abfolge (genauer: ein Array) von Strings – Arrays werden wir dann in Kapitel 4 genauer kennenlernen:

Tipp: Besonderheiten

Das Extrahieren von Informationen mit `split(":")` scheint einfach zu sein. Versuchen wir uns an der Extraktion der Werte einer Datumsangabe:

```
jshell> String dateInfo = "23.11.2020"
dateInfo ==> "23.11.2020"
```

Probieren wir das mit `split()` wie zuvor aus:

```
jshell> String[] dateParts = dateInfo.split(".")
dateParts ==> String[0] {  }
```

Merkwürdig, wieso ist das Ergebnis leer? Das liegt daran, dass man `split()` einen regulären Ausdruck übergibt. Wenn also alle Zeichen (`"."`) als Trennzeichen behandelt werden, verbleiben keine Nutzzeichen. Um wirklich den Punkt als Zeichen zu verwenden, müssen wir diesem ein `\` voranstellen. Man spricht auch von escapen. Weil der `\` wiederum in Java-Strings ein besonderes Zeichen ist, müssen wir dies ebenfalls escapen und wiederum ein `\` voranstellen:

```
jshell> String[] dateParts = dateInfo.split("\\.")
dateParts ==> String[3] { "23", "11", "2020" }
```

Formatierte Ausgabe

Wenn man Zahlen und Texte formatiert aufbereiten möchte, kann die Methode `format()` eine gute Hilfe sein. Als ersten Parameter übergibt man einen Text mit verschiedenen Platzhaltern, die auch Formatierungsangaben enthalten können. Mit den weiteren Parametern gibt man für diese Platzhalter die entsprechenden Werte an. Beim Aufruf werden diese dann mit den übergebenen Werten befüllt.

Das mag kompliziert klingen, beginnen wir also besser mit einem Beispiel:

```
jshell> String.format("Integer-Value: %d", 42)
$48 ==> "Integer-Value: 42"
```

Mit dem Formatbezeichner `%d` können Sie als Argument einen beliebigen ganzzahligen Typ verwenden. Es gibt eine Vielzahl weiterer Platzhalter. Nachfolgend sind einige für Strings, Zahlen und auch Datumsangaben gezeigt. Die gesamte Liste ist so ausführlich, dass sich ein Blick in die Onlinedokumentation lohnt, außerdem weicht die Formatierung gegebenenfalls geringfügig abhängig von ihrer Locale, also Deutschland, Schweiz usw., ab.

```
jshell> import java.time.*

jshell> String.format("%s is %d years old. Birthday: %tD", "Michael", 50,
      LocalDate.of(1971, 2, 7))
$52 ==> "Michael is 50 years old. Birthday: 02/07/71"
```

Auf einige Besonderheiten möchte ich noch eingehen: Es ist recht einfach möglich, etwa die Länge einer Ganzzahl mit `%<Anzahl>d` zu beschränken sowie für Gleitkommazahlen die Anzahl an Nachkommastellen mit `%.<Anzahl>f`. Nachfolgend sehen wir eine Begrenzung auf 10 Stellen bei einem Gehalt und zwei Nachkommastellen für PI.

```
jshell> String.format("%s's salary is |%10d| a year. What about PI? %.2f", "Tom"
      , 123_456_789, Math.PI)
$350 ==> "Tom's salary is | 123456789| a year. What about PI? 3.14"
```

Anstatt den formatierten String zunächst über `String.format()` aufzubereiten

```
jshell> String michasBirthday = String.format("%s's birthday: %tD", "Michael",
      LocalDate.of(1971, 2, 7))
michasBirthday ==> "Michael's birthday: 02/07/71"
```

und mit `System.out.println()` auszugeben, gibt es mit `System.out.printf()` eine kürzere, direktere Variante, bei der man jedoch noch einen Zeilentrenner mit `%n` oder `\n` angeben muss – sofern man einen Zeilenumbruch wie zuvor erhalten möchte:

```
jshell> System.out.printf("%s's birthday: %tD%n", "Michael", LocalDate.of(1971,
      2, 7))
Michael's birthday: 02/07/71
```

3.2 Nächste Schritte

Bevor wir uns mit einigen weiteren Details der Verarbeitung von Strings beschäftigen, möchte ich nochmals auf die umfangreiche Onlinehilfe zum JDK verweisen. Beispielsweise finden Sie diverse Erläuterungen zur Klasse `String` auf der Seite `https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/String.html`. Das gilt ebenfalls für die im Anschluss besprochene Klasse `Scanner`. Für diese finden Sie auf der Seite `https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Scanner.html` weitere Informationen. Generell bietet die Startseite `https://docs.oracle.com/en/java/javase/16/docs/api/` einen Einstieg. Von dort können Sie dann passend zu den gewünschten Informationen navigieren. Seit Java 9 bietet die Onlinehilfe auch eine Suche in der oberen rechten Ecke.

3.2.1 Die Klasse `Scanner`

Manchmal möchte man nicht nur Informationen per `System.out` auf die Konsole schreiben, sondern es sollen auch Benutzereingaben entgegengenommen werden. Dabei hilft die recht praktische Klasse `Scanner` in Kombination mit `System.in`.

Die Klasse `Scanner` kann Eingabestrings aus verschiedenen Datenquellen in einzelne Bestandteile, sogenannte ***Tokens***, zerlegen und dabei primitive Typen direkt aus der Eingabe parsen. Bei der Konstruktion eines `Scanner`-Objekts ist die gewünschte Eingabequelle zu übergeben. Diese kann vom Typ `String`, `File`, `InputStream` oder `Readable` sein. Nach der Konstruktion lässt sich über die Methode `useDelimiter(String)` spezifizieren, welcher reguläre Ausdruck die Trennzeichenfolge beschreibt. Sofern dies nicht geschieht, wird als Trennzeichenfolge jede Folge von Whitespace-Zeichen (Spaces, Tabs und Zeilenumbruch) verwendet.

Scanner und Systemressourcen

Beim Einsatz des Scanners muss man beachten, dass dieser Systemressourcen belegt, die man normalerweise auch wieder freigeben muss. Dabei hilft das sogenannte ARM (Automatic Resource Management), das ausführlicher in Abschnitt 11.3 beschrieben wird. Hier reicht das Wissen, dass man einen Scanner wie folgt in einem Java-Programm verwendet:

```
try (Scanner scanner = new Scanner(source))
{
    // Anweisungen
}
```

Durch dieses `try` mit den runden Klammern wird von Java automatisch die Ressource wieder korrekt freigegeben.

Daten extrahieren

Um Daten aus einer Eingabe zu extrahieren, bietet die Klasse `Scanner` verschiedene Methoden. Die Methode `next()` gibt das nächste Token als `String` zurück. Dies ist jedoch nur dann sicher möglich, wenn die Methode `hasNext()` die Existenz eines nachfolgenden Elements bestätigt hat. Ansonsten wird eine `NoSuchElementException` ausgelöst, wenn keine weiteren Daten mehr folgen.

```
jshell> try (final Scanner scanner = new Scanner("Dies ist ein Test"))
   ...> {
   ...>     while (scanner.hasNext())
   ...>         System.out.println(scanner.next());
   ...> }
Dies
ist
ein
Test
```

Beispiel: Texte mit Scanner verarbeiten

Wir nutzen nun die Trennzeichenfolge aus den Zeichen '.', '_' und '–' in Form eines regulären Ausdrucks, um eine Versionsinformation, etwa `Version-2.17_45`, in Einzelbestandteile aufzusplitten.

```
try (final Scanner scanner = new Scanner("Version-2.17_45"))
{
    scanner.useDelimiter("\\.|_|-");
    while (scanner.hasNext())
    {
        System.out.print(scanner.next() + " ");
    }
}
```

Führt man die Programmzeilen aus, so erhält man folgende Ausgabe:

```
Version 2 17 45
```

Beispiel: Informationen mit Scanner verarbeiten

Das Einlesen von Werten primitiver Datentypen erledigen `next<Typ>()`-Methoden, etwa `nextLong()`. Analog zur `hasNext()`-Methode existieren `hasNext<Typ>()`-Methoden, z. B. `hasNextLong()`. Durch deren Aufruf lässt sich feststellen, ob ein weiteres Token vom gewünschten Typ in der Eingabe vorhanden ist. Liest man jedoch ungeprüft beispielsweise einen `long`-Wert aus einer beliebigen Eingabequelle und entspricht der gelesene Wert nicht dem erwarteten Typ, so reagiert der `Scanner` mit einer `InputMismatchException`.

Wir verändern die Eingabe ein wenig, um nun auch ein paar weitere Verarbeitungsmöglichkeiten kennenzulernen. Als Trennzeichen dienen Doppelpunkt, Unterstrich, Minus und Zeilenumbruch:

```
try (final Scanner scanner = new Scanner("Version:2:17.45\nLine2"))
{
    scanner.useDelimiter(":|_|-|\\n");

    var text = scanner.next();
    var intValue = scanner.nextInt();
    var floatValue = scanner.nextFloat();
    var remaining = scanner.next();

    System.out.print(text + " / " + intValue + " / " +
                     floatValue + " / " + remaining);
}
```

Das führt zu folgenden Ausgaben:

```
Version / 2 / 17.45 / Line2
```

Beispiel: Daten zeilenweise mit Scanner verarbeiten

Außerdem lassen sich Daten mit der Methode `nextLine()` zeilenweise verarbeiten.

```
try (var scanner = new Scanner("Line1,Info\nLine2.Special\nLine3:Additional"))
{
    var line1 = scanner.nextLine();
    var line2 = scanner.nextLine();
    var line3 = scanner.nextLine();

    System.out.print(line1 + " / " + line2 + " / " + line3);
}
```

Das führt zu folgenden Ausgaben:

```
Line1,Info / Line2.Special / Line3:Additional
```

Beispiel: Eingabe von der Konsole verarbeiten

Man kann den Scanner nicht nur für Strings nutzen, sondern auch für andere Eingabekanäle. Möchte man beispielsweise Eingaben von der Konsole lesen, so kann man dies in Kombination mit `System.in` wie folgt erreichen, wobei die Protokollierung auf der Konsole diverse Wertebelegungen des `Scanner`s zeigt:

```
jshell> Scanner scanner = new Scanner(System.in)
scanner ==> java.util.Scanner[delimiters=\p{javaWhitespace}+] ... \E][infinity
      string=\Q?\E]
```

Nun lassen sich die Methoden des `Scanner`s aufrufen und erwarten jeweils eine Eingabe. Im letzten Fall soll ein `int` gelesen werden. Da aber ein Gleitkommawert vorliegt, kommt es zu einer Exception. Zudem sollte der Scanner am Ende explizit durch Aufruf von `close()` geschlossen werden, da er hier nicht mit try-with-resources (vgl. Abschnitt 11.3) automatisch verwaltet wird:

```
jshell> scanner.next()
Test
$86 ==> "Test"

jshell> scanner.nextInt()
47
$87 ==> 47

jshell> scanner.nextLine()
MICHAEL
$88 ==> "MICHAEL"

jshell> scanner.nextDouble()
47.11
$91 ==> 47.11

jshell> scanner.nextInt()
47.11
|  Exception java.util.InputMismatchException

jshell> scanner.close()
```

3.2.2 Mehrzeilige Strings (Text Blocks)

Seit JDK 15 unterstützt Java auch die Angabe von mehrzeiligen Strings (auch Text Blocks genannt). Zuvor war es mühselig und aufwendig, da sämtliche Zeilentrenner sowie gegebenenfalls Anführungszeichen zu escapen waren:

```
jshell> String multiLineOldStyle = "This is line 1\n" +
   ...>                            "Second line with \"quotes\"\n" +
   ...>                            "Last line with 'single quotes'\n"
multiLineOldStyle ==> "This is line 1\nSecond line with \"quotes\"\nLast line
      with 'single quotes'\n"
```

Mehrzeilige Strings erleichtern dies und werden durch drei Anführungszeichen eingeleitet und abgeschlossen:

```
jshell> String multiLineString = """
   ...> This is line 1
   ...> Second line with "quotes"
   ...> Last line with 'single quotes'
   ...> """
multiLineString ==> "This is line 1\nSecond line with \"quotes\"\nLast line with
       'single quotes'\n"
```

Lassen wir uns den Text mal in der JShell ausgeben:

```
jshell> System.out.println(multiLineString)
This is line 1
Second line with "quotes"
Last line with 'single quotes'
```

Nach der mehrzeiligen Definition kann man wie zuvor gezeigt auf dem String agieren, also dessen Länge bestimmen, Bausteine ersetzen usw. Das liegt daran, dass ein solcher mehrzeiliger String ebenfalls ein normaler String ist und damit alle zuvor beschriebenen Aktionen auch unterstützt werden.

```
jshell> String multiLine2 = """
   ...> Zeile 1
   ...> Zeile 2 BLA BLA BLA
   ...> Zeile 3 und Ende"""
multiLine2 ==> "Zeile 1\nZeile 2 BLA BLA BLA\nZeile 3 und Ende"

jshell> multiLine2.length()
$190 ==> 44

jshell> multiLine2.replace("BLA", "WICHTIG")
$191 ==> "Zeile 1\nZeile 2 WICHTIG WICHTIG WICHTIG\nZeile 3 und Ende"
```

Besonderheit Platzhalter Außerdem kann man Platzhalter definieren und durch Aufruf von `formatted()` mit Werten befüllen – diese Methode entspricht in ihrer Handhabung exakt der statischen Methode `String.format()`:

```
String placeholders = """
       Michael %s hat am "%tF"
       %d Bücher in '%s' gekauft.
       """.formatted("Inden", LocalDate.of(2020, 1, 20), 7, "Bremen");
```

Durch diese Anweisungen wird folgende Ausgabe produziert:

```
Michael Inden hat am "2020-01-20"
7 Bücher in 'Bremen' gekauft.
```

Beispiel: HTML-Code

Auch wenn man HTML-Code in Java aufbereiten möchte, sind die mehrzeiligen Strings sehr hilfreich. Bisher musste man umständlich Folgendes schreiben:

```
var helloWorldHtmlOld = "<html>\n" +
                        "    <body>\n" +
                        "           <p>Hello World</p>\n" +
                        "    </body>\n" +
                        "</html>\n";
```

In der Tat ist es viel angenehmer, dies wie folgt anzugeben:

```
var helloWorldHtml = """
                     <html>
                         <body>
                             <p>Hello World</p>
                         </body>
                     </html>
                     """;
```

Eine Sache noch: Die Leerzeichen vor den Zeilen werden bis zu der Position der unteren drei Anführungszeichen gelöscht.

3.2.3 Strings und `char[]`s

Umwandlung in ein `char[]`

Mitunter ist es praktisch, den Inhalt eines Strings mit der Methode `toCharArray()` in eine geordnete Menge (ein Array) von Einzelzeichen, also vom Typ `char`, zu überführen. Arrays werden wir dann in Kapitel 4 genauer besprechen, hier reicht das Verständnis, dass es sich dabei um eine über die Position geordnete Abfolge von Elementen, hier Zeichen, handelt.

Schauen wir uns zunächst ein einfaches Beispiel an, das nach jedem Zeichen einen Unterstrich in der Ausgabe ergänzt:

```
jshell> String title = "Der Weg zum Java-Profi"
title ==> "Der Weg zum Java-Profi"

jshell> for (char ch : title.toCharArray())
   ...>     System.out.print(ch + "_");
D_e_r_ _W_e_g_ _z_u_m_ _J_a_v_a_-_P_r_o_f_i_
```

Wir lernen in diesem Beispiel mit `print()` eine Abwandlung der Methode `println()` kennen. Hierbei werden die Zeichen einfach ohne Zeilenumbruch hintereinander ausgegeben.

Zuvor erwähnte ich, dass es praktisch ist, einen String in ein `char[]` zu wandeln. Warum? Das gilt immer dann, wenn man an gewissen Stellen den String ändern möchte. Strings erlauben das ja nicht. Wenn wir nun aber beispielsweise jedes dritte Zeichen großschreiben wollen (oder für Sie als kleine Fingerübung durch ein Leerzeichen ersetzen), dann ist dies auf Basis eines `char[]` möglich. Die gewünschten Modifikationen müssen dazu auf einzelnen Zeichen erfolgen. Danach soll dann das `char[]` im Allgemeinen wieder in einen String gewandelt werden. Das lernen wir im Anschluss kennen.

Schauen wir uns vorab noch die Modifikation innerhalb eines `char[]` für jedes dritte Zeichen an. Zunächst definieren wir die Ausgangsdaten:

```
jshell> String alphabet = "abcdefghijklmnopqrstuvwxyz"
alphabet ==> "abcdefghijklmnopqrstuvwxyz"
```

Nun erstellen wir eine passende Methode und rufen diese einmal auf:

```
jshell> void modifyEvery3rdToUpper(char[] values)
   ...> {
   ...>     for (int i = 0; i < values.length; i+=3)
   ...>     {
   ...>         char current = values[i];
   ...>         values[i] = Character.toUpperCase(current);
   ...>     }
   ...> }
|  created method modifyEvery3rdToUpper(char[])

jshell> char[] asCharArray = alphabet.toCharArray()
asCharArray ==> char[26] { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' ...  'v', 'w',
       'x', 'y', 'z' }

jshell> modifyEvery3rdToUpper(asCharArray)

jshell> Arrays.toString(asCharArray)
$13 ==> "[A, b, c, D, e, f, G, h, i, J, k, l, M, n, o, P, q, r, S, t, u, V, w, x
      , Y, z]"
```

Einen String aus einem `char[]` erzeugen

Gerade haben wir gesehen, wie man aus einem String ein korrespondierendes `char[]` erhält. Manchmal möchte man auch aus einem `char[]` (wieder) einen String erhalten. Dazu dient ein Aufruf des Konstruktors `String(char[])`.

Als Beispiel sehen wir einen Gruß in Form eines `char[]`, der dann in einen `String` gewandelt wird:

```
jshell> char[] message = { 'H', 'o', 'i', ' ', 'S', 'o', 'p', 'h', 'i', 'e'}
message ==> char[10] { 'H', 'o', 'i', ' ', 'S', 'o', 'p', 'h', 'i', 'e' }

jshell> String fromCharArray = new String(message)
fromCharArray ==> "Hoi Sophie"
```

3.3 Praxisbeispiel: Text in Title Case wandeln

In diesem Praxisbeispiel wollen wir eine Funktionalität entwickeln, die einen Text in das spezielle Format einer Titelzeile umwandelt, wo jedes Wort großgeschrieben wird, etwa folgendermaßen:

```
Dies ist ein Titel für diese Übung   =>  Dies Ist Ein Titel Für Diese Übung
```

Modifikation ermöglichen

Strings sind ja bekanntermaßen unveränderlich. Um aber einen bestehenden Text wie gewünscht umwandeln zu können, müssen wir etwas tricksen. Zuvor hatten wir erfahren, wie ein Text mit `toCharArray()` in ein `char[]` gewandelt werden kann. Dort lassen sich Änderungen durchführen. Schließlich haben wir auch die Konvertierung aus einem `char[]` in einen String kennengelernt.

Funktionalität `toTitleCase()`

Zunächst wandeln wir den String in ein `char[]`. Darauf können wir dann die Modifikationen vornehmen. Dazu durchlaufen wir das Array mit einer `for`-Schleife von vorne nach hinten auf der Suche nach einem neuen Wortanfang. Als Indikator nutzen wir ein boolesches Flag `capitalizeNextChar`. Dies zeigt an, dass der erste Buchstabe des nächsten Worts großgeschrieben werden soll. Initial ist dieses Flag `true`, sodass das aktuelle (erste) Zeichen in einen Großbuchstaben umgewandelt wird. Nach der Umwandlung wird das Flag zurückgesetzt und es werden so lange Buchstaben übersprungen, bis man auf ein Leerzeichen oder Minuszeichen trifft. Dann setzt man das Flag wieder auf `true`. Das Prozedere wiederholt sich, bis man am Ende des Arrays angekommen ist. Aus dem an einigen Positionen modifizierten Array wird dann schließlich ein neuer String erzeugt:

```
String toTitleCase(String input)
{
    char[] inputChars = input.toCharArray();

    boolean capitalizeNextChar = true;
    for (int i = 0; i < inputChars.length; i++)
    {
        char currentChar = inputChars[i];

        if (capitalizeNextChar)
        {
            inputChars[i] = Character.toUpperCase(currentChar);
            capitalizeNextChar = false;
        }
        if (Character.isWhitespace(currentChar) || currentChar == '-')
        {
            capitalizeNextChar = true;
        }
    }
    return new String(inputChars);
}
```

Probieren wir das Ganze einmal in der JShell aus:

```
jshell> toTitleCase("alles in ordnung")
$137 ==> "Alles In Ordnung"

jshell> toTitleCase("der weg zum java-profi")
$138 ==> "Der Weg Zum Java-Profi"

jshell> toTitleCase("Dies ist ein Titel für diese Übung")
$139 ==> "Dies Ist Ein Titel Für Diese Übung"
```

Das war doch ein guter Start. Tatsächlich funktioniert das Ganze selbst dann, wenn der String nicht mit einem Buchstaben, sondern mit Leerzeichen startet.

Wenn Sie etwas experimentieren möchten, verwenden Sie weitere Trennzeichen.

Elegantere Umsetzung Je mehr Spezialbehandlungen in einem Programm erfolgen, desto leichter geschieht auch einmal ein Flüchtigkeitsfehler. In unserer Methode prüfen wir, ob das nächste Zeichen großgeschrieben werden soll. Danach prüfen wir, ob wir ein Trennzeichen gefunden haben, um dann wieder das Großschreiben zu aktivieren. Man kann das Setzen des Flags in eine Methode verlagern:

```
boolean shouldCapitalize(char currentChar)
{
    return Character.isWhitespace(currentChar) || currentChar == '-';
}
```

Diese Methode liefert immer `true`, wenn ein Leerraum oder Minuszeichen erkannt wird. Im Falle eines normalen Zeichens wird die Bedingung zu `false` ausgewertet.

Schauen wir uns die Auswirkungen im Kontext der ursprünglichen Methode an, dann erkennen wir, dass wir das Flag `capitalizeNextChar` einfacher besetzen und auf das Rücksetzen auf `false` verzichten können:

```
String toTitleCaseV2(String input)
{
    char[] inputChars = input.toCharArray();

    boolean capitalizeNextChar = true;
    for (int i = 0; i < inputChars.length; i++)
    {
        char currentChar = inputChars[i];

        if (capitalizeNextChar)
        {
            inputChars[i] = Character.toUpperCase(currentChar);
        }

        capitalizeNextChar = shouldCapitalize(currentChar);
    }
    return new String(inputChars);
}
```

Probieren wir das Ganze einmal in der JShell aus:

```
jshell> toTitleCaseV2("das ist die kürzere variante")
$142 ==> "Das Ist Die Kürzere Variante"
```

3.4 Aufgaben und Lösungen

3.4.1 Aufgabe 1: Länge, Zeichen und Enthaltensein

In dieser ersten Aufgabe geht es darum, grundlegende Methoden der Klasse `String` anzuwenden. Sie sollen die Länge eines Texts abfragen, dann ein Zeichen an einer beliebigen Position, etwa der 13., ermitteln und schließlich prüfen, ob ein gewünschtes Wort im String enthalten ist.

Lösung

Die geforderten Aktionen lassen sich direkt mit den passenden Methoden der Klasse `String` umsetzen, nämlich mit `length()`, `charAt()` und `contains()`:

```
jshell> String nachricht = "Hallo lieber Leser! Viel Spaß mit Java!"
nachricht ==> "Hallo lieber Leser! Viel Spaß mit Java!"

jshell> nachricht.length()
$60 ==> 39

jshell> nachricht.charAt(13)
$61 ==> 'L'

jshell> nachricht.contains("Java")
$63 ==> true
```

3.4.2 Aufgabe 2: Title Case mit `Scanner`

Als Praxisbeispiel haben wir gesehen, wie wir einen beliebigen Text in einen Text im Stil einer Überschrift mit jeweils großgeschriebenen ersten Buchstaben konvertieren können. Die präsentierte Lösung konnte nicht nur Leerzeichen, sondern auch mit '–' als Worttrenner umgehen.

In dieser Aufgabe soll eine vereinfachte Variante erstellt werden, die lediglich Leerzeichen als Worttrenner erlaubt. Das Ganze ist mithilfe der Klasse `Scanner` basierend auf einem Eingabestring zu implementieren. Schreiben Sie eine Methode `String toTitleCase(String)`.

Lösung

Als Erstes erstellen wir einen `Scanner` basierend auf dem Eingabestring. Danach durchlaufen wird diesen, solange es noch weitere Wörter gibt. Das erste Zeichen konvertieren wir in Upper Case und den restlichen Text erhält man per `substring(1)`. Nun hängen wir noch ein Leerzeichen an. Um ein zum Ende überschüssiges Leerzeichen zu entfernen, rufen wir abschließend `trim()` auf: