Java 8 Praktikum 1

Maximilian Heinrich Zender, Hector Smith, Steffen Giersch, Hendrik Albers, Jeremias Jonathan Twele

Aufgabe 1

1.: Zu finden in Aufgabe1/FunctionalInterfaces

2.: JDK 8 muss verwendet werden.
Ein Interface mit nur genau einer abstrakten Methode darf verwendet werden.
Die Argumente der Lambda-Funktion können ohne Datentypen angegeben werden.
Ein Lambda-Pfeil wird gesetzt.
Der Funktions-Block wird hinter dem „Lambda-Pfeil“ in geschweiften Klammern eingefügt.

3.: Anstatt für jedes Objekt eine neue .class-Datei zu erzeugen, wird nur eine Datei erzeugt.
Die neue „main“.class-Datei ist sehr viel größer, da sie die Inhalte aller anderen .class-Dateien enthält.
Die alte „main“.class-Datei enthält eine Referenz auf die .class-Datei der erstellten anonymen Klasse.

4.: In Nanosekunden messen.
Die Lambda-Notation scheint sehr viel langsamer als die anonyme-Klassen-Notation zu sein.

Aufgabe 2

1.:
Für das näherungsweise Integrieren von Funktionen haben wir die Trapezregel verwendet. Mit der Trapezregel lässt sich eine stetige Funktion *f* wie folgt näherungsweise integrieren

$$\sum_{k=a}^{b} d * \frac{f(k) + f(k+d)}{2}$$

Wobei $d = \frac{b-a}{g}$ gilt und die Schrittgröße *d* beträgt und *g* die Genauigkeit der Messung angibt. Dies lässt sich wie folgt in Pseudocode fassen:

```
Integriere(f,start,end,granularität)
schrittGröße = end – start
akku = 0

      if(start == end)
            return 0

for(a = start; a < end; a += schrittGröße)
            akku = akku + schrittGröße * ((f(a) + f(a + schrittGröße)) / 2)

return akku
```

2.:
Um „schriftliche“ und nicht näherungsweise Ableitungen zu bestimmen eignen sich binäre Bäume als Datenstruktur. Zu diesem Zweck wird eine Klasse *Node* genutzt die folgende Signatur hat:

```
### Variablen ###
String value;
Node leftChild;
Node rightChild;


### Konstruktoren ###
Node(String value, Node rightChild, Node leftChild)
Node(String value)
Node(Node other)


### Funktionen ###
applyOnValues(Function<String,String> f)
applyOnNode(Function<Node,Node> f)
```

Dabei können leftChild und/oder rightChild weitere Nodes enthalten, dann ist diese Node ein Knoten und muss einen Operator enthalten, oder sind beide leer, dann ist diese Node ein Blatt und muss entweder eine Variable oder eine Zahl enthalten.
Mit dieser Datenstruktur lassen sich alle mathematischen Funktionen als Baumstruktur darstellen. Zudem kann auf dieser Datenstruktur relativ einfach abgeleitet werden.

Um abzuleiten lässt sich eine einzelne Funktion mit der Signatur

```
Node differentiate(Node root)
```

Aufrufen, die abhängig der Eigenschaften von `root` die jeweilige Ableitungsregel (als Lambda-Funktion) auf `root` anwendet.

Aufgabe 3

Für den Aufgabenteil "Collections" haben wir folgende neue Methoden zur Erprobung ausgewählt:
- "filter" aus dem Interface "Stream":
  Gibt einen Stream zurück, der alle Elemente enthält, die mit der geg. Bedingung konform sind.
- "map" aus dem Interface "Stream":
  Wendet auf alle Elemente die geg. Funktion an und gibt diese als neuen Stream zurück.
- "forEach" aus dem Interface "Stream":
  Wendet die geg. Funktion auf alle Elemente an ohne einen neuen Stream zurückzugeben.
- "reduce" aus dem Interface "Stream":
  Akkumuliert die Elemente in dem Stream mittels der geg. Funktion.
- "removeIf" aus der Klasse "CopyOnWriteArrayList":
  Entfernt alle Elemente aus der Collection, für die die geg. Bedingung wahr ist.

Jede Methode werden wir mit Collections bestehend aus 100 und bestehend aus 1.000.000 Elementen ausführen und mit äquivalenten Nicht-Java-8-Methoden vergleichen.

Bei dem Vergleichen der verschiedenen Methoden der Collections haben sich folgende Messwerte ergeben:

100 Elemente
forEach lambda time: 133137489ns
iterator time: 145974ns

1.000.000 Elemente
forEach lambda time: 35828821ns
iterator time: 72023242ns

---

100 Elemente
map lambda time: 72704750ns
iterator time: 1419363ns

1.000.000 Elemente
map lambda time: 72322ns
iterator time: 589663ns

---

100 Elemente
filter lambda time: 1993941ns
iterator time: 255122ns

1.000.000 Elemente
filter lambda time: 405532ns
iterator time: 19703811ns

---

100 Elemente
reduce lambda time: 37668801ns
iterator time: 17748ns

1.000.000 Elemente
reduce lambda time: 10266533ns
iterator time: 11852280ns

---

100 Elemente
removeIf lambda time: 815058ns
iterator time: 9761ns

1.000.000 Elemente
removeIf lambda time: 15228757ns
iterator time: 6372714ns