Programmieren II

Informatik (B. Sc.)
2. Semester

Unit 02

Sequenzbasierte Rekursionen

Prof. Dr. Nane Kratzke

Units

Rekursion

01 Einführung in die rekursive Programmierung
02 Sequenzbasierte Rekursionen
03 Rekursive Datenstrukturen

Funktionale Programmierung

04 Einführung in die funktionale Programmierung
05 Funktionale Programmierung mit Streams
06 Thinking in Filter - Map - Reduce

Generische Datentypen <T>

07 Einführung in Gen. Datentypen (Type Erasure)
08 Bounded Types

Graphical User Interfaces

09 Einführung in Swing (Typvertreter)
10 Model - View - Controller (MVC)
11 MVC an einer Beispielanwendung

Unit 02

Sequenzbasierte Rekursion

mit volldynamischen Datenstrukturen

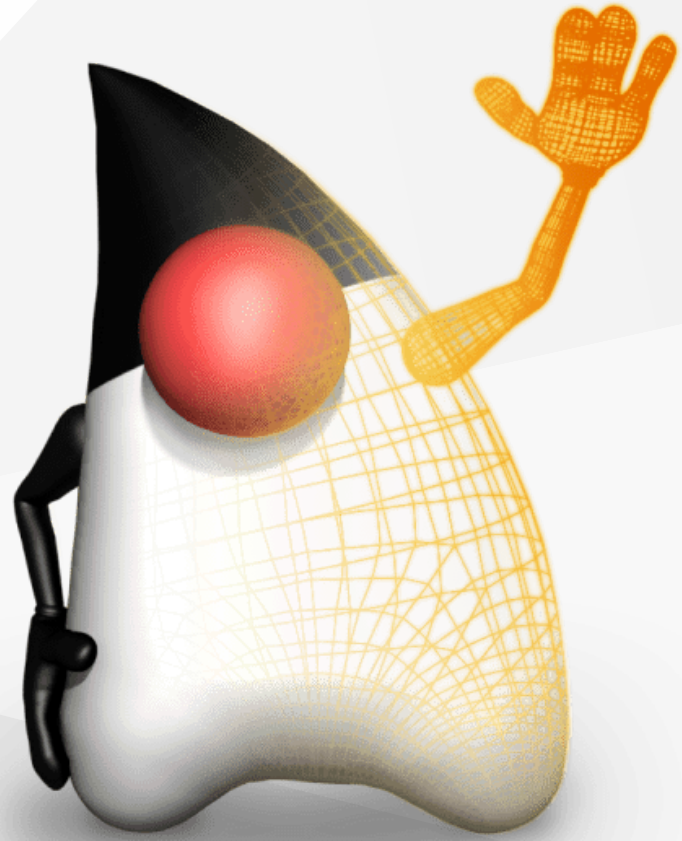
- `List`, `String` und `Iterator`
- Kopf und Rest von Sequenzen

mit semidynamischen Datenstrukturen

- Arrays
- Trick #1: Rekursion mit Zählparameter (Unit 01)
- Trick #2: Rekursion mit Iterator

Übungsaufgaben

- Rekursion auf Zeichenketten
- Rekursion auf Listen
- Rekursion auf Iteratoren
- Rekursion auf Arrays



Sequenzbasierte Rekursion

Head • Tail | Rest

Einführung in Rekursion mit Sequenzen

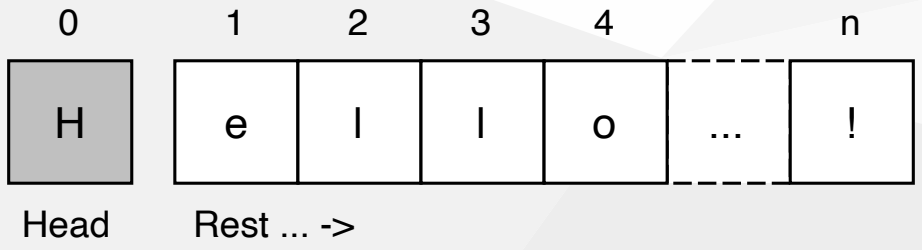
Rekursion ist ein fundamentaler Ansatz in der Informatik, um Probleme durch Zerlegung in kleinere Teilprobleme zu lösen.

Auch die Verarbeitung von Sequenzen (z. B. Arrays oder Strings) kann rekursiv erfolgen.

Grundprinzip: Head und Rest

Die Idee vieler rekursiver Ansätze auf sequenzartigen Datenstrukturen (Listen, Strings, etc.) besteht darin, eine Sequenz in zwei Teile zu unterteilen:

- Head: Das vorderste Element der Sequenz
- Rest: Die verbleibende Sequenz ohne das erste Element



Kochrezept für einfache sequenzbasierte Rekursionen

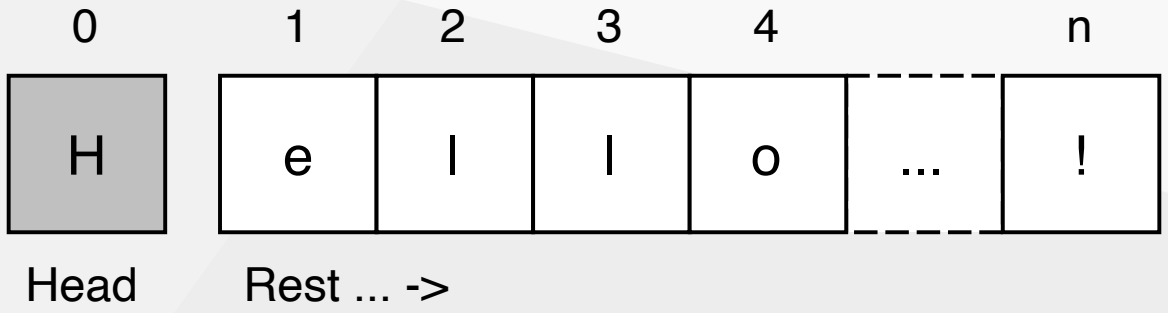
1. Methodensignatur (Methodenkopf):
   Jede rekursive Methode beginnt mit einer definierten Signatur, die den Rückgabetyp, den Methodennamen und die Parameter enthält. *Nicht anders als bei anderen Methoden auch.*
   Beispiel: `public static int countVowel(String text)`
2. Basisfall (Abbruchbedingung):
   Der Basisfall ist entscheidend, um eine endliche Rekursion zu gewährleisten. Er definiert den Punkt, an dem die Methode nicht mehr rekursiv aufgerufen wird. Normalerweise ist der Fall, wenn die Sequenz leer ist (trivialer Fall.)
   Beispiel: `if (text.isEmpty()) return 0; // Basisfall`
3. Bestimmung des Kopf des Problems:
   Normalerweise findet sich der Kopf an der Indexposition 0 bei Sequenzen.
   Beispiel: `char head = text.charAt(0);`
4. Bestimmung des Rest des Problems:
   Normalerweise findet sich der Rest ab der Indexposition 1 bei Sequenzen.
   Beispiel: `String rest = text.substring(1);`
5. Bestimmung des Kopfbeitrags zum Problem:
   Analyse des Kopfs. Das ist der Beitrag des für den Kopf verantwortlichen "Mönchs".
   Beispiel: `boolean vowel = "aeiou".contains("" + head).toLowerCase());`
6. Rekursiver Aufruf: Der rekursive Aufruf erfolgt innerhalb der Methode nach dem Basisfall auf dem Rest. Der Beitrag des Kopfs wird mit der Rückgabe des Rests kombiniert.
   Beispiel: `return (vowel ? 1 + 0) + countVowel(rest);`

Beispiel 1: Vokale zählen

Eine rekursive Implementierung zum Zählen wieviele Vokale in einem Text enthalten sind.

```java
public static int countVowel(String text) {                    // 1: Methodensignatur
  if (text.isEmpty()) return 0;                                 // 2: Basisfall (trivialer Sonderfall)
  char head = text.charAt(0);                                   // 3: Kopf des Problems
  String rest = text.substring(1);                              // 4: Rest des Problems (für die Mönche)
  int myPart = "AEIOUaeiou".contains(head + "") ? 1 : 0;        // 5: Kopfbeitrag zum Gesamtproblem
  return myPart + countVowel(rest);                             // 6: Rekursiver Aufruf (Antwort der Mönche)
}
```

Im Wesentlichen sind dabei nur die Schritte 5 und 6 problemspezifisch. Die Schritte 1 bis 4 können schematisch abgeleitet werden.



Beispiel 2: Summe einer Liste ganzer Zahlen

Eine rekursive Implementierung zum Summieren einer Liste:

```java
public static int sum(List<Integer> values) {          // 1: Methodensignatur
  if (values.isEmpty()) return 0;                      // 2: Basisfall (trivialer Sonderfall)
  int head = values.get(0);                            // 3: Kopf des Problems
  List<Integer> rest = values.subList(1, values.size()); // 4: Rest des Problems (für die Mönche)
  return head + sum(rest);                             // 5 + 6: Rekursiver Aufruf
}
```

Exemplarischer Aufruf mit folgenden Werten:

```java
import java.util.List;

[...]

public static void main(String[] args) {
  List<Integer> values = List.of(1, 2, 3, 4, 5);
  int n = sum(values);
  System.out.println(n); // => 15
}
```

Rekursion auf großen Datenstrukturen
Nutzung eines Iterators

Insbesondere wenn die zu verarbeitenden Listen sehr groß werden, kann es sinnvoll sein, eine speicherschonendere Form der rekursiven Iteration mittels eines Iterators zu definieren. Dies vermeidet, dass viele Teilkopien der zu verarbeitenden Liste im Hauptspeicher vorgehalten werden müssen.

Beispiel: Eine rekursive Implementierung zum Summieren mittels eines Iterators:

```
public static int sum(Iterator<Integer> values) {          // 1: Methodensignatur
  if (!values.hasNext()) return 0;                         // 2: Basisfall (trivialer Sonderfall)
  int head = values.next();                                // 3 + 4: Kopf des Problems
                                                           // values.next() bestimmt als
                                                           // Seiteneffekt den Rest automatisch.
  return head + sum(values);                               // 5 + 6: Rekursiver Aufruf
}
```

Oder vereinfacht sogar als Einzeiler:

```
public static int sum(Iterator<Integer> values) {
  return values.hasNext() ? values.next() + sum(values) : 0;
}
```

Rekursionen auf Arrays

Wie kürzt man etwas, was sich nicht kürzen lässt?

Problem

Das Prinzip von Rekursionen ist es,
ein Problem Aufruf für Aufruf kleiner zu machen.

Arrays kann man aber nicht kürzen.

Wir brauchen also einen Trick.

Trick #1
Einführen eines Zählparameters

Angenommen, wir wollen `int[] values = { 1, 2, 3, 4, 5 };` mit einer rekursiven Methode `int sum(int[])` verarbeiten, dann geht dies nicht, weil wir `values` nicht kleiner bekommen (Array Eigenschaft in Java).

Wir können aber einen Zählparameter `i` einführen, den wir von links (Index-Position 0) nach rechts bis zum Ende des Arrays laufen lassen.

```
public int sum(int[] values) {
  return sum(values, 0); // Einführen eines Zählparameters
                         // (wir beginnen unsere Rekursion ganz links bei 0)
}
```

Rückführung der Rekursion auf eine einfache Zählrekursion (siehe Unit 01). Wir überladen hierzu einfach die Methode `sum()` mit einem Zählparameter `i`.

```
public int sum(int[] values, int i) {    // i ist der Zählparameter
  if (i >= values.length) return 0;      // Rekursionsabbruch (rechter Rand des Arrays erreicht)
  return values[i] + sum(values, i + 1); // Einführen eines Zählparameters
}
```

Trick #2
Nutzung eines Iterators

Wir haben schon gesehen, dass man mit Iteratoren sehr kompakte Rekursionen schreiben kann.Wenn wir für ein Array also einen Iterator bauen könnten, hätten wir das Problem auch gelöst.

- Wir können einen Umweg über einen `Stream` machen. Der ist `iterable`. Wir müssen dann allerdings die Rekursion auf `Integer[]` und nicht auf `int[]` definieren (betrifft nur primitive Datentypen).
- Streams? `Integer` statt `int`? Werden wir alles noch im Detail in Unit 04 - 06 und Unit 07 behandeln.

```
import java.util.stream.Stream;
import java.util.Iterator;

public Integer sum(Integer[] values) {
  return sum(Stream(values).iterator()); // Umweg über Iterator.
}
```

Rückführung der Rekursion auf eine Iterator-basierte Rekursion.

```
public Integer sum(Iterator<Integer> it) {    // it ist der Iterator durch das Array
  return it.hasNext() ? it.next() + sum(it) : 0;
}
```

Übungsaufgaben

Rekursion auf Zeichenketten | Listen | Iteratoren | Arrays

1. Aufgabe: String-Rekursion

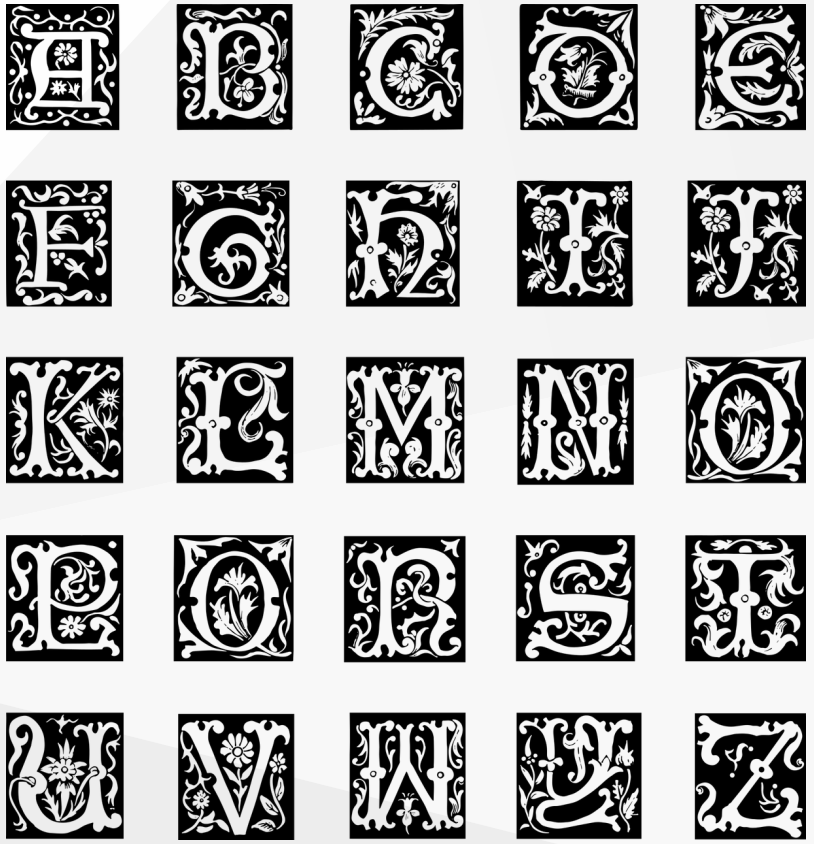
Schreiben Sie eine rekursive Methode `countChar()`, die ein Zeichen und eine Zeichenkette als Eingabe erhält und die Anzahl der Vorkommen des Zeichens (non case-sensitive) in der Zeichenkette zurückgibt.

Beispiele:

- `countChar('a', "banana")` sollte `3` zurückgeben.
- `countChar('o', "HELLO world")` sollte `2` zurückgeben.
- `countChar('X', "example")` sollte `1` zurückgeben.

Lösung:

>>> YOUR-TURN <<<



2. Aufgabe: Listen-Rekursion

Schreiben Sie eine rekursive Methode `findMax()`, die eine Liste von Ganzzahlen als Eingabe erhält und die maximale Zahl in dieser Liste zurückgibt.

- `findMax([1, 5, 3, 9, 2])` sollte `9` zurückgeben.
- `findMax([-3, -1, -7, -4])` sollte -1 zurückgeben.
- `findMax([10])` sollte 10 zurückgeben.
- `findMax([])` sollte `Integer.MIN_VALUE` zurückgeben (error indicator).

Lösung:

>>> YOUR-TURN <<<

Bildquelle: BenKirb (pixabay.com)

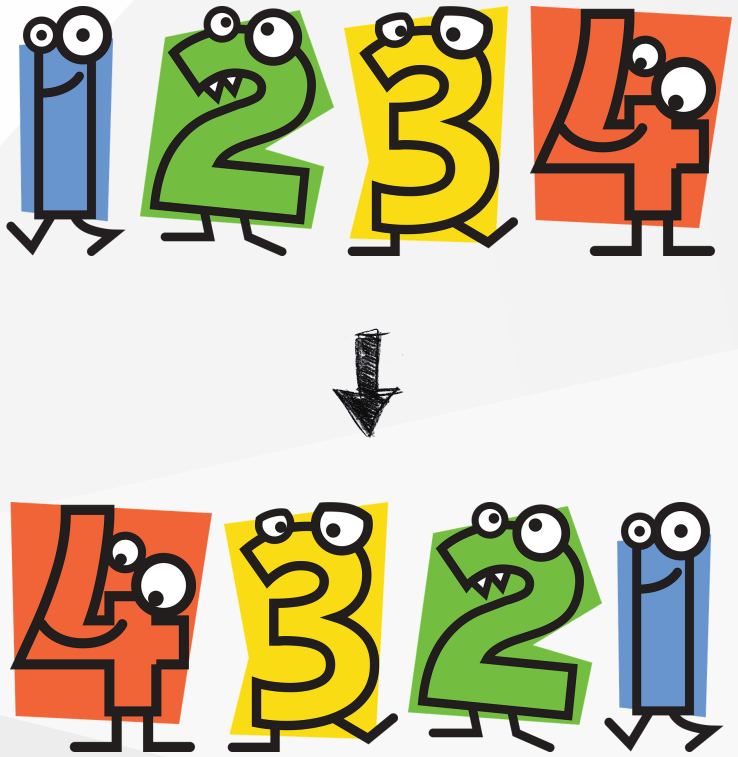
3. Aufgabe: Iteratoren-Rekursion

Schreiben Sie eine rekursive Methode `reverse()`, die einen Iterator von Ganzzahlen als Eingabe erhält und die Elemente als Liste in umgekehrter Reihenfolge ausgibt.

```
Iterator<Integer> in = List.of(1, 2, 3).iterator();
List<Integer> out = reverse(in);
System.out.println(out); // => [3, 2, 1]
```

Lösung:

>>> YOUR-TURN <<<

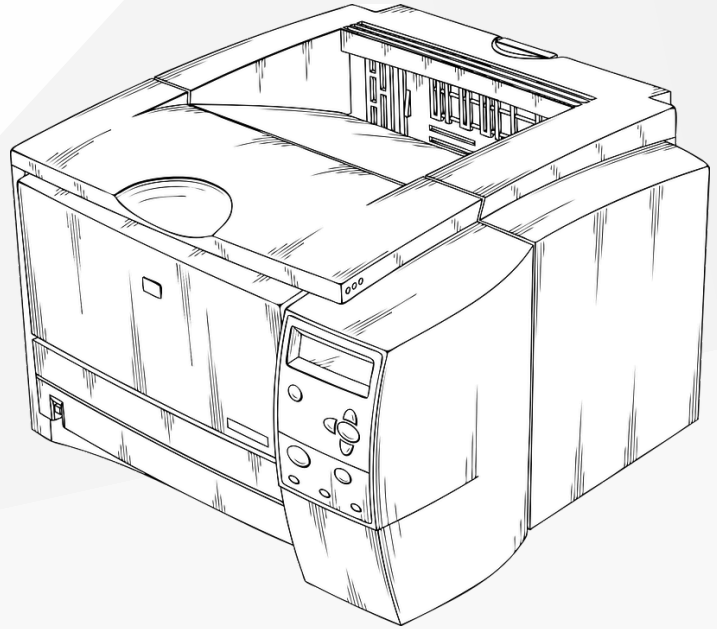


Bildquelle: BenKirb, geralt (pixabay.com)

4. Aufgabe: Rekursion auf Arrays

Schreiben Sie eine rekursive Methode `prettyPrint()`, die ein Array von Zeichen als Eingabe erhält und eine Ausgabe des Arrays als String erzeugt, wie auch in Listen in Java standardmäßig ausgegeben werden.

```
Character[] example = { 'a', 'x', 'z' };
String pretty = prettyPrint(example);
System.out.println(pretty); // => [a, x, z]
```



Entwickeln Sie die Rekursion einmal durch einführen

- eines Zählparameters
- eines Iterators

Rekursive Lösung mit Zählparameter:

>>> *YOUR-TURN* <<<

Rekursive Lösung mit Iterator:

>>> *YOUR-TURN* <<<

Zusammenfassung

- Rekursion auf Sequenzen
  Die Idee vieler rekursiver Ansätze auf sequenzartigen Datenstrukturen (Listen, Strings, etc.) besteht darin, eine Sequenz in zwei Teile zu unterteilen. Das vorderste Element der Sequenz und den Rest der Sequenz ohne das erste Element.
- Speicherschonende Rekursion mittels Iteratoren
  Iteratoren ermöglichen es, Rekursionen auf großen Listen zu definieren, ohne dass dafür viele Teilkopien der zu iterierenden Liste im Hauptspeicher vorgehalten werden müssen.
- Rekursion auf Arrays
  Datenstrukturen deren Länge nicht anpassbar ist, kann man entweder einen Indexzähler als zusätzlichen Parameter einführen, oder (ggf. über Umwege) einen Iterator definieren und auf diesen die Rekursion definieren.
- Rekursiver Mindset
  *Sei wie die **Mönche** im Tempel von Benares - faul aber clever! Kümmere dich nur um den Kopf der Sequenz! Andere (die Rekursion) sollen den Rest machen.*

Kontakt

Prof. Dr. Nane Kratzke

Technische Hochschule Lübeck

Mail: nane.kratzke(at)th-luebeck.de

Code Strong!

Alle Icons stammen von iconify.design • Bilder von Pixabay

Lösungsvorschlag zur 1. Aufgabe (Rekursion auf Zeichenketten)

```java
public class Main {

  public static int countChar(char c, String text) {
    if (text.isEmpty()) return 0;
    char head = Character.toUpperCase(text.charAt(0));
    String rest = text.substring(1);
    return (head == Character.toUpperCase(c) ? 1 : 0) + countChar(c, rest);
  }

  public static void main(String[] args) {
    int n = countChar('a', "banana"); // => 3
    n = countChar('X', "example"); // => 1
    n = countChar('o', "HELLO world"); // => 2
    System.out.println(n);
  }
}
```

Lösungsvorschlag zur 2. Aufgabe (Rekursion auf Listen)

```java
import java.util.List;

public class Main {

  public static int findMax(List<Integer> values) {
    if (values.isEmpty()) return Integer.MIN_VALUE;
    int head = values.get(0);
    List<Integer> rest = values.subList(1, values.size());
    return Math.max(head, findMax(rest));
  }

  public static void main(String[] args) {
    int n = findMax(List.of(1, 5, 3, 9, 2)); // => 9
    n = findMax(List.of(-3, -1, -7, -4)); // => -1
    n = findMax(List.of(10)); // => 10
    n = findMax(List.of()); // => Integer.MIN_VALUE
    System.out.println(n);
  }
}
```

Lösungsvorschlag zur 3. Aufgabe (Rekursion auf Iteratoren)

```java
import java.util.*;

public class Main {

  public static List<Integer> reverse(Iterator<Integer> it) {
    if (!it.hasNext()) return new ArrayList<>();
    int head = it.next();
    List<Integer> rest = reverse(it);
    rest.add(head);
    return rest;
  }

  public static void main(String[] args) {
    Iterator<Integer> in = List.of(1, 2, 3).iterator();
    List<Integer> out = reverse(in);
    System.out.println(out); // => [3, 2, 1]
  }
}
```

Lösungsvorschlag zur 4. Aufgabe (Rekursion auf Arrays)

```java
import java.util.*;
import java.util.stream.*;

public class Main {

    public static String prettyPrint(Character[] values) {
        return "[" + prettyPrint(values, 0) + "]";
        // String inner = prettyPrint(Stream.of(values).iterator());
        // return "[" + inner.substring(0, inner.length() - 2) + "]";
    }

    public static String prettyPrint(Character[] values, int i) { // <= Trick #1: Einführen eines "Zählparameters"
        if (i == values.length - 1) return values[i] + "";
        return values[i] + ", " + prettyPrint(values, i + 1);
    }

    public static String prettyPrint(Iterator<Character> it) { // <= Trick #2: Einführen eines Iterators
        return it.hasNext() ? it.next() + ", " + prettyPrint(it) : "";
    }

    public static void main(String[] args) {
        Character[] example = { 'a', 'x', 'z' };
        String pretty = prettyPrint(example);
        System.out.println(pretty); // => [a, x, z]
    }
}
```