

Informatik I - Zusammenfassung

16. Juni 2021

Inhaltsverzeichnis

1 Funktionen und Module	1
1.1 Methoden	1
1.1.1 Felder als Formalparameter	1
1.1.2 Der Laufzeitstapel	1
1.1.3 Rekursion	2
1.2 Modulare Programmierung	4
2 Strukturierte Daten	5
2.1 Weiteres zu Feldern	5
2.2 Verbunddaten	6
2.3 Verkettete Datenstrukturen	8
2.4 Speicherorganisation	9
3 Objektorientierte Programmierung	10
3.1 Klassentypen verwenden	10

1 Funktionen und Module

1.1 Methoden

1.1.1 Felder als Formalparameter

Merke 1.1.1: Call-by-reference (Strukturierte Datentypen)

Übergabe der Parameter an Funktionen mithilfe einer Referenz. Änderungen wirken sich auf den ursprünglichen Wert aus!

Operationen mit Variablen des methodenaufrufenden Codes selbst.

Strukturierte Daten werden per **Referenz** übergeben! Änderungen der per Referenz übergebenen Parameter sind im **aufrufenden Code** sichtbar!

Beispiel: Einträge in übergebenen Feldern werden durch Methoden verändert! Das passiert bei reinen Integern (unstrukturierten Daten) nicht!

1.1.2 Der Laufzeitstapel

Merke 1.1.2: Definition

Datenstruktur zur Verwaltung des Speichers der von den Methoden verarbeitet wird.
Speicherbereich, in dem lokale Variablen abgespeichert werden.

Merke 1.1.3: Stack Frame - Aktivierungsrahmen

Abschnitt im Laufzeitstapel mit entsprechenden Speicherplätzen wird bei Aufruf **neu angelegt** und mit **[V,R]** initialisiert. Wird nach Wertübergabe (z.B. return) an Aufrufer freigegeben.

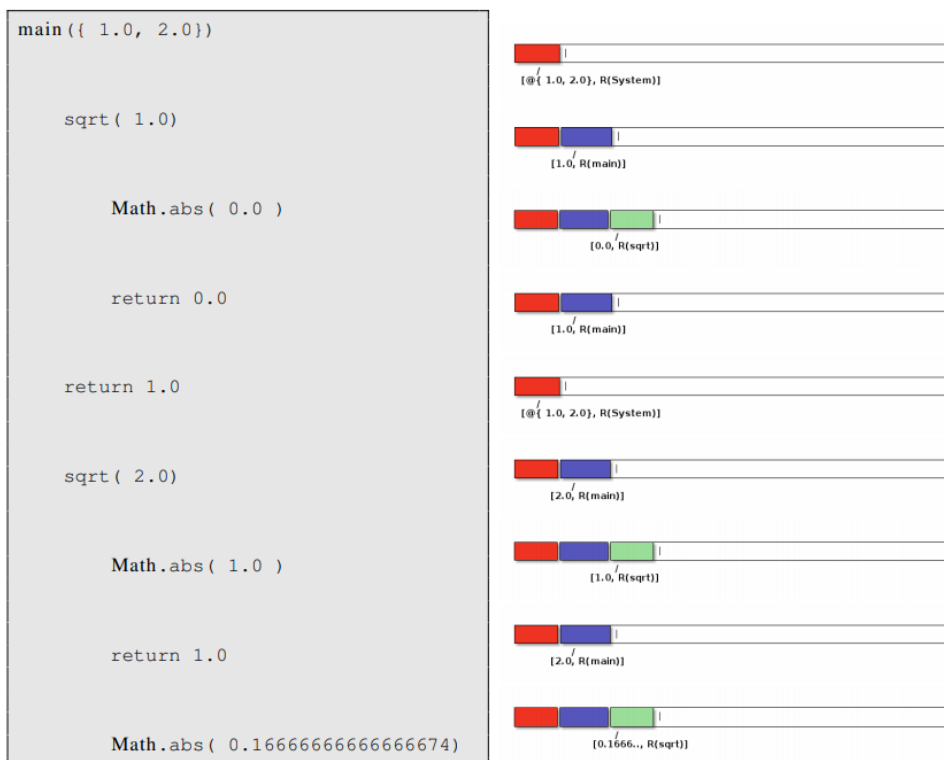
Merke 1.1.4: Relevante Daten: V,R

- **V:** Werte aller **lokalen Variablen** (Formalparameter + im Rumpf deklariert)
- **R:** **Rücksprungsadresse** = Adresse des Befehls der dem Aufruf folgt (z.B. Vorherige Methode)

Der **Laufzeitstapel**

= Speicherbereich, in dem lokale Variablen abgelegt werden

Methodentracing aus der Vogelperspektive



Analogie: Aktenstapel

1.1.3 Rekursion

Merke 1.1.5: Rekursion

Wird durch den Laufzeitstapel ermöglicht.

Rekursion = sich selbst aufrufende Funktionen.

Beispiel: Fakultätsfunktion.

Grundmuster: **Initialisiere** - **Terminierte (falls trivial)** - **Rekurriere**

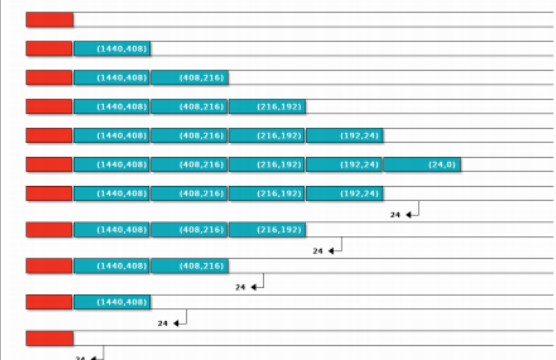
Jede Rekursion muss zu einem **Basisfall** führen. Sonst: `STACKOVERFLOWERROR`

- **Einfache Rekursion:** Lösung wird aus einem kleinen Vorgängerproblem gelöst. Geringe Aufwandreduktion pro Rekursionsschritt.
- **Mehrfache Rekursion:** Gesamtaufwand gleichmäßig aufteilen. Verwende **Teile- und Herrsche- Algorithmus**:
 - Teile Problem in mehrere Unterprobleme ähnlicher Größe
 - Löse die Unterprobleme durch Teillösungen
 - Kombiniere die Teillösungen zur Problemlösung

Beispiel: Berechnung des größten gemeinsamen Teilers

```
public static int ggT( int x, int y) {  
    if ( y == 0) return x;  
    return ggT( y, x % y);  
}
```

<https://www.stud.informatik.uni-goettingen.de/inf01/java/EuklidRek>



```
ggT(1440, 408)  
  ggT(408, 216)  
    ggT(216, 192)  
      ggT(192, 24)  
        ggT(24, 0)  
          return 24  
        return 24  
      return 24  
    return 24  
  return 24  
return 24
```

Korrektheitsbeweis

- Für beliebige $x \in \mathbb{N}$ gilt $\text{ggT}(x, 0) = x$.
- Ist $y \neq 0$, so gilt für jeden Teiler t von y :

$$t \text{ teilt } x \iff t \text{ teilt den Rest } r = x \bmod y,$$

- **Folglich** Ist $y \neq 0$, so $\text{ggT}(x, y) = \text{ggT}(y, x \bmod y)$.

Merke 1.1.6: Rekursionstiefe

ist die Tiefe seines Rekursionsbaums (maximale Anzahl von Kanten auf einem Wurzel -> Blatt-Pfad)

Merke 1.1.7: Traversierung

- **Inorder-Travesierung:** Links - Wurzel - Rechts (z.B. Scheiben)
- **Postorder-Travesierung:** Links - Rechts - Wurzel (z.B. Fibonacci)

Merke 1.1.8: Memoisation

Wiederholungen vermeiden durch **zwischenspeichern** in einem Feld.
Beispiel Fibonacci:

```
static int[] f = new int[N];  
public static void mFib() {  
    f[0] = 0; f[1] = 1;  
    for (int i = 2; i < N; ++i)  
        f[i] = f[i-1] + f[i-2];  
}
```

Wichtig 1.1.1: Rekursion

- Es wird ein Basisfall benötigt!
- Wir müssen die Problemgröße durch Rekursion reduzieren.
- Der Speicherbedarf darf nicht zu groß werden.
- Wiederholte Berechnungen vermeiden!

1.2 Modulare Programmierung

Verwendung von (eigens) programmierten Klassen in anderen Programmen/Klassen.

Merke 1.2.1: public, Clienten

Zugriff wird durch das Schlüsselwort PUBLIC ermöglicht!

- **Clienten:** Eine Klasse, die Dienste einer anderen Klasse **C** nutzt, heißt **Clientklasse** von **C**.
- Klassendefinition: `PUBLIC CLASS C` ermöglicht Benutzung aller `PUBLIC` Elemente der Klasse durch Clienten.

Merke 1.2.2: Modul

Ein **Modul** wird durch den Java-Code einer *.java Datei definiert.
Paradigma der modularen Programmierung:

- Aufteilung in übersichtliche Einheiten.
- implementiere, dokumentiere und teste die Module **einzeln!**
- implementiere EV(S)A

Idee: Zusammenfassung thematisch ähnlicher Methoden

- Dokumentation in einer **API**.
- **PUBLIC CLASS** und **public** Methoden.
- **Testclient**: MAIN Methode im Modul, die den gesamten Code ausführt und daten-gesteuerte Tests ermöglicht.
- Nutzerbibliotheken sind keine eigenständige Applikationen!

2 Strukturierte Daten

2.1 Weiteres zu Feldern

Merke 2.1.1: Heap

Speicherbereich (dynamischer Speicher), in welchem zur Laufzeit eines Programms Speicherabschnitte angefordert werden können.

Hier werden beispielsweise die **Feldinhalte** eines Feldes abgespeichert.

Merke 2.1.2: Abspeicherung von Feldern

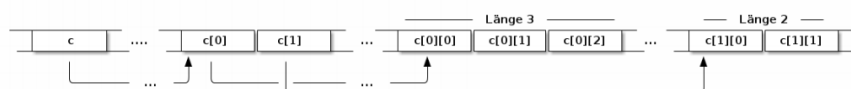
Die Feldlänge entsteht unter Umständen erst zur Laufzeit. Dementsprechend wird zunächst eine **Referenz** (=Adresse) (`NEW INT[N]`) auf das neu angelegte Feld in einer Variable abgelegt.

Änderungen?

- Feldinhalt: Ja!
- Länge: Nein!
- Referenz: Ja!

Beispiel:

```
int[][] c = new int[2][];
c[0] = new int[3];
c[1] = new int[2];
```



Merke 2.1.3: Call-by-value-Prinzip

Übergabe der Parameter an Funktionen mithilfe einer Kopie. Änderungen wirken sich **nicht** auf den ursprünglichen Wert aus!

Beispiel: Bei der Übergabe eines Feldes wird die Anfangsadresse in die Formalparameter kopiert.

Merke 2.1.4: Sortierung

Zwei Einfache Sortiervverfahren:

- **Selection-Sort:** Minima nach vorn tauschen:

```
public static void sort(double[] f) { // Selectionsort
    final int N = f.length;
    for (int i = 0; i < N-1; i++) {
        int m = i;
        for (int j = i+1; j < N; j++) // Index m des Minimums von ..
            if ( f[j] < f[m])          // .. [f[i],f[i+1],...,f[N-1]] ..
                m = j;                // .. bestimmen und nach Position ..
        double tmp=f[m]; f[m]=f[i]; f[i]=tmp; // .. i tauschen
    }
}
```

- **Insertion-Sort:** In Anfangsabschnitte sortiert einfügen.

```
public static void sort(double[] f) { // Insertionsort
    final int N = f.length;
    for (int i = 1; i < N; i++) { // f[0], ..., f[i-1] ist bereits sortiert
        // f[i] an richtiger Stelle einfüegen, d.h soweit noetig ..
        for (int j = i; j > 0 && f[j] < f[j-1]; j--) { // f[i] nach links ..
            double tmp=f[j]; f[j]=f[j-1]; f[j-1]=tmp; // .. "durchtauschen".
        }
    }
}
```

2.2 Verbunddaten

Wichtig 2.2.1: Verbund

Ein **Verbund** (Synonym: Datensatz) ist ein Datenobjekt, das eine Anzahl von Komponenten beliebigen Typs zusammenfasst.

Beispiel: **Studenten:** [name, matrikelnummer, fach]

Merke 2.2.1: Programmierung von Verbund-Daten

- Bauplan wird durch eine Klasse beschrieben.

- Ein Datenobjekt heißt **Instanz**.
- Der aktuelle Zustand einer Instanz wird in den **Instanzvariablen** gespeichert.

```
public class Student{
    String name; // Instanzvariablen ..
    int matrikelnr; // .. (auch Instanz- ..
    String fach; // .. attribute genannt )
    ...
}
```

- Konstruktionsaufruf und Zugriff:

```
Student s;
s = new Student("Fritze Bollmann", 20091234, "Biologie");
...
System.out.println(s.name); // Ausgabe: Fritze Bollmann
```

Speicherbelegung vor dem Konstruktoraufruf



Speicherbelegung nach dem Konstruktoraufruf



Merke 2.2.2: this

Das Schlüsselwort **this** speichert eine Referenz auf das aktuelle Datenobjekt.

```
public Student(String name, int matrikelnr, String fach) {
    this.name = name; // Formalparameter verdecken ..
    this.matrikelnr = matrikelnr; // .. gleichnamige Attribute!
    this.fach = fach; // this macht sie wieder unterscheidbar.
}
```

Merke 2.2.3: Zugriffsmodifizierer

Schlüsselwörter **private** und **public** steuern die Sichtbarkeit von Klassenelementen vom Clientcode aus gesehen.

- **public-Elemente:** sind von jeder anderen Klasse im Klassenpfad sichtbar.

- private-Elemente: sind außerhalb der definierenden Klasse nicht sichtbar.

Merke 2.2.4: Hash-Codes

Drucken von Verbunddaten / Arrays: Ausgabe eines Identifikators: Klassenname zusammen mit einem hexadezimalen INT-Wert = **Hashcode** des Datenobjekts (**Speicheradresse!**).

2.3 Verkettete Datenstrukturen

Merke 2.3.1: Statische Datenstrukturen

Elementare und Feld- oder Verbunddaten haben veränderliche Werte aber Struktur und Größe sind **statisch**, d.h. bleiben über die gesamte Laufzeit gleich:

- `int x = 42;` - 4 aufeinanderfolgende Bytes
- `int[] f = new int[n];` - aufeinanderfolgende `int`- Speicherstellen
- `Person p = new Person(..);` - Verbund verschiedener Komponenten

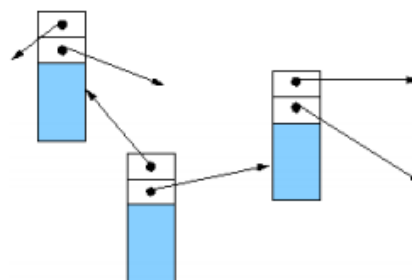
Merke 2.3.2: Dynamische Datenstrukturen

bestehen aus gleichartigen statischen Datenobjekten (**Knoten**) veränderlicher Anzahl und Beziehungsstruktur:

- Listen veränderlicher Länge
- baumartige Strukturen (linker/rechter Nachfolger...)
- Netzwerke beliebiger Nachbarschaften

Genereller Ansatz

Knoten sind Verbunde aus „Nutzdaten“ von einem Grundtyp T und Referenzen, die die Verkettung angeben.



Merke 2.3.3: Eine einfache Kette

Wir benötigen eine **dynamische Datenstruktur**:
Knotentyp und Verkettungsstruktur:

```

class ZahlKnoten {
    double zahl;
    ZahlKnoten next;
}
  
```



```

public ZahlKnoten (double zahl, ZahlKnoten next){
    this.zahl = zahl;
    this.next = next;
}
}

```

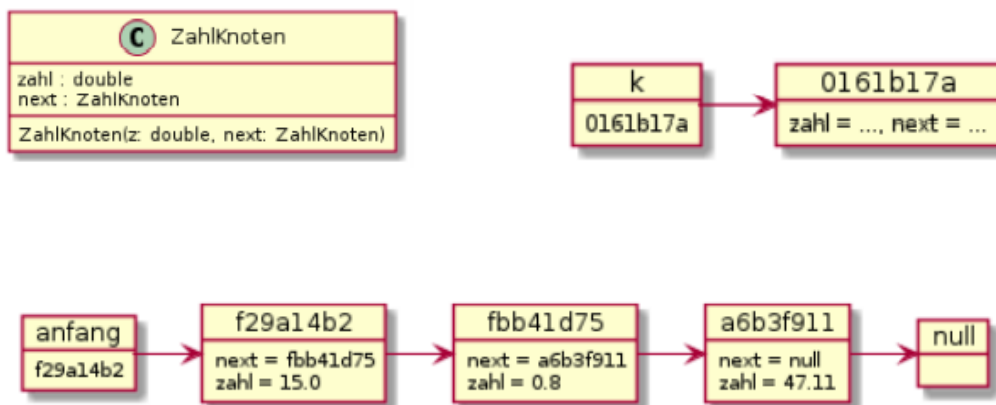
Aufbau der Liste:

```

public class Zahlpeicher {
    private static ZahlKnoten anfang = null;
    public static void speichere(double zahl){
        anfang = new ZahlKnoten(zahl, anfang);
    }
    public static void main(String[] args){
        while (!StdIn.isEmpty()){
            speichere (StdIn.readDouble()); //Einlesen 47.11, 0.8, 15.0,..
        }

        ZahlKnoten iter = anfang;
        while (iter != null){
            System.out.println(iter.zahl);
            iter = iter.next;
        }
    }
}

```

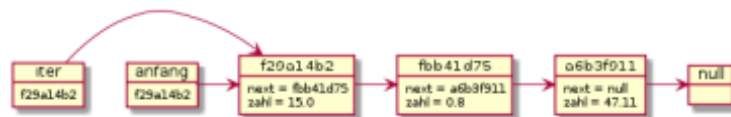


2.4 Speicherorganisation

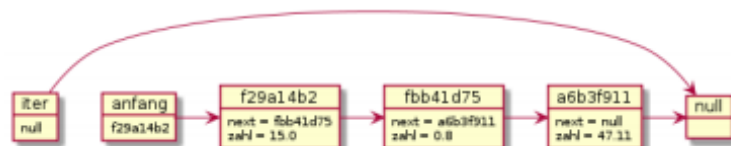
Merke 2.4.1: Abläufe, die die Speicherung beeinflussen

- Compilezeit: Compiler legt anhand der Deklarationen relative Adressen fest.
- Ladezeit: statischer Code wird in den Hauptspeicher geladen (z.B. Zuordnung absoluter Speicheradressen an statische Variablen)
- Laufzeit:

Initialisierung:



Erreichen der Abbruchbedingung:



- statischer Code bleibt im Speicher
- bei jedem Methodenaufruf wird im Laufzeitstapel Platz für lokale Variablen angelegt und initialisiert.
- bei jedem Konstruktoraufruf wird im **Heap** Speicher für erzeugte Datenobjekte angelegt und initialisiert.

Merke 2.4.2: Lebenszeit eines Bezeichners

Zeit, in der er physikalischen Speicher belegt (innerhalb der Laufzeit)

Merke 2.4.3: Speicherbereiche

- **Method area:** Speicherung des Bytecodes benötigter Klassen (inkl. ihrer statischen Variablen); *Lebenszeit: Laufzeit der Klassen*
- **Stack:** Speicherung von Aktivierungssätzen von Methoden und Konstruktoren, d.h. lokale Variablen und Rücksprungadressen; *Lebenszeit: Ausführungszeit der Methode /des Konstruktors, Speicherfreigabe wenn unbenötigt!*
- **Heap:** Speicherung von per Konstruktor angelegten Datensätze; *Lebenszeit: Solange, wie das Datenobjekt durch Referenzkette von MA / Stack erreichbar ist.*

Merke 2.4.4: Garbage-Collection

Verweiste Objekte, die Speicher belegen, aber auf die nicht mehr zugegriffen werden kann werden durch einen **Garbage-Collector** überwacht und gegebenenfalls Speicher freigegeben.

3 Objektorientierte Programmierung

3.1 Klassentypen verwenden

Merke 3.1.1: Referenzdatentypen

Zugriff erfolgt anhand von Referenzen. Man spricht auch von **Klassentyp**.
Beispiele: Felder und Verbunde, Strings (Speicherung im Heap).

Merke 3.1.2: Klassentypen

- Standardklassen: Klassentypen, die (so wie **String**) zur Hava-API gehören.
- Nutzerklassen: Klassentypen aus fremden Quellen oder selbst implementierte Klassentypen.

Merke 3.1.3: Instanzen

Jedes Objekt ist eine **Instanz** eines Datums des jeweiligen Typs.

Vergleich

	Instanzmethode	statische Methode
Beispielaufruf	<code>c.potentialAt(x,y)</code>	<code>Math.sqrt(2.0)</code>
Aufruf mit	Objektname	Klassenname
Parameter	Referenz auf Objekt und Argument/e	Argument/e
Zweck	z.B. Objektwerte liefern	Ergebniswert liefern

Merke 3.1.4: Unterschied zu Verbunddatentypen

Wir haben nun lebendige Objekte. Sie haben einen **Zustand**, beschrieben durch den aktuellen Wert und ein **Verhalten**, beschrieben durch die Instanzmethoden.