Technische Universität Dortmund
Fachschaft Informatik
Simon Dierl
Sebastian Lukas Hauer
Philip Molares

Handbuch zum

Caffeine & Code

{☕}

Wintersemester 2017/2018

Willkommen

Herzlich willkommen zum diessemestrigen *Caffeine & Code*-Event der Fachschaft Informatik. Wir freuen uns, dass Du zu uns gefunden hast und hoffen, dass Du beim Bearbeiten dieser Aufgaben genau so viel Spaß haben wirst wie wir bei der Erstellung. Auf den folgenden Seiten findest Du Aufgaben, sowie zusätzlich ein paar Texte, die Dich auf die Aufgaben vorbereiten sollen. Wir bitten Dich, diese sorgfältig und der Reihe nach durchzulesen, damit alle Teilnehmer auf dem gleichen Wissensstand sind und unsere Betreuer nicht mehrfach dasselbe erklären müssen.

Zum Bearbeiten der Programmieraufgaben setzen wir keine besonderen Programme voraus, jedoch halten wir die Verwendung von **Eclipse** oder **IntelliJ** als IDE für eine vernünftige Wahl. Eine Anleitung, wie Du aus einzelnen Java-Klassen eine ausführbare `.jar`-Datei erstellst, findest Du ebenfalls hier im Handbuch.

Die Aufgaben in diesem Handbuch sind vollständig mit den gegebenen Hinweisen und den bisher vermittelten Kenntnissen von DAP1 lösbar. Diese werden hier mithilfe eines kleinen, erweiterbaren Projekts in die Praxis umgesetzt. Solltest Du an der Programmieraufgabe selbst kein Interesse haben und Dich lieber mit der Wiederholung von Vorlesungsinhalten beschäftigen, haben wir im hinteren Bereich des Handbuchs ein paar Hinweise und Antworten auf Fragen, die uns letztes Jahr beim *Caffeine & Code* besonders aufgefallen sind.

Reverse Polish Notation

Umgekehrte polnische Notation, auch Postfixnotation genannt, ist eine Notation für mathematische Terme. Anders als in der üblichen, auch als Infixnotation bekannten Schreibweise, werden Operationen nicht in der Form

`Operand-A Operator Operand-B` $(3+4)$

notiert, sondern als

`Operand-A Operand-B Operator` $(34+)$

aufgeschrieben.

Dies hat bei der Auswertung von Termen einen großen Vorteil: Die Reihenfolge, in der Operatoren angewendet werden, ist eindeutig. Klammern sind daher überflüssig. Ein Infixterm $(2+5)\times 7$ kann z. B. als $25+7\times$ oder $725+\times$ notiert werden.

Hier zeigt sich, wie Postfixterme „ausgerechnet“ werden: Ein Operator wird immer auf die direkt vor ihm stehenden Operanden angewendet. Beim Auswerten wird also in jedem Schritt der „innerste“ Teilterm durch sein Ergebnis ersetzt:

$$725+\times = 77\times = 77\times = 49$$

Ein weiterer Vorteil ist die Möglichkeit, dass Postfixterme sehr leicht maschinell ausgewertet werden können. Die einzige Datenstruktur, die wir dafür benötigen, ist ein **Stapel** (*eng.* **Stack**). Wir lesen den Term von links nach rechts. Lesen wir eine Zahl, legen wir sie auf den Stapel. Lesen wir einen Operator, so entfernt dieser die obersten zwei Zahlen vom Stapel, verarbeitet sie und legt das Ergebnis wieder oben auf dem Stapel ab.

Eine Anweisungsfolge für den oben stehenden Term sähe z. B. wie folgt aus:

Term	Anweisung	Stapel
$725+\times$	push 7	$[\,]$
$725+\times$	push 2	$[7]$
$725+\times$	push 5	$[7,2]$
$725+\times$	$b \leftarrow$ pop	$[7,2,5]$
$725+\times$	$a \leftarrow$ pop	$[7,2]$
$725+\times$	push $a+b$	$[7]$
$725+\times$	$b \leftarrow$ pop	$[7,7]$
$725+\times$	$a \leftarrow$ pop	$[7]$
$725+\times$	push $a\times b$	$[\,]$
$725+\times$	done	$[49]$

Das Ergebnis der Auswertung eines Terms liegt anschließend immer oben auf dem Stapel. Auch falsch notierte Terme lassen sich so beim Auswerten leicht erkennen: Es sind nicht genügend Operanden auf dem Stapel, um eine gegebene Operation auszuführen oder es liegt am Ende der Auswertung mehr als eine Zahl auf dem Stapel.

Um sich mit dem Konzept auseinanderzusetzen, findest Du auf dieser Seite ein paar Aufgaben zum Thema **Reverse Polish Notation**.

Aufgabe 1: Umformung in Reverse Polish Notation

Schreibe folgende **Infixterme** in **Postfixterme** um:

Infixterm	**Postfixterm**
$2 + 4 \times 3$	$2\,4\,3\,\times\,+$
$(2 + 4) \times 3$	
$1 + 2 + 3 + 4$	
$1 \times 2 + 3 \div 4 - 5 + 6$	
$\sin(4 \times \pi)$	
sin(cos(sqrt(100)))	

Aufgabe 2: Auswertung von Reverse Polish Notation

Werte folgende **Postfixterme** aus:

Postfixterm	**Wert**
$2\,4\,3\,\times\,+$	14
$2\,1\,-\,3\,+$	
$4\,\mathrm{sqrt}\,2\,\times\,2\,+$	
$10\,20\,5\,5\,+\,\times\,+$	
$10\,20\,+\,5\,5\,+\,\times$	
$\pi\,2\,\div\,\sin\,\pi\,\times\,\sin\,\sin\,\pi\,\times\,\sin\,\sin$	

Java von Hand kompilieren

Ein Softwareprojekt, hier beispielhaft „meinprojekt“ genannt, wird für gewöhnlich so (oder so ähnlich) aufgebaut:

```
meinprojekt                         # Wurzelverzeichnis des Projektes
├── bin                             # Kompilierte Quelldateien
│   ├── data
│   │   └── 1.txt                  # Eine Kopie von res/data/1.txt
│   └── de
│       └── tudo
│           └── meinprojekt
│               ├── Main.class     # Die kompilierte Main.java
│               ├── Student.class  # Die kompilierte Student.java
│               └── Util.class     # Die kompilierte Util.java
├── res                             # Daten, die das Programm zum Arbeiten braucht
│   └── data
│       └── 1.txt                  # Eine Datei, die nicht kompiliert wird
└── src                             # Quelldateien
    └── de
        └── tudo
            └── meinprojekt         # Entspricht dem Package de.tudo.meinprojekt
                ├── Main.java       # Eine Quelldatei für die Klasse Main
                ├── Student.java    # Eine Quelldatei für die Klasse Student
                └── Util.java       # Eine Quelldatei für die Klasse Util
```

Das Projektverzeichnis sollte einen Namen haben, mit dem Du es leicht wiederfinden kannst. Der Projektname bietet sich natürlicherweise an. In jenem Verzeichnis gliedern wir unsere Daten in Unterverzeichnisse. Die hier benutzte Benennung ist natürlich nur ein Beispiel, das sich allerdings an gängigen Praktiken orientiert.

Um ein Java-Projekt zu organisieren, wird zu Beginn einer jeden Quelldatei das `package`-Statement (Paketdeklaration) verwendet. Damit wird der relative Pfad der Quelldatei zum Projektursprung angegeben. In der Regel wird ein Pfad verwendet, der einer umgedrehten URL entspricht:

```
package de.tudo.meinprojekt.util;
```

Wenn Du diese Paketdeklaration verwendest, muss die entsprechende Quelldatei im Verzeichnis `de/tudo/meinprojekt/util/` in Deinem Projekt abgelegt sein. Diese langen Pfade haben sich durchgesetzt, um die Chance, dass jemand anderes dasselbe Paket benutzt, zu reduzieren – zwei gleichnamige Klassen im selben Paket können nämlich nicht kombiniert werden!

Das Verzeichnis `src` (kurz für `source`, Quellcode) enthält Java-Quellcode, der kompiliert werden muss. Die Verzeichnisstruktur muss den Paketen entsprechen, die in den Quelldateien deklariert werden.

Oft brauchen Programme jedoch weitere Dateien, die mit ausgeliefert werden müssen. Beispiele sind Grafiken, Übersetzungen der Benutzerschnittstelle oder Voreinstellungen des Programms. Diese Dateien werden im Verzeichnis `res` (kurz für `resources`, Ressourcen) abgelegt. Sie müssen nicht kompiliert oder verarbeitet werden.

Das Verzeichnis `bin` (kurz für `binaries`, Binärdateien) ist der Aufbewahrungsort für das „fertige“ Programm. Hier werden die kompilierten `.class`-Dateien abgelegt. Die Ressourcen aus `res` werden einfach hierhin kopiert. Die namensgebenden Binärdateien sind die kompilierten `.class`-Dateien, die nicht mehr als Text lesbar sind, sondern die nur binär betrachtet werden können.

Das Übersetzen von `.java`- in `.class`-Dateien hat bis jetzt Deine IDE erledigt. Allerdings gibt es Situationen, in denen Du einfach keine IDE zur Hand hast: z. B. auf einem Server, einem leistungsschwachen Rechner oder wenn Du das Kompilieren mit einem Skript automatisieren willst. Der eigentliche Java-Compiler ist das Werkzeug `javac`, das von der Kommandozeile aus aufgerufen wird.

Aufgabe 3: Der Java-Compiler

Lies Dich in die Dokumentation des Java-Compilers ein! Dies kannst Du entweder online auf den Webseiten von Oracle tun, oder indem Du auf einem Linux-System per `man javac` die Handbuchseite öffnest.

Nun solltest Du die folgenden Fragen selbstständig beantworten können:

1. Angenommen, wir führen den folgenden Befehl aus:

   ```
   javac -d dir de/example/Example.java org/library/Util.java
                 com/acme/ExpensiveTool.java
   ```

   Welche der folgenden Aussagen sind korrekt?

   - ☐ Der Quellcode befindet sich im Verzeichnis `dir`.
   - ☐ Es werden drei Dateien kompiliert.
   - ☐ Der gesamte Quellcode wird in die Datei `ExpensiveTool.class` kompiliert.
   - ☐ Es werden mindestens drei `.class`-Dateien erzeugt.
   - ☐ Ist das System auf italienische Sprache eingestellt, müssen wir `it/example/Example.java` schreiben.
   - ☐ Die kompilierten `.class`-Dateien werden nicht auf alten Java-Versionen laufen.
   - ☐ Enthält der Quellcode Syntaxfehler, wird dies auf der Kommandozeile angezeigt.
   - ☐ Die Quelldateien müssen in der Kodierung `ISO-8859-1` vorliegen.

2. Wozu dient der Parameter `-g` des Java-Compilers? In welchen Situationen ist welche Einstellung sinnvoll?
3. Über die Optionen `-source` und `-target` kann man Kompatibilität mit alten Java-Versionen herstellen. Was genau machen die Optionen und wodrin unterscheiden sie sich?
4. Welchen Befehl müsste man benutzten, um in dem obigen Beispielprojekt alle Java-Dateien zu kompilieren?

Java-Projekte und `.jar`-Dateien

In der Veranstaltung DAP1 hast Du bisher vermutlich alle Java-Dateien nur innerhalb von BlueJ geschrieben, übersetzt und getestet. In der Praxis möchte man für ein fertiges Softwareprodukt natürlich eine ausführbare Datei ausliefern können.

Für Java sind dies `.jar`-Dateien: Komprimierte (ZIP-)Archive, in denen alle `.class`-Dateien, Ressourcen und Metainformationen über das Programm zu finden sind.

Damit Java in der Lage ist, Deine `.class`-Dateien wiederzufinden, müssen nicht nur die Quelldateien richtig „einsortiert“ werden; die Struktur des `.jar`-Archivs muss ebenfalls die Paketstruktur widerspiegeln.

Alle gängigen IDEs bieten an, das kompilierte Java-Projekt mit einigen Mausklicks als `.jar`-Datei zu exportieren. Genau wie beim Kompilieren ist es auch hier hilfreich, zu verstehen, was „unter der Haube“ geschieht. Für das manuelle Erstellen von `.jar`s wird das Programm `jar` verwendet:

```
jar cf JAR-DATEINAME KLASSENDATEIEN
```

Was einer vollständigen `.jar`-Datei noch fehlt, ist ein sogenanntes **Manifest**. Dies ist eine Datei mit dem Namen `MANIFEST.MF`, in der Informationen zum Projekt stehen – insbesondere der Pfad zur Hauptklasse mit der `main`-Methode. Eine minimale Manifest-Datei sieht daher folgendermaßen aus:

```
1 Manifest-Version: 1.0
2 Created-By: 1.7.0_06 (Oracle Corporation)
3 Main-Class: de.tudo.meinprojekt.Hauptklasse
```

Diese Datei kann dann dem `.jar`-Archiv manuell hinzugefügt werden. Sie muss dabei im Unterverzeichnis `META-INF` zu finden sein. Eine `.jar`-Datei sollte also folgende interne Struktur besitzen:

```
1 META-INF/MANIFEST.MF
2 de/tudo/meinprojekt/Hauptklasse.class
3 de/tudo/meinprojekt/util/Foo.class
4 de/tudo/meinprojekt/data/Bar.class
```

Wem das Einbinden des Manifestes zu viel Aufwand ist, der kann die Datei auch beim Erstellen des Archivs automatisch generieren lassen, indem das Werkzeug folgendermaßen benutzt wird:

```
jar cfe JAR-DATEINAME HAUPTKLASSENNAME KLASSENDATEIEN
```

In der Praxis könnte dies so aussehen:

```
jar cfe myapp.jar de.tudo.meinprojekt.Hauptklasse de/*.class
```

Aufgabe 4: Das Jar-Werkzeug

Lies Dich in die Dokumentation des `jar`-Werkzeugs ein! Wie beim Java-Compiler kannst Du dies wieder auf den Webseiten von Oracle tun oder auf einem Linux-System `man jar` ausführen.

Nun solltest Du die folgenden Fragen selbstständig beantworten können:

1. Angenommen, wir führen den Befehl

   ```
   jar -cfe ../example.jar de.example.Example de/example/Example.class
           org/library/Util.class com/acme/ExpensiveTool.class
   ```

   aus. Welche der folgenden Aussagen sind korrekt?

   - ☐ Wir könnten statt `de.example.Example` auch `de/example/Example.class` schreiben.
   - ☐ Wenn wir versuchen, eine `.java`-Datei hinzuzufügen, wird diese automatisch kompiliert.
   - ☐ Wir können statt `-cfe` auch `fec` schreiben.
   - ☐ `jar -xf ../example.jar` löscht die Datei `../example.jar` und extrahiert die Dateien darin ins aktuelle Verzeichnis.
   - ☐ Wird die Option `-0` verwendet, wird das Archiv größer.
   - ☐ Lassen wir die Option `-f ../example.jar` weg, wird ein Dateiname automatisch gewählt und auf der Konsole ausgegeben.

2. `.jar`-Archive sind „nur“ normale ZIP-Archive. Welche Vor- und Nachteile hat dies?
3. Welchen Befehl müsste man benutzten, um in dem Beispielprojekt zu Aufgabe 3 alle Dateien in ein `.jar` zu packen?

Dateiverarbeitung

Wie viele andere Programmiersprachen auch, ist Java im stetigen Wandel. Seit der Version 7 von Java gibt es in der Standardbibliothek neue Möglichkeiten, mit Dateisystemen zu arbeiten. Diese sind im `java.nio.files`-Paket organisiert. Die Klassen in diesem Paket stellen eine große Menge an statischen Methoden zur Verfügung, die Pfad-Objekte zurückgeben. Pfad-Objekte abstrahieren Pfade im Dateisystem. Weitere Klassen in diesem Paket können mithilfe der Pfad-Objekte auf dort zu findende Dateien zugreifen und diese bearbeiten.

Als **Pfad** bezeichnen wir einen String wie `/home/simon/Documents/CnC.pdf` (Linux) oder `C:\Users\Simon\Documents\CnC.pdf` (Windows). Unter einem solchen Pfad kann sich eine Datei oder ein Verzeichnis befinden – dies muss aber nicht der Fall sein! In Java können wir mit der statischen Methode `Paths.get(String pathname)` ein `Path`-Objekt erzeugen, das einen solchen Pfad kapselt. `Path`-Objekte haben natürlich Komfortfunktionen, wie beispielsweise die Erzeugung des Elternpfades.

Um an den Inhalt einer Datei zu gelangen, die unter einem Pfad abgelegt ist, verwenden wir die statischen Methoden der **Files**-Klasse (`java.nio.file.Files`). Ob ein gegebener Pfad existiert, kann z. B. mit der `exists(Path path)`-Methode überprüft werden.

Aus Dateien lesen

Die Methode `Files.newBufferedReader(Path path)` erstellt ein neues **BufferedReader**-Objekt, welches aus der durch den Pfad beschriebenen Datei liest. Dabei wird die Datei implizit „geöffnet“. Scheitert das Öffnen, werden wir darüber durch eine `IOException` benachrichtigt.

Um Zeilen aus der Datei zu extrahieren, existiert die Methode `readLine()`. Diese liest eine Zeile aus der mit vom **BufferedReader** geöffneten Datei aus und gibt sie als String zurück. Hat man alle Zeilen der Datei ausgelesen, gibt die Methode stattdessen `null` zurück.

Nachdem die Arbeit mit einer Datei beendet ist, muss sie auch wieder „geschlossen“ werden. Dies kann durch Aufrufen der Methode `close()` des **BufferedReader**s erledigt werden. Da jedoch das Schließen der Datei wiederum eine Ausnahme auslösen kann, erzeugt das Behandeln dieser redundanten und unnützen Code.

Seit Java 7 können `try`-Blöcke mit der Deklaration einer Ressource versehen werden. In unserem Fall ist dies die Konstruktion eines **BufferedReader**s. Diese Ressource muss die Schnittstelle `AutoCloseable` implementieren. Sie wird beim Verlassen des `try`-Blocks wieder geschlossen. Dieses Konstrukt nennt sich **try-with-resources** und kann auch für viele weitere Probleme genutzt werden.

Im folgenden werden zwei Möglichkeiten präsentiert, eine Datei einzulesen und auszugeben.

Hier zunächst ohne **try-with-resources**:

```
1  public static void printFile(String pathname) {
2      Path path = Paths.get(pathname);
3      BufferedReader reader = null;
4      try {
5          reader = Files.newBufferedReader(path);
6          String line = reader.readLine();
7          while(line != null) {
8              System.out.println(line);
9              line = reader.readLine();
10         }
11     } catch (FileNotFoundException e) {
12         System.err.println("File not found!");
13     } catch (IOException e) {
14         System.err.println("I/O Error occurred!"); //other i/o errors
15     } finally {
16         if(reader != null) {
17             try {
18                 reader.close();
19             } catch (IOException e) {
20                 System.err.println(e.getMessage());
21                 // we can't recover the reader from this
22             }
23         }
24     }
25 }
```

Zum Vergleich der gleiche Code mit dem neuen **try-with-resources**-Konstrukt:

```
1  public static void printFile(String pathname) {
2      Path path = Paths.get(pathname);
3      try (BufferedReader reader = Files.newBufferedReader(path)) {
4          while(true) {
5              String line = reader.readLine();
6              if(line == null) {
7                  break; //using if(...){break;} prevents redundant code
8              }
9              System.out.println(line);
10         }
11     } catch (FileNotFoundException e) {
12         System.err.println("File not found!");
13     } catch (IOException e) {
14         System.err.println("I/O Error occurred!"); //other i/o errors
15     }
16 }
```

In Dateien schreiben

Die Methode `Files.newBufferedWriter(Path path)` erstellt ein neues **BufferedWriter**-Objekt, welches in die durch den Pfad beschriebene Datei schreiben kann. Die `newBufferedWriter`-Methode nimmt zusätzlich eine beliebige Anzahl an Optionsargumenten entgegen. Diese Argumente sind vom Typ OpenOption. Eine Standardsammlung an gültigen Optionen findet man in der Klasse `java.nio.files.StandardOpenOption`. Wenn man z. B. nur Daten an eine Datei anfügen möchte, anstatt ihren gesamten Inhalt zu überschreiben, benötigt man die Option `APPEND`. `CREATE` würde die Datei zusätzlich erstellen, sollte sie nicht bereits existieren.

Die Methode `write(String output)` des **BufferedWriter**s schreibt einen String in die verknüpfte Datei. Die `write`-Methode kann auch mit primitiven Datentypen aufgerufen werden.

Folgender Code kann z. B. den Inhalt einer Datei einlesen und schreibt jedes in ihr enthaltene Wort in eine eigene Zeile in eine Datei mit dem selben Namen und der zusätzlichen Endung „.token“.

```
1  public static void cutMyFileIntoPieces(String pathname) {
2      Path inPath = Paths.get(pathname);
3      Path outPath = Paths.get(pathname+".token");
4      try(BufferedWriter writer =
5          Files.newBufferedWriter(outPath,
6              StandardOpenOptions.APPEND,
7              StandardOpenOptions.CREATE)) {
8          try(BufferedReader reader = Files.newBufferedReader(inPath)) {
9              String line;
10             while((line = reader.readLine()) != null) {
11                 StringTokenizer tokenizer = new StringTokenizer(line, " ");
12                 while(tokenizer.hasMoreTokens()) {
13                     writer.write(tokenizer.nextToken()+"\n");
14                 }
15             }
16         } catch (IOException e) {
17             System.err.println("Error opening input file");
18         }
19     } catch (IOException e) {
20         System.err.println("Error opening output file");
21     }
22 }
```

Programmierprojekt: Mein eigener Taschenrechner

Ziel dieser Programmieraufgabe ist es, ein Programm zu schreiben, das vom Benutzer eingegebene Terme in umgekehrter polnischer Notation auswertet. Zusätzlich soll es möglich sein, dem Programm einen Dateipfad als Argument zu übergeben, um in einer Datei hinterlegte Terme auszuwerten.

Aufgabe 5: Generische Programmierung: Vorbereitung des Stapels

Erstelle eine neue Java-Klasse mit dem Namen **Stack** und generischem Typparameter `T`. Objekte dieser Klasse sollen einen Stack für Elemente beliebigen Typs repräsentieren.

Erstelle zum Ablegen der Elemente auf dem Stapel eine weitere, innere Klasse `Element` mit den Attributen `value` vom Typ `T` und `next` vom Typ **Element**.

Der Stapel selbst soll ein Attribut mit dem Namen `first` vom Typ `Element` besitzen.

Implementiere anschließend zwei Methoden `void push(T)` und `T pop()`, die ein Objekt vom Typ `T` auf den Stapel legen bzw. vom Stapel entfernen und zurückgeben sollen. Eine dritte Methode `T peek()` soll das oberste Element vom Stapel zurückgeben, ohne es zu entfernen.

Darüber hinaus müssen auch die Methoden `void clear()` und `boolean isEmpty()` implementiert werden. Erstere soll den Stapel leeren indem alle Elemente entfernt werden, die andere soll genau dann `true` zurückgeben, wenn der Stapel leer ist (und ansonsten `false`).

Aufgabe 6: Programmieren mit Strings: Tokenizer

Ein **Tokenizer** soll einen String, z. B. „Hallo Welt! Ich kann programmieren!“, in seine Einzelteile, sogenannte Token, zerlegen können. Im oberen Beispiel sind dies die Token „Hallo“, „Welt!“, „Ich“, „kann“ und „programmieren!“. Für unsere Zwecke soll z. B. der Term „3 2 5 × +“ in die Token 3, 2, 5, × und + zerlegt werden. Auf dieser so entstandenen Folge von Token kann der Rest unseres Programms arbeiten, ohne sich um Probleme wie Leerzeichen, Tabulatoren etc. kümmern zu müssen.

Erstelle eine neue Klasse mit dem Namen `Tokenizer`. Objekte dieser Klasse sollen einen String in seine Einzelteile aufzuspalten:

Der `Tokenizer` soll bei Instantiierung einen String übergeben bekommen, den er intern weiter verwaltet.

Die Methode `boolean hasMore()` soll `true` zurück geben, wenn der `Tokenizer` den String noch weiter zerlegen kann, ansonsten `false`.

Die Methode `String getNext()` soll den internen String nach dem nächsten Token durchsuchen, dieses zurückgeben und bei Nichtvorhandensein `null` zurückgeben. Verwende zum Suchen des nächsten Tokens am besten zwei Hilfsmethoden: `int skip(int)` und `int scan(int)`.

`skip` soll von der übergebenen Position aus alle Leerzeichen (`' '`, `'\n'`, `'\r'`, `'\f'`, `'\t'`) erkennen und die erste Position zurückgeben, an der kein Leerzeichen mehr steht. `scan` soll von der übergebenen Position aus alle normalen Symbole erkennen, bis es ein Leerzeichen erkennt und die erste Position eines solchen Zeichens zurückgeben. Die `getNext`-Methode kann mithilfe

von `skip` und `scan` die Anfangs- und Endposition des nächsten Tokens finden und soll dieses zurückgeben.

Hinweis: Schau Dir die Dokumentation von `String.substring(...)` an.

Aufgabe 7: Funktionale Programmierung: Vorbereitung der Operatoren

Erstelle zwei neue funktionale Java-Schnittstellen (**functional interface**) mit den Namen `Binary-Operator` und `UnaryOperator` mit generischem Typparameter `T`. Diese sollen binäre ($+, -, \times, \div$) bzw. unäre ($\sin, \cos, \sqrt{\cdot}$) Operatoren implementieren.

Binäre Operatoren erhalten eine Methode `T apply(T, T)`, die zwei Objekte vom Typ `T` entgegen nimmt und ein Objekt vom Typ `T` zurück gibt. Die unären Operatoren bekommen eine Methode `T apply(T)`, die ein Objekt vom Typ `T` entgegen nimmt und ein Objekt vom Typ `T` zurück gibt. Diese Operatoren sollen später unter Benutzung von Lambda-Ausdrücken bei der Implementierung unseres Taschenrechners verwendet werden.

Aufgabe 8: Ausnahmebehandlung: Vorbereitung

Während der Ausführung unseres Taschenrechnerprogramms können natürlich auch Dinge schief gehen. Zum Beispiel kann ein Nutzer unzureichend viele Zahlen auf dem Stapel ablegen, bevor er einen Operator ausführen lässt. Ebenso kann ein Benutzer des Taschenrechners einen Text eingeben, der für uns keinen Sinn ergibt.

Für den ersteren Fall soll eine **StackEmptyException** vorbereitet werden.

Für den zweiten Fall soll eine **IllegalInputException** vorbereitet werden.

Beide Fälle können – oder wollen – wir als Aufrufender nicht vorhersehen. Damit wir die Ausnahmen zwangsweise behandeln, sollten sie als **checked exception** angelegt und in einer `throws`-Klausel deklariert werden.

Aufgabe 9: Objektorientierte Programmierung: Der Taschenrechner – Die Anwendungslogik

Erstelle eine neue Klasse mit dem Namen `Calculator`. Objekte dieser Klasse repräsentieren einen Taschenrechner, der Terme in Postfixnotation auswerten soll.

Taschenrechner sollen einen **Stack** für **Double**-Objekte besitzen, den sie intern verwalten. Implementiere zwei Methoden, beide mit dem Signatur `void performOperation(...)`. Eine soll einen unären, die andere eine binären Operator entgegen nehmen. Die Methoden sollen die Argumente der Operatoren vom Stapel des Taschenrechners entnehmen und ihr Ergebnis wieder auf den Stapel legen. Sollten nicht genügend Elemente auf dem Stapel liegen, so soll die **StackEmptyException** geworfen werden.

Damit der Taschenrechner Eingaben „verstehen“ kann, wird eine Methode `void parse(String)` implementiert. Diese nimmt einen **String** entgegen und soll mithilfe einer großen `switch-case`-Anweisung zwischen den Symbolen `+, -, *, /` unterscheiden.

Die einzelnen Fälle sollen durch das Aufrufen der Methode `performOperation` mit einem passenden Lambda-Ausdruck behandelt werden. Der Standardfall soll erwarten, dass in der übergebenen Zeichenkette eine Zahl enthalten ist und diese mithilfe von `Double.parseDouble(String`

`text)` in einen Double umwandeln. Dieser soll dann auch auf den Stapel gelegt werden. Falls sich der Wert nicht parsen lässt, soll eine **IllegalInputException** geworfen werden.

Hinweis: Die **NumberFormatException** der `Double.parseDouble(String text)`-Methode ist unchecked. Diese muss hier also explizit abgefangen werden.

Füge im Anschluss der `parse`-Methode noch den Fall hinzu, dass das eingegebene Token der Begriff `exit` ist. In diesem Fall soll eine Exception vom Typ **ExitException** geworfen werden. Da wir diese Anforderung gerade eben noch nicht kannten, müssen wir diese Klasse noch anlegen.

Aufgabe 10: Objektorientierte Programmierung: Der Taschenrechner – Textanalyse

Damit die `parse`-Methode nur mit einzelnen Symbolen oder Zahlen aufgerufen wird, müssen wir eventuell eingegebene Folgen von diesen vorher auseinander nehmen. Implementiere eine Methode `Double lex(String)`, die einen String entgegen nimmt und ihn in mithilfe eines **Tokenizers** in seine Einzelteile zerlegt. Jedes extrahierte Token soll an die `parse`-Methode weitergegeben werden.

Nachdem `lex` den gesamten an ihn übergebenen Term ausgewertet hat, soll das oberste Element des Stapels betrachtet und als Ergebnis zurückgegeben werden.

Durch den Aufruf von `parse` können natürlich Ausnahmen geworfen werden. Diese wollen wir hier jedoch nicht behandeln. Deswegen wirft die `lex`-Methode diese einfach weiter.

Aufgabe 11: Softwareentwicklung: Gute Praktiken – Die Hauptklasse

Erstelle eine neue Klasse mit dem Namen `Application`. Diese Klasse wird unsere Hauptklasse und enthält die `void main(String[])`-Methode unserer Anwendung. Objekte der Klasse `Application` sollen einen Taschenrechner als Attribut besitzen und eine Methode namens `run` implementieren.

Zuerst wollen wir diese Klasse zum Testen unseres Taschenrechners verwenden. Rufe in `run` die `lex`-Methode des Taschenrechners mit einem Term in Postfixnotation auf und lasse das Ergebnis ausgeben. Die Ausnahmen **StackEmptyException**, **IllegalInputException** und **ExitException** sollen hier behandelt werden. Gib in allen Fällen hilfreiche Informationen mit `System.err.println(String)` aus bzw. verabschiede Dich höflich vom Benutzer.

Die `main`-Methode soll zunächst nichts weiter machen, als das Programm in Gang zu setzen. Erstelle also eine neue Instanz der Applikation und führe ihre `run`-Methode aus.

Aufgabe 12: Bauen und Testen

Du bist nun an dem Punkt angekommen, an dem das Programm ausführbar ist. Teste das Programm in Deiner IDE, exportiere eine ausführbare `.jar`-Datei oder kompiliere das Projekt und erstelle die `.jar`-Datei selbst.

Aufgabe 13: Dateien einlesen und auswerten

Der Taschenrechner soll nun die Fähigkeit erhalten, Dateien einzulesen und auszuwerten. Wird ein Argument an das Programm übergeben, soll erwartet werden, dass sich dahinter ein relativer Pfad zu einer Datei verbirgt.

Implementiere eine Methode `run(String filePath)`, die eine Datei unter dem gegebenen Pfad öffnet, sie zeilenweise ausliest und die gelesenen Zeilen an die `lex`-Methode des Taschenrechners weiterreicht. Wurden alle Zeilen gelesen, soll das Ergebnis der letzten Zeile ausgegeben werden.

Auch hier muss natürlich die Behandlung der Ausnahmen implementiert werden, kann in diesem Fall jedoch anders erfolgen, z. B. kann die Zeilennummer der fehlerhaften Zeile mit angegeben werden; ein `exit` in einer Datei ergibt keinen Sinn und kann anders behandelt werden. Hier kannst Du Einfallsreichtum beweisen.

Die `main`-Methode soll nun unterscheiden, ob ein Argument übergeben wurde oder nicht (`args.length == 0`) und im ersten Fall die `run`-Methode mit dem ersten übergebenen Argument aufrufen (`args[0]`).

Das resultierende Programm sollte mithilfe eines Aufrufs wie z. B.

```
java -jar calc.jar input.rpn
```

aufrufbar sein und ein Ergebnis ausgeben. Der Inhalt der Datei soll dabei ein Term in umgekehrter polnischer Notation sein.

Aufgabe 14: Benutzerinteraktion

Wird das Programm ohne Argument gestartet, soll es interaktiv ablaufen. Der Benutzer soll einen Term eingeben können, das Ergebnis ausgegeben bekommen und weitere Terme eingeben können.

Du kannst einen `BufferedReader` für die Standardeingabe `System.in` erstellen, indem Du zuvor einen `InputStreamReader` erstellst, an dessen Konstruktor du `System.in` übergibst und den `InputStreamReader` an einen Konstruktor des `BufferedReader`s übergibst.

Objekte, die aus einem Eingabe-Stream lesen, blockieren für gewöhnlich die Programmausführung, solange keine neuen Daten im Stream zu finden sind. Die Standardeingabe `System.in` ist ein solcher Stream und beinhaltet die Symbole, die ein Benutzer im Terminal, in dem das Programm läuft, eingegeben und abgesendet hat. Wichtig ist hier zwischen dem Zustand „Es sind keine Daten vorhanden.“ und „Der Stream ist zuende.“ zu unterscheiden. Letzteres wird üblicherweise durch besondere Steuersymbole repräsentiert (ASCII-Zeichen mit Werten zwischen 0 und 31), die beim Auslesen des Streams nicht zurückgegeben werden. Man kann diese z. B. durch die Tastenkombination STRG+D an das Programm senden.

Aufgabe 15: Funktionalitäten erweitern

Bisher unterstützt unser Taschenrechner nur die Operatoren +,-,$\times$ und $\div$. Erweitere die Fallunterscheidung der `parse`-Methode um das Erkennen der Token sin, cos und $\sqrt{\cdot}$ (sqrt). Ein Operator % für Modulo und Operatoren für das Bestimmen des Minimums (min) und Maximums (max) zweier Zahlen bieten sich auch an.

Aufgabe 16: Interner Speicher

Viele Taschenrechner haben auch die Möglichkeit, Zwischenergebnisse zu speichern und zu laden. Füge dem Taschenrechner ein intern verwaltetes `Double`-Array der Größe 16 hinzu. Füge der Fallunterscheidung der `parse`-Methode die Operatoren `save` und `load` hinzu. `save` soll zwei Argumente entgegen nehmen und das erste Argument am Index des zweiten Argumentes im

internen Speicher hinterlegen. Das Ergebnis, das wieder auf den Stack gelegt werden soll, ist der gespeicherte Wert. Der Operator `load` ist ein unärer Operator, soll ein Argument entgegen nehmen und den Wert des internen Speichers am gegebenen Index laden und auf den Stack legen.

Um den Speicher des Taschenrechners zu manipulieren, fügen wir noch zwei weitere Befehle hinzu: `clear` soll den aktuellen Stack bereinigen, `reset` zusätzlich noch den Speicher zurücksetzten. Diese Befehle sind vor allem bei der direkten Benutzerinteraktion hilfreich.

Häufig gestellte Fragen

Wie funktioniert dieser QuickSort-Algorithmus aus der Vorlesung nochmal?

Zunächst wird das Array in zwei Teile (linker und rechter Teil) aufgeteilt. Das geschieht indem man ein **Pivotelement** auswählt und dann alle Elemente die **kleiner** sind in den **linken Teil** und alle Elemente die **größer oder gleich** sind in den **rechten Teil** verschiebt bzw. jeweils tauscht.

Das **Pivotelement** ist somit korrekt einsortiert worden (alle kleineren Elemente stehen links, alle größeren rechts von ihm). Anschließend wird der Algorithmus rekursiv auf beiden Teilen aufgerufen. Dabei arbeitet QuickSort auf einem Array und verwendet lediglich Indizes, um die linken und rechten Grenzen des Teils, auf dem der Algorithmus arbeitet, zu bestimmen. Die Rekursion bricht ab, wenn der Algorithmus auf einem Array-Teil mit nur einem Element aufgerufen wird.

Bisher haben wir noch nicht geklärt, wie man das **Pivotelement** ausgesucht wird. In DAP1 wird das letzte Element des jeweiligen Array-Teils gewählt. Die Implementierung von QuickSort aus DAP1 verwendet vier Indizes: `leftBound`, `rightBound`, `leftOfGreaterPart` und `candidate`. `leftBound` bezeichnet die linke, `rightBound` die rechte Grenze des Arrays. Zudem ist `rightBound` – wie eben besprochen – das **Pivotelement**. `leftOfGreaterPart` ist zu jedem Zeitpunkt das erste Element des **rechten Teils**. `candidate` ist der Index des aktuell betrachteten Elements.

Die Referenzimplementierung lautet wie folgt:

```
1  public static int[] array = {3, 7, 9, 4, 1, 8, 5, 2}; // example array
2
3  public static void swap(int i, int j) {
4      if (i != j) {
5          int temp = array[i];
6          array[i] = array[j];
7          array[j] = temp;
8      }
9  }
10
11 public static void quicksort(int leftBound, int rightBound) {
12     if (leftBound >= rightBound) {
13             return; // single element or bad parameters
14     }
15     int leftOfGreaterPart = leftBound;
16     for (int candidate = leftBound; candidate < rightBound; candidate++) {
17         if (array[rightBound] > array[candidate]) {
18             swap(candidate, leftOfGreaterPart);
19             leftOfGreaterPart++;
20         }
21     }
22     swap(leftOfGreaterPart, rightBound);
23     quicksort(leftBound, leftOfGreaterPart - 1);
24     quicksort(leftOfGreaterPart + 1, rightBound);
25 }
```

Meine Implementierung von `Stack` und `Element` erzeugt Generics-Fehler!

Wenn Du die Klassen wie beschrieben angelegt hast, sollte das nicht passieren. Schau Dir vielleicht noch einmal die Vorlesungsfolien 843 und folgende an – ein generischer Typparameter wird in alle inneren Klassen weitergegeben. Wenn Du also so etwas geschrieben hast:

```
1 public class Stack<T> {
2     private class Element<T> {
3         // code
4     }
5 }
```

hast Du *zwei* generische Parameter angelegt. Du willst aber, dass die zu einem Stack gehörenden Elemente Werte vom Typ `T` erhalten – der Typparameter der Klasse Element ist also unnötig.

Weitere Fragen für nächstes Mal

Wenn Du ein Detail der Vorlesung für sehr verwirrend oder missverständlich hältst, kannst Du uns gerne darauf ansprechen. Vielleicht nehmen wir die Frage in das Handbuch des nächsten Caffeine & Code auf.

Viel Spaß beim Bearbeiten der Aufgaben!

Das Team von Caffeine & Code

Lösungen

Aufgabe 17: Umformung in Reverse Polish Notation

Hinweis: Die Lösungen sind *nicht* eindeutig!

Infixterm	Postfixterm
$2 + 4 \times 3$	$2\,4\,3\,\times\,+$
$(2 + 4) \times 3$	$3\,2\,4\,+\,\times$
$1\;1 + 2 + 3 + 4$	$1\,2\,3\,4\,+\,+\,+$
$1 \times 2 + 3 \div 4 - 5 + 6$	$1\,2\,\times\,3\,4\,\div\,+\,5\,-\,6\,+$
$\sin(4 \times \pi)$	$4\,\pi\,\times\,\sin$
$\sin(\cos(\text{sqrt}(100)))$	$100\,\text{sqrt}\,\cos\,\sin$

Aufgabe 18: Auswertung von Reverse Polish Notation

Vorkommen von $*$ sind als $\times$ zu verstehen.

Postfixterm	Wert
$2\,4\,3\,\times\,+$	14
$2\,1\,-\,3\,+$	4
$4\,\text{sqrt}\,2\;\times\,2\,+$	6
$10\,20\,5\,5\,+\,\times\,+$	210
$10\,20\,+\,5\,5\,+\,\times$	300
$\pi\,2\,\div\,\sin\,\pi\,\times\,\sin\,\sin\,\pi\,\times\,\sin\,\sin$	0

Aufgabe 19: Der Java-Compiler

1. Angenommen, wir führen den Befehl

   ```
   javac -d dir de/example/Example.java org/library/Util.java
                   com/acme/ExpensiveTool.java
   ```

   aus. Welche der folgenden Aussagen sind korrekt?

   ☐ Der Quellcode befindet sich im Verzeichnis `dir`.

   Nein. `-d` benennt die **D**estination, das Verzeichnis, in dem die `.class`-Dateien angelegt werden.

   ☐ Es werden drei Dateien kompiliert.

   Ja.

- ☐ Der gesamte Quellcode wird in die Datei `ExpensiveTool.class` kompiliert.

  Nein. Die Namen der `.class`-Dateien bestimmen sich nach den Namen der kompilierten Klassen.

- ☐ Es werden mindestens drei `.class`-Dateien erzeugt.

  Ja. Enthält eine der Dateien jedoch innere Klassen, bestimmte Arten von Lambdas oder weitere, nicht-`public`-Klassen, werden diese in weitere `.class`-Dateien abgelegt.

- ☐ Ist das System auf italienische Sprache eingestellt, müssen wir `it/example/Example.java` schreiben.

  Nein. Das ist Blödsinn. Nur weil sich die Systemsprache ändert, ändert sich nicht der Dateibaum.

- ☐ Die kompilierten `.class`-Dateien werden nicht auf alten Java-Versionen laufen.

  Ja. Zu diesem Zweck dient die `-target`-Option.

- ☐ Enthält der Quellcode Syntaxfehler, wird dies auf der Kommandozeile angezeigt.

  Ja. Wie bei vielen Kommandozeilenprogrammen erfolgt die Ausgabe, wenn nichts anderes eingestellt wird auf der Kommandozeile.

- ☐ Die Quelldateien müssen in der Kodierung `ISO-8859-1` vorliegen.

  Nein. Die Dateien müssen in der Systemkodierung vorliegen, wenn nichts anderes angegeben ist.

2. Wozu dient der Parameter `-g` des Java-Compilers? In welchen Situationen ist welche Einstellung sinnvoll?

   Hier ist eine freie Antwort gefragt, das Ziel ist, dass die Teilnehmer über die Frage nachdenken, Wikipedia lesen etc.. Eine Antwort kann sein:

   Der Parameter steuert, welche Debug-Symbole in der `.class`-Datei abgelegt werden. Um das Programm debuggen zu können, ist es sinnvoll, alle Symbole anzulegen. Wenn das Programm an andere gegeben werden soll, brauchen diese einen Teil der Informationen nicht (z. B. über lokale Variablen), wenn diese eiegenen Code zur Verwendung mit unserem schreiben. Für Anwender sind die Symbole uninteressant und verschwenden nur Speicherplatz.

3. Über die Optionen `-source` und `-target` kann man Kompatibilität mit alten Java-Versionen herstellen. Was genau machen die Optionen und worin unterscheiden sie sich?

   `-source` steuert, welche Sprachfeatures (Lambdas, Try-with-resources etc.) verwendet werden dürfen. `-target` steuert, mit welcher JVM-Version die `.class`-Dateien als kompatibel markiert werden. Dadurch werden keine Features „zurückportiert“ – Lambdas auf alten JVMs gehen einfach nicht!

4. Welchen Befehl müsste man benutzten, um in dem obigen Beispielprojekt alle Java-Dateien zu kompilieren?

   ```
   javac -d bin src/de/tudo/meinprojekt/*.java
   ```

Aufgabe 20: Das Jar-Werkzeug

1. Angenommen, wir führen den Befehl

   ```
   jar -cfe ../example.jar de.example.Example de/example/Example.class
           org/library/Util.class com/acme/ExpensiveTool.class
   ```

   aus. Welche der folgenden Aussagen sind korrekt?

   ☐ Wir könnten statt `de.example.Example` auch `de/example/Example.class` schreiben.

   Ja. Beide Notationen sind äquivalent.

   ☐ Wenn wir versuchen, eine `.java`-Datei hinzuzufügen, wird diese automatisch kompiliert.

   Nein. Dazu dient der `javac`.

   ☐ Wir können statt `-cfe` auch `fec` schreiben.

   Ja. Der `-` ist optional, die Reihenfolge der Elemente auch. Insbesondere muss der Dateiname *nicht* unmittelbar auf die Option `f` folgen.

   ☐ `jar -xf ../example.jar` löscht die Datei `../example.jar` und extrahiert die Dateien darin ins aktuelle Verzeichnis.

   Nein. Die Dateien werden extrahiert, aber das Archiv nicht gelöscht.

   ☐ Wird die Option `-0` verwendet, wird das Archiv größer.

   Ja. Die Option deaktiviert die ZIP-Kompression, daher wird das Archiv größer.

   ☐ Lassen wir die Option `-f ../example.jar` weg, wird ein Dateiname automatisch gewählt und auf der Konsole ausgegeben.

   Nein. Beim Erstellen und Verändern von Archiven wird das neue *Archiv* auf der Kommandozeile ausgegeben, beim Aufzählen und Extrahieren wird der Dateiname von der Kommandozeile *gelesen*.

2. `.jar`-Archive sind „nur“ normale ZIP-Archive. Welche Vor- und Nachteile hat dies?

   Vorteile: Die Archive lassen sich auf jedem aktuellen System ohne Java-spezifische Werkzeuge auspacken und sogar erstellen. Das Format hat sich bereits in der Praxis bewährt, das Rad wird nicht neu erfunden. Außerdem ist das `.jar`-Archiv komprimiert, nimmmt also weniger Platz ein.

   Nachteil: Man bindet sich an das ZIP-Format. Will man Kompatibilität wahren, kann man keine neuen Kompressionsalgorithmen und andere Verbesserungen einführen, ohne alle ZIP-Tools erweitern zu müssen.

3. Welchen Befehl müsste man benutzten, um in dem Beispielprojekt zu Aufgabe 3 alle Dateien in ein `.jar` zu packen?

   Im Verzeichnis `bin` ausführen:

   ```
   jar cfe ../meinprojekt.jar de.example.Example de/example/*.class
   org/library/*.class com/acme/*.class data/*.txt
   ```

Sonstiges

Das Package-Statement muss nur in Quelldateien, nicht allen Dateien angegeben werden.