LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Einführung in die Informatik:
Programmierung und Softwareentwicklung

Wintersemester 2018/19

Überblick und Wiederholung

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik

Institut für Informatik, LMU München

WS 2018/19

Stand der Folien: 6. Februar 2019

Klausurinformationen

- 6 ECTS: Klausur 08.02.2019 um 12:15 (120 Minuten Bearbeitungszeit)
  - Raum: Hörsaal B 101, Hauptgebäude, Geschw.-Scholl-Pl. 1

- 9 ECTS: Klausur 08.02.2019 um 12:15 (150 Minuten Bearbeitungszeit)
  - Raum: Hörsaal B 201, Hauptgebäude, Geschw.-Scholl-Pl. 1

- Keine Hilfsmittel (weder gedruckt noch elektronisch)!
- Eine Anmeldung per UniWorX ist zwingend nötig.

Bitte seien Sie um **12:00 Uhr** im jeweiligen Raum,
damit wir so schnell wie möglich beginnen können!

- Einsicht: 26.02.2019 um 13 Uhr F 003 Oettingenstr. 67

Nachholklausur

- 25.03.2019 12:00 Uhr
  (Raum wird auf der Website bekannt gegeben)

- Die Nachholklausur kann mitgeschrieben werden, **egal** ob Sie an der regulären Klausur angemeldet waren oder teilgenommen haben.

- Mitschreiben zur Notenverbesserung muss mit dem jeweiligen Prüfungsamt geklärt werden.

- Eine Anmeldung per UniWorX ist zwingend nötig.

Probeklausur

- Auf der Website steht eine Probeklausur mit Musterlösung für beiden Varianten der Vorlesung zu Verfügung.

- Die Probeklausur wird nicht in Vorlesung, Zentralübung oder Tutorien besprochen.

Inhaltsverzeichnis der Vorlesung

Für 9 ECTS: Stoff der Vorlesung am 6.2.2019 nicht klausurrelevant

Kapitel 1: Java Übersicht

- Java ist eine imperative objekt-orientierte Programmiersprache.
- Die Programme sind plattformunabhängig, d.h. sie können ohne Änderungen z.B. unter Windows, OS X, Linux ausgeführt werden.
  - Java-Programme werden mit dem Compiler `javac` in Bytecode übersetzt.
  - Der Bytecode wird mit der Java Virtual Machine `java` ausgeführt.
- Geeignete Formatierung steigert die Verständlichkeit von Programmcode.
- Kommentare
  - `// KOMMENTAR` für einzeilige Kommentare
  - `/* KOMMENTAR */` für mehrzeilige Kommentare
  - `/** JAVADOC */` für javadoc-Kommentare

Kapitel 2: Syntax – EBNF-Grammatik, Ableitung

- Die Backus-Naur-Form ist eine Notation für Grammatiken.
- Aufbau einer Grammatik:
  - Startsymbol
  - Regeln der Form `Nichtterminal = Ausdruck`, wobei `Ausdruck` eine Kombination aus Nichtterminalen, Terminalen und Operatoren ist
    - `E1 E2`
    - `E1 | E2`
    - `[E1]`
    - `{E1}`
- Ableitung eines Worts: lang/kurz
  - Ersetzung von Nichtterminalen durch die rechte Seite der Regel oder/und
  - Ausführung von Operatoren

Eingesendete Fragen: Zu Ableitungen

Frage

Darf man bei der langen Ableitung in einem Schritt dieselben Nichtterminalsymbole/Operatoren mehrmals ersetzen/anwenden (wie in den Vorlesungsfolien; vgl. VL 2 Folie 21) oder ist nur ein Nichtterminalsymbol/Operator pro Schritt zulässig (wie in der ZentralÜbung)?

Man darf in einem langen Ableitungsschritt

- **entweder** (ein oder mehrere) Nichtterminalsymbole ersetzen (aber keine die entstehen)
- **oder** (einen oder mehrere) Operatoren anwenden

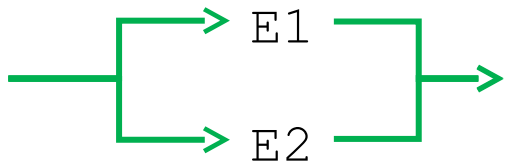
$\rightarrow$ Immer nur eines ist eine lange Ableitung,
$\rightarrow$ aber man darf auch mehr in einem Schritt machen

Kapitel 2: Syntax – Syntaxdiagramm

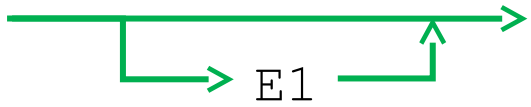
BNF-Grammatiken und Syntaxdiagramme sind äquivalent.

- Nichtterminale -> Rechtecke
- Terminale -> Ovale
- Operatoren -> Pfeile bzw. Verzweigungsstruktur
  - `E1 E2`

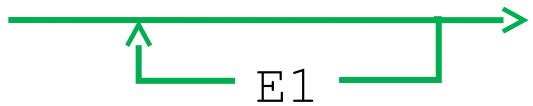
  - `E1 | E2`



  - `[E1]`



  - Zusätzlich: `{E1}`



Kapitel 3: Grunddatentypen

- Zahlen:
  - Ganze Zahlen:
    byte (2^7-1) < **short** $(2^{15}-1)$ < **int** $(2^{31}-1)$ < **long** $(2^{63}-1)$
  - Gleitkommazahlen: **float** (7 Stellen) < **double** (15 Stellen)
  - Operationen: `+ - * / %` (`< <= > >= ==`)

  Typecasting = Erzwingen einer Typkonversion zum Typ `type`
  durch Voranstellen von `(type)`
  ! Automatische Typkonversion zum größeren Typ !
- Einzelne Zeichen: **char** z.B. `'a'`
- (Zeichenketten: `String`)
- Wahrheitswerte: **boolean** mit Operationen `!, &&, &, ||, |`

Kapitel 3: Korrektheit und Auswertung

- Syntaktische Korrektheit von Ausdrücken
  = Ableitbarkeit in der EBNF-Grammatik
- Typkorrektheit von Ausdrücken
  = Zuordbarkeit eines Typen
  ! Achtung: Präzedenzen

Operation	Präzedenz
`!`, unäres `+-`	14
`(type)`	13
`*`, `/`, `%`	12
binäres `+`, `-`	11
`>`, `>=`, `<`, `<=`	9
`==`, `!=`	8
`&`	7
`|`	6
`&&`	4
`||`	3

- Auswertung von Ausdrücken
  - Vollständige Klammerung gemäß *Präzedenzen*
  - Auswertung unter Berücksichtigung der Klammern
    - Der Wert von Variablen ist durch den Zustand (σ,η) bestimmt.
    - Operationen, Methoden, Attributzugriff, Arrayzugriff… sind auszuwerten.

Kapitel 4: „Praktisches Programmieren”

- `main`-Methode
- Lokale Variablendeklaration z.B. `int i = 1;`
- Zuweisung z.B. `i = 3;`
- Block -> Gültigkeitsbereiche
- Speicherentwicklung
- `if`-Anweisung
- `for`-Anweisung
- `while`-Anweisung

Übungsaufgaben anschauen und *selbst* programmieren!

Kapitel 5+6: Klassen

- Aufbau einer Klassendeklaration (UML-Notation)
  - Attribute
  - Konstruktoren
  - Methoden
- Initialisierung und Verwendung von Objekten

➡ Unterschied Klasse-Objekt

- Objekte im Speicher (Stack und Heap)
- Statische Methoden vs. Instanzmethoden

Übungsaufgaben anschauen und *selbst* programmieren!

Kapitel 7: Vererbung

- Subtyping: Oberklasse – Unterklasse mit `extends`
  - Vererbung von Attributen
    ➡ Achtung: auf geerbte private Attribute kein Zugriff mit `.`
  - Keine Vererbung von Konstruktoren
    ➡ Aber Aufruf mit `super(…)`
  - Vererbung von Methoden
    ➡ Überschreiben in der Unterklasse möglich

Übungsaufgaben anschauen und *selbst* programmieren!

- (Abstrakte Klassen und) Interfaces
  (Benutzung siehe Grafische Benutzeroberflächen)

Kapitel 8: Grafische Benutzeroberflächen

- Grafische Benutzeroberflächen bieten eine benutzerfreundliche Kommunikationsmöglichkeit mit Programmen.
- Vorgehensweise:
  1. Erstellung des strukturellen Aufbaus der GUI:
     Aufbau aus den Übungen (`JFrame`, `JButton`, `JTextArea`, `JPanel`…)
     - `ContentPane`
     - `GridLayout` mit Zeilen und Spalten
  2. Verbindung der GUI mit den inhaltlichen Objekten der Anwendung: Stichwort "Modell einer GUI"
  3. Ereignisgesteuerte Behandlung von Benutzereingaben (z.B. Knopfdruck):
     - `actionPerformed(ActionEvent e)`
     - `JOptionPane.showInputDialog(…)`

Kapitel 9: Arrays

- Ein Array ist ein **Tupel** von Elementen **gleichen** Typs
  z.B. Grunddatentyp, Klassentyp, Arraytyp.
  - **Feste** Länge, d.h. Vergrößerung nur durch Kopieren möglich (O(n))
    z.B. `char[] charArray = new char[5];`
    z.B. `int[] intArray = {1,2,3};`
  - Direkter Zugriff in **O(1)** z.B. `int a = intArray[1];`
  - Veränderung über Index z.B. `intArray[0] = 10;`
- Arrays im Speicher (Stack und Heap)

Algorithmen müssen meist durch das komplette Array laufen:
`for`-Schleife

Kapitel 10: Komplexität von Algorithmen

- Zeitbedarf und Speicherplatzbedarf: O-Notation
  - Worst-case
  - Average-case
  - Best-case
- Algorithmen
  - Binäre Suche in einem sortierten Array:
    Zeitkomplexität O(log n), Speicherplatzbedarf O(1)
  - Bubble-Sort:
    Zeitkomplexität $O(n^2)$, Speicherplatzbedarf O(1)
  - Selection-Sort:
    Zeitkomplexität $O(n^2)$, Speicherplatzbedarf O(1)

ENDE 6 ECTS

Eingesendete Fragen: Komplexität und $\mathcal{O}$-Notation

Frage

Wie man bei Methoden die Speicher und Zeitkomplexität berechnet und wie man das dann in O-Notation beschreibt. wie man Zeit- und Speicherplatzkomplexitäten bestimmt? Oder sollen wir diese „wichtigsten“ Algorithmen auswendig lernen?

Eingesendete Fragen: Komplexität und $\mathcal{O}$-Notation (2)

Zeitkomplexität:

- Einzelne Anweisungen (wie Zuweisung, Wert nachschauen, if-then-else-Zweig auswählen) und Auswertung von Operatoren zählen je als **ein Schritt**,
- Schleifendurchläufe führen diese Operationen so oft durch, wie man durch die Schleife läuft.
- Bei rekursiven Methoden muss man Zählen wie oft die Methode aufgerufen wurden.

Speicherkomplexität:

- Jede Variable für einen Grundtyp zählt 1
- Arrays mit Grundtypinhalt zählen: Länge des Arrays
- Objekte: alle Attribute zählen
- Rekursive Methoden: Zusätzlicher Platz von 1 für jeden rekursiven Aufruf (um das Ergebnis nicht zu verlieren)
- **Platz für die Eingabe** zählt bei uns nicht mit

$\mathcal{O}$-Notation

- Die $\mathcal{O}$-Notation „fasst konstante Anteile zusammen, d.h. statt genau zu zähhlen wird die Größenordnung bestimmt
- $\mathcal{O}(1) \subseteq \mathcal{O}(\log n) \subseteq \mathcal{O}(n) \subseteq \mathcal{O}(n \log n) \subseteq \mathcal{O}(n^2) \ldots \subseteq \mathcal{O}(n^k) \subseteq \mathcal{O}(2^n)\ (k \geq 3)$

```
// Beispiel 1
for (int i=1; i <= n; i++) { // Kode mit konstant vielen Operationen
}
// Beispiel 2
for (int i=1; i <= n; i++) {
   for (int j=1; j <= n; j++) { // Kode mit konstant vielen Operationen
  }
}
// Beispiel 3
for (int i=1; i <= n; i++) {
   for (int j=1; j <= m; j++) { // Kode mit konstant vielen Operationen
  }
}
// Beispiel 4
for (int i=1; i <= n; i++) {
   for (int j=1; j <= m; j=j*2) { // Kode mit konstant vielen Operationen
  }
}
```

Kapitel 11: Rekursion

- Eine Methode ist rekursiv, wenn in ihrem Rumpf die Methode selbst wieder aufgerufen wird.
- Implementierung mit Hilfe einer **`if`**-Anweisung:
  - Basisfall: sofortige Terminierung z.B. **`return`** `1;`
  - „**`else`**“-Fall: rekursiver Aufruf

Meist geben Aufgaben ein rekursives Konzept vor!

- Jeder rekursive Algorithmus kann auch iterativ gelöst werden, aber rekursive Algorithmen sind oft „schöner“.
  z.B. Quicksort

Eingesendete Fragen: Rekursion

Frage

Ein paar Beispiele mehr für rekursive Funktionen wäre super: (eventuell noch einmal erklären, wie die Übung 11-2 zustande kommt).

Rekursion: Beispiele

Summe der Zahlen von 1 bis n

- Basisfall ($n = 1$) Summe der Zahlen von 1 bis 1 ist 1
- Rekursiver Fall: Summe der Zahlen von 1 bis n für $n > 1$: $n +$ (Summe der Zahlen von 1 bis $n - 1$)

```
public static int sum(int n) {
 if (n==1) {return 1;}
 else {return n + sum (n-1);}
}
```

Rekursion: Beispiele

Verzinsung:

- Auf ein Sparbuch werden 5000 EUR einbezahlt.
- Es wird jährlich mit 2% verzinst.
- Wie hoch ist der Sparbetrag nach n Jahren?

Rekursive Lösung:

- Basisfall: Kapital im Jahr 0: 5000 EUR
- Rekursiver Fall:

  Kapital im Jahr $n =$
  (Kapital im Jahr $n-1$) + $0.02*$(Kapital im Jahr $n-1$)

```
public static double kapital(int n) {
    if (n==0) {
      return 5000;
    }
    else {
      return (kapital(n-1)+ 0.02*kapital(n-1));
   }
}
```

Möglichkeiten beim Lotto (siehe Zentralübung 10)

Rekursiv die Anzahl der Möglichkeiten beim Lotto für „k aus n“ berechnen (z.B. „6 aus 49“)

- Basisfall: n aus n: 1 Möglichkeit
- Basisfall: 1 aus n: n Möglichkeiten
- Allgemeiner Fall:

$$\begin{array}{rl} & \underbrace{\text{Alle Möglichkeiten ohne die Zahl } n \text{ zu berücksichtigen}}_{= \text{ Möglichkeiten für } k \text{ aus } n-1} \\ + & \underbrace{\text{Möglichkeiten, wenn eine der gezogenen Zahlen die Zahl } n \text{ ist}}_{= \text{ Möglichkeiten für } k-1 \text{ aus } n-1} \\ \hline = & \text{Alle Möglichkeiten für } k \text{ aus } n \end{array}$$

```
public static int lotto(int k, int n) {
    if (k == n) {return 1;}
    else {if (k == 1) {return n;}
      else {return lotto(k,n-1) + lotto(k-1,n-1);}
  }
}
```

Kapitel 12: Ausnahmen

- Ein Programm heißt **robust**, falls es für jede (auch fehlerhafte) Eingabe eine sinnvolle Reaktion produziert.
- checked exceptions…
  - …erben von **`Exception`**
  - …müssen behandelt werden
- unchecked exceptions…
  - …erben von **`RuntimeException`**
  - …müssen **nicht** behandelt werden
- Ausnahmesituation erkennen
  - Objekte: Methodenaufruf auf Objekt mit Wert `null`
  - Arrays/Listen: Arrayzugriff außerhalb der Länge des Arrays
- Ausnahme auslösen: `throw`-Anweisung
  Achtung: `throws`-Deklaration für checked exceptions nötig!
- Ausnahme behandeln: `try-catch-finally`-Block

Kapitel 13: Listen

- Eine Liste ist eine endliche **Folge** von Elementen **gleichen** Typs.
  - **Dynamische** Länge, d.h. Vergrößern möglich
  - Zugriff je nach Implementierung
  - Veränderung je nach Implementierung
- Implementierung: Einfach-verkettete Liste
  - Direkter Zugriff in **O(n)**,
    da intern sequentiell durch die Liste gelaufen werden muss.
  - Veränderung über Vorne/Hinten-Anhängen bzw. über Index
- Implementierung: Doppelt-verkettete Liste

  ➡ Hinzufügen/Löschen am Ende in konstanter Zeit möglich

Praxis: `LinkedList<E>` und `Iterator<E>`

Eingesendete Fragen: Iteratoren

Frage

Nochmal kurz erklären, wie der Iterator genau funktioniert und ein Beispiel.

- Ein Iterator-Objekt wird mit der `iterator()`-Methode erzeugt
- Standard-Listen-Implementierung haben diese Methode (weil sie das Interface Iterable implementieren)

```
Iterator<Double> meinIterator = list.iterator();
```

- Ein Iterator hat nur 2 Methoden:
  - `boolean hasNext()  // gibt es noch Elemente`
  - `Double next()      // gib das naechste Element`

Eingesendete Fragen: Iteratoren (2)

Verwendung (beispielhaft):

```
LinkedList<Double> list = new LinkedList<Double> ();
list.addFirst(1.0);
list.addFirst(2.0);
list.addFirst(3.0);
double sum = 0.0;
Iterator<Double> myIterator = list.iterator();
while (myIterator.hasNext()){
 sum += myIterator.next();
}
```

Kurzform:

```
...
for (Double d : list) {
 sum += d;
 }
```

Kapitel 14: Bäume

- Ein Baum ist eine **hierarchische** Struktur von Elementen **gleichen** Typs.
  - Knoten speichern Elemente.
  - Knoten sind durch Kanten zu einer hierarchischen Struktur verbunden.
  - Jeder Knoten kann mehrere Nachfolger haben.
- Implementierung: Analog zu verketteten Listen, aber mit zwei oder mehr „nächsten“ Elementen (=Nachfolgern)
  - Zugriff auf ein Element in **O(log n)** für binäre Suchebäume
  - Einfügen eines neuen Elements in **O(log n)** für binäre Suchebäume

Operationen auf Bäumen sind meist rekursiv!
(siehe Tiefen- oder Breitendurchlauf)

Viel Erfolg bei der Klausur!