

RBs: 11. RBs: 11.-15.2. → Listen im Schr. MASOM

Voraussichtlich: Tausend von Übungen und Vorlesungen in den kommenden Wochen.

Erinnerung: Eine rekursive Funktion wird durch sich selbst definiert

Bsp: Fakultät $n!$ wird definiert über

$$n! = \begin{cases} 1 & \text{falls } n=0 \\ n \cdot (n-1)! & \text{falls } n \in \mathbb{N} \setminus \{0\} \end{cases}$$

Beispiele rekursiver Funktionen mit:

- extremen Speicherplatzbedarf
- Terminierung nicht garantiert

Ackermann-Funktion: Für $m, n \in \mathbb{N} \cup \{0\}$ sei

$$a(m, n) := \begin{cases} m+1, & m=0 \\ a(m-1, 1), & m>0, n=0 \\ a(m-1, a(m, n-1)), & m>0, n>0 \end{cases}$$

Theorie der Berechenbarkeit: Ackermann-Funktion wächst stärker als jede primitive rekursive Funktion!

Java-Implementation:

```
int a(int m, int n) { // m, n ≥ 0
```

```

    if (m == 0) return m+1;
    if (m == 0) return a(m-1, 1);
    return a(m-1, a(m, m-1));
}

```

Bsp. Berechnung von $a(1, 3)$ "per Hand":

$$\begin{array}{l}
 a(1, 3) = a(0, \underbrace{a(1, 2)}) \\
 \vdots \\
 \underbrace{a(0, \underbrace{a(1, 1)})} \\
 \quad \underbrace{a(0, \underbrace{a(1, 0)})} \\
 \quad \quad \underbrace{a(0, \underbrace{a(0, 1)})} \\
 \quad \quad \quad 2 \\
 \quad \quad \quad \underbrace{a(0, 2)} \\
 \quad \quad \quad \quad 3 \\
 \quad \quad \quad \underbrace{a(0, 3)} \\
 \quad \quad \quad \quad 4 \\
 a(0, 4) = 5
 \end{array}$$

Rekursionstiefe wird zu groß,
 Laufzeitstack läuft über!

Satz: Aufruf $a(m, n)$ terminiert für
 alle $m, n \in \mathbb{N} \cup \{0\}$ nach endlich vielen
 Schritten.

Beweis: Zwei ineinander geschachtelte

Induktionen:

- äußere Induktion mache m
- innere Induktion mache n bei festem m

Ind. Anf. $m=0 \Rightarrow a(0, n) = n+1 \Rightarrow$ nur ein Aufruf
 \Rightarrow Terminierung

Ind. Vor.: Terminierung für alle $0 \leq k < m$ (*)
und für alle n

Schluss auf m :

Ind. Anfang: $n=0$

$a(m, 0) \stackrel{\text{Def.}}{=} a(m-1, 1)$ terminiert nach (*)

Ind. Vor.: $a(m, k)$ terminiert für alle $0 \leq k < m$ (**)

Schluss auf n :

$$\begin{aligned} a(m, n) &\stackrel{\text{Def.}}{=} a(m-1, \underbrace{a(m, n-1)}_{\substack{\text{terminiert wegen (**),} \\ \text{ergibt } r}}) \\ &= \underbrace{a(m-1, r)}_{\text{terminiert wegen (*)}} \end{aligned}$$

□

□

Folgerung: Ackermann-Funktion terminiert zwar immer, die Rekursionstiefe wird aber schnell extrem groß. Berechnung scheitert dann in der Praxis.

Ulam-Funktion erzeugt ausgehend vom Startwert $a_0 \in \mathbb{N}$ eine Folge

$a_0, a_1, a_2, \dots, a_m$ mit

$$a_{m+1} := \begin{cases} a_m / 2 & \text{falls } a_m \text{ gerade} \\ 3 \cdot a_m + 1 & \text{falls } a_m \text{ ungerade} \end{cases}$$

Folge endet, sobald $a_m = 1$ für ein m .

Beispiel:

$$a_0 = 3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

Offenes Problem, ob die Folge für jede Eingabe abbricht (terminiert),

d.h. ob die Länge der Folge immer endlich ist.

Rekursive Java-Methode zur Berechnung der Länge:

```
long ulamLength(long n) {  
    if (n == 1) return 0;  
    if (n % 2 == 0) return (ulamLength(n/2) + 1);  
    return (ulamLength(3*n+1) + 1);  
}
```

basiert auf rek. Def. der Länge $l(n) := \begin{cases} 1 & \text{falls } n=1 \\ 1+l(n/2) & \text{falls } n \text{ gerade} \\ 1+l(3n+1) & \text{sonst} \end{cases}$

Tipp: erst mathem. Rekursionsformel überlegen, und dann diese 1-1 in Java Methode übersetzen.

Wann sollte man Rekursion wählen bzw. vermeiden?

- hängt von Größe / Tiefe des Rekursionsbaums ab!
- iterative Methoden meist besser, falls offensichtlich existieren.

§ 9. Mehr zu Klassen:

Vererbung, Interfaces, Modifizierer,
Packages, Klassen in Klassen

Vererbung

neu: eine bestehende Klasse erweitern

Beispiel: Quadrate und Rechtecke

```
public class Square {  
    protected double width;  
  
    public Square() { // default constructor  
        this.width = 1;  
    }  
  
    public Square(double width) {  
        this.width = width;  
    }  
  
    public double area() {  
        return this.width * this.width;  
    }  
}
```

≙ Einheitsquadrat

Klasse „Rectangle“ erweitert Klasse „Square“:

```

public class Rectangle extends Square {

    private double height;

    public Rectangle() { // default constructor
        this.height = 1; // calls super() implicitly!
    } super(); überflüssig, macht Compiler automatisch

    public Rectangle(double width, double height) {
        super(width); // calls Square(width)
        this.height = height;
    }

    public double area() { // overwrite area() of class Square
        return this.width * this.height;
    }
}

```

- Klasse Rectangle ist Unterklasse / Tochterklasse / Sohnklasse
- Klasse Square ist Oberklasse / Mutterklasse / Vaterklasse
- Klasse Rectangle „erbt“ von Klasse Square.
- Tochterklassen haben einiges zusätzlich, was die Mutterklasse nicht besitzt, z.B. height
- Square besitzt bereits Methode area(), diese wird in Rectangle überschrieben.
- Tochterklasse „erbt“, d.h. sie hat Zugriff auf alle nicht privaten Felder und Methoden der Mutterklasse (bis auf Konstruktoren)
- Tochterklasse darf Methoden und Felder der Mutterklasse neu definieren (überschreiben)
z.B. area()

- Tochterklasse darf zusätzliche Methoden und Felder einführen.
- Zugriff auf Felder und Methoden der Mutterklasse geschieht mit Referenz `super`.
- `super.super` geht nicht!

Besondere Regeln für Konstruktor:

Konstruktoren

- werden nicht mit vererbt
- Konstruktor der Sohnklasse muss einen Konstruktor der Vaterklasse aufrufen, dies geschieht mit
 - explizitem Aufruf `super(...)` oder
 - implizitem Aufruf von `super()`, d.h. Defaultkonstruktor der Vaterklasse wird aufgerufen
 - => Compilerfehler, wenn Vaterklasse keinen Defaultkonstruktor hat
- Aufruf von Konstruktor der Oberklasse muss erste Anweisung im Konstruktor der Sohnklasse sein

Die Ur-Ur-Ur-...-Oma aller Klassen ist Object

- Klasse Object ist der Ur-Ur-...-Großvater aller Klassen
- Object ist default Oberklasse jeder Klasse
`public class MyClass {...}` ist dasselbe wie
`public class MyClass extends Object {...}`
- Klasse Object hat die Methode `public String toString()`
 Damit hat auch jede Unterklasse diese Methode, sie muss aber i.A. überschrieben werden
- Mehrfachvererbung ist in Java nicht möglich, eine Klasse erbt immer nur von einer Klasse

↑
 Daher bildet die Klassenhierarchie einen Baum.

```
public class Test {
```

```
    public static void main (String[] args) {  
        Square[] vec = new Square[10];
```

```
        // initialize this array with squares and rectangles  
        for (int i = 0; i < vec.length; i++) {  
            if (i % 2 == 0) vec[i] = new Square(i);  
            else vec[i] = new Rectangle(i, i+1);  
        }
```

```
        // write type and area to console  
        for (int i = 0; i < vec.length; i++) {  
            System.out.println("i = " + i + ": "  
                + vec[i].getClass().getName()  
                + " mit Flaeche "  
                + vec[i].area());  
        }
```

```
    }  
}
```

↖ funktioniert, da jedes
Rectangle auch ein
Square ist

↖ Klassenname

↑
Jedes Objekt weiß kennt seine Klasse
und ruft zugehörige area()-Methode auf.

↗
sehr komfortabel für Datenstrukturen, die Objekte
verschiedener Klassen einer Hierarchie verwalten.

Beispiel 2:

Abstrakte Klasse für Uni-Angestellte.

Jeder Angestellte hat Eigenschaften, die er/sie
mit allen anderen teilt, aber auch spezif. Eigenschaft.

↑

↑

im Mutterklasse

in verschiedene
Tochterklassen

```
public abstract class Employee {  
    // abstract class (since there is at least one abstract method)  
    // abstract classes cannot have objects  
  
    private String name;  
  
    protected double salary;  
    /* protected is almost like private, except that classes derived from  
    * class have access to protected variables and methods */  
  
    public Employee() {  
        // default constructor, doesn't do anything  
        // still useful since we want to derive classes from this class  
    }  
  
    public Employee(String name) {  
        this.name = name;  
    }  
  
    protected double getSalary() { // protected since salary is confidential  
        return this.salary;  
    }  
  
    public String getName() {  
        return this.name;  
    }  
  
    public abstract void doWork();  
    // no definition, only declaration and therefore abstract  
    // implementation depends on type of employee  
}
```

abstrakte Klasse, d.h. keine Objekte dieser Klasse, nur Nachfahren.

abstrakte Methode, hat keinen Körper, nur Deklaration
→ Bauanleitung für Tochterklassen.

Zwei konkrete Tochterklassen,

```
public class Lecturer extends Employee {  
  
    public Lecturer() {  
        // calls default constructor of superclass Employee implicitly  
    }  
  
    public Lecturer(String name) {  
        super(name); // calls Employee(name)  
    }  
  
    public void doWork() { // overwrites doWork() in Employee  
        System.out.println("Halte die Vorlesung");  
    }  
}
```

```

public class TA extends Employee {

    public TA() {
        // calls default constructor of superclass Employee implicitly
    }

    public TA(String name) {
        super(name); // calls Employee(name)
    }

    public void doWork() { // overwrites doWork() in Employee
        System.out.println("Erstelle Übungszettel");
    }

    public double getSalary() { // more public than in Employee
        return this.salary; // more private is not possible
    }
}

```

Verwendung dieser Klassen:

```

public class Test {

    public static void main (String[] args) {
        Lecturer peis = new Lecturer("Peis");
        TA martin = new TA("Gross");
        doSomething(peis);
        doSomething(martin);
    }

    static void doSomething(Employee e) {
        // ...
        e.doWork();
    }
}

```

Output:
 Halte die Vorlesung
 Erstelle Übungszettel

- Überall, wo Mutterklasse stehen kann, kann auch Tochterklasse stehen
- Alles, was man mit Objekt vom Typ Employee machen kann, kann man auch mit Lecturer oder TA machen.

- Referenz martin zeigt auf Objekt vom Typ TA, die Methode doSomething(Employee e) ist zwar allg. für Employee definiert, aber beim Aufruf für martin wird doWork() der Klasse TA aufgerufen.