



## **Vorlesung an der Berufsakademie Oldenburg**

### **Unterrichtseinheit 15 Funktionen und Parameter**

**Dr. Dietrich Boles**

- Prozeduren
  - Motivation
  - Prozedurdefinition
  - Prozeduraufruf
- Funktionen
  - Motivation
  - Funktionsdefinition
  - Funktionsaufruf
- Parameter
  - Motivation
  - Parameterdefinition
  - Parameterübergabe
  - varargs
  - Zusammenfassung
- Gültigkeitsbereich und Lebensdauer von Variablen
- Globale und lokale Variablen
- Gültigkeitsbereich / Überladen von Funktionen
- Beispiele
- Aufgaben

## Definition:

- Teil eines Programmes, das eine in sich abgeschlossene Aufgabe löst.

## Vorteile:

- bessere Übersichtlichkeit von Programmen
- separate Lösung von Teilproblemen
- Platzeinsparung
- einfachere Fehlerbeseitigung
- Flexibilität
- Wiederverwendbarkeit

**Java-spezifisch:** Prozeduren sind eigentlich so genannte *Methoden*

## zwei Aspekte:

- Prozedurdefinition
- Prozeduraufruf

# Prozeduren / Definition

---

```
<Proc-Def>      ::= <Proc-Kopf> <Proc-Rumpf>
<Proc-Kopf>     ::= ["public"] "static" "void" <Proc-Name>
                  "(" [ <Param-Defs> ] ")"
<Proc-Name>     ::= <Bezeichner>
<Proc-Rumpf>    ::= <Block>
<Param-Defs>    später
```

## Semantik:

- keine Auswirkungen auf den Programmablauf
- führt neue Prozedur mit dem angegebenen Namen ein

## Ort im Programm:

- innerhalb einer Klasse
- vor/nach main-Prozedur
- keine Schachtelung von Prozedurdefinitionen möglich

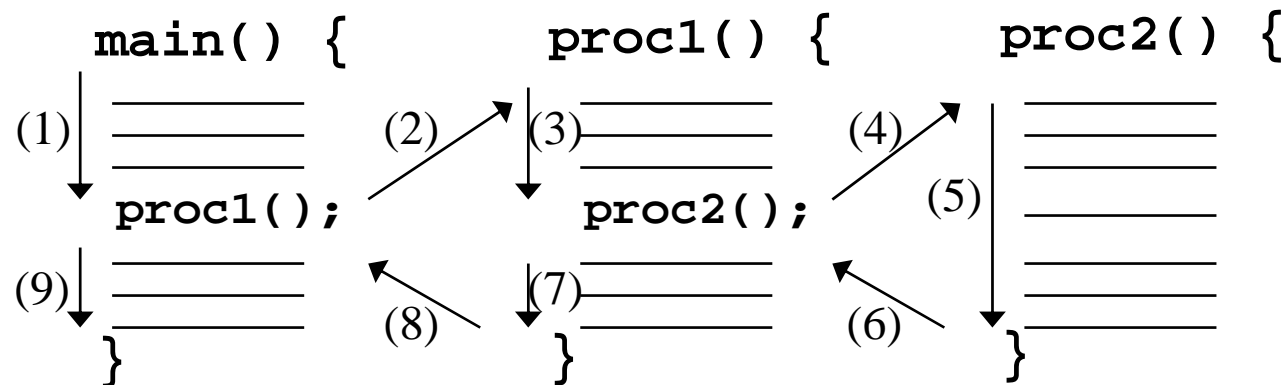
# Prozeduren / Aufruf

**<Proc-Aufruf> ::= <Proc-Name> "(" [ <Param-List> ] ")" ";"**

**<Param-List>      später**

## Semantik:

- Prozeduraufruf ist eine **Anweisung**
- beim Aufruf einer Prozedur werden die Anweisungen im Prozedurrumpf ausgeführt
  - bis der Prozedurrumpf vollständig abgearbeitet ist oder
  - bis eine return-Anweisung ausgeführt wird



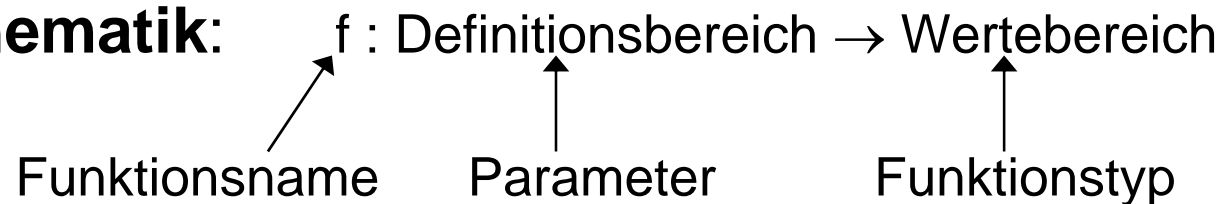
## Definition:

- Teil eines Programmes, das durch die Ausführung von Anweisungen einen Wert berechnet

## Abgrenzung:

- "Prozeduren tun etwas"
- "Funktionen berechnen und liefern einen Wert"

## Mathematik:



## Drei Aspekte:

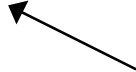
- Wertberechnung und -rückgabe
- Funktionsdefinition
- Funktionsaufruf

# Funktionen / Definition (1)

```

<Funk-Def>      ::= <Funk-Kopf> <Funk-Rumpf>
<Funk-Kopf>     ::= ["public"] "static" <Typ> <Funk-Name>
                  "(" [ <Param-Defs> ] ")"
<Funk-Name>     ::= <Bezeichner>
<Funk-Rumpf>    ::= <Block>
  
```

Funktionstyp



## Nebenbedingung:

- im Rumpf muss es **in jedem möglichen** Funktionsdurchlauf eine return-Anweisung geben, wobei der Typ des return-Ausdrucks konform zum Funktionstyp ist

## Semantik:

- keine Auswirkungen auf den Programmablauf
- führt neue Funktion mit dem angegebenen Namen und Funktionstyp ein

## Ort im Programm:

- innerhalb einer Klasse
- vor/nach main-Prozedur
- keine Schachtelung von Funktionsdefinitionen möglich

## Beispiel:

```
class FunkProbe {  
  
    static int funkEins() {  
        IO.println("in funkEins");  
        return 1;  
    }  
  
    public static void main(String[] args) {  
        ...  
    }  
  
    static char funkZwei() {  
        if ( IO.readInt() == 0 ) return 'a';  
        IO.println("in funkZwei");  
        return 'b';  
    }  
}
```



## Fehlerhafte Beispiele:

```
class FunkProbe2 {
    static int liefereWert() {
        if ( IO.readInt() == 0 ) return -2;
        // Fehler: im else-Fall wird kein return ausgeführt!
    }
    static double funkDrei() {
        if (IO.readInt() == 0)
            return 'a';          // ok: impliziter Typcast!
        IO.println("in funkDrei");
        return 'a' == 'b';      // Fehler: ungültiger Typ!
    }
    static boolean test() {
        return 2 == 0;
        int wert = 2; // Fehler: Anweisung wird nicht erreicht!
    } }
```

`<Funk-Aufruf> ::= <Funk-Name> "(" [ <Param-List> ] ")"`

## ➤ Semantik:

- Funktionsaufruf ist ein **Ausdruck**
- beim Aufruf einer Funktion werden die Anweisungen im Funktionsrumpf ausgeführt, bis eine return-Anweisung ausgeführt wird
- nach der Berechnung des Wertes des return-Ausdrucks der return-Anweisung wird die Funktion verlassen und der Wert als Funktionswert zurückgeliefert

## Beispiel:

```
class FunkProbe {  
    static int funk() {  
        int zahl = IO.readInt("Zahl:");  
        return 2*zahl;  
    }  
    public static void main(String[] args) {  
        int wert = funk() * 4 * funk();  
    }  
}
```

# Parameter / Motivation (1)

- bisher sind Prozeduren und Funktionen sehr unflexibel
- Parameter erhöhen die Flexibilität
- Beispiel:

$$\binom{n}{m} = \frac{n!}{m! (n-m)!} \quad \text{für } 0 \leq m \leq n$$

$$\binom{6}{3} = \frac{6!}{3! (6-3)!}$$

- benötigt wird

```
static int fak6() {...}
static int fak3() {...}
```

- gewünscht:

```
static int fakn() {...}
```

wobei der Wert n erst zur Laufzeit angegeben werden muss

## Parameter / Motivation (2)

```
class ParameterMotivation {
    static int fak3() {
        int zahl = 3; int erg = 1;
        for(int zaehler=2; zaehler<=zahl; zaehler++)
            erg = erg * zaehler;
        return erg;
    }
    static int fak6() {
        int zahl = 6; int erg = 1;
        for(int zaehler=2; zaehler<=zahl; zaehler++)
            erg = erg * zaehler;
        return erg;
    }
    public static void main(String[] args) {
        int ueber_6_3 = fak6() / (fak3() * fak3());
    } }
```

## Parameter / Motivation (3)

---

```
class ParameterMotivation {  
  
    static int fak(int zahl) {  
        int erg = 1;  
        for(int zaehler=2; zaehler<=zahl; zaehler++)  
            erg = erg * zaehler;  
        return erg;  
    }  
  
    public static void main(String[] args) {  
        int ueber_6_3 = fak(6) / (fak(3) * fak(3));  
    }  
}
```

## Parameter / Motivation (4)

```
class ParameterMotivation {
    static int fak(int zahl) {
        int erg = 1;
        for(int zaehler=2; zaehler<=zahl; zaehler++)
            erg = erg * zaehler;
        return erg;
    }
    static int ueber(int n, int m) {
        if ((0 <= m) && (m <= n))
            return fak(n) / (fak(m) * fak(n-m));
        return -1; // Fehlerfall (spaeter Exceptions)
    }
    public static void main(String[] args) {
        int ueber_6_3 = ueber(6, 3);
        int ueber_7_4 = ueber(7, 4);
    } }
```

```
<Param-Defs> ::= <Typ> <Param-Name>  
                { " ," <Typ> <Param-Name> }  
<Param-Name> ::= <Bezeichner>
```

## Semantik:

- Einführung einer lokalen Variable für die Prozedur/Funktion
- Bezeichnung: "formaler Parameter"

## Beispiele:

```
static int summe(int op1, int op2) {  
    return op1 + op2;  
}  
static void ausgabe(char zeichen, int anzahl) {  
    for (int i=0; i<anzahl; i++)  
        IO.println(zeichen);  
}
```

# Parameter / Übergabe (1)

`<Param-List> ::= <Ausdruck> { ", " <Ausdruck> }`

## Semantik:

- Übergabe eines "Initialwertes" für den formalen Parameter
- Bezeichnung: aktueller Parameter"

## Bedingungen:

- Anzahl an formalen Parametern = Anzahl an aktuellen Parametern
- für alle Parameter in der entsprechenden Reihenfolge: Typ des aktuellen Parameters typkonform zum Typ des formalen Parameters

## Schema:

```
static int summe(int op1, int op2) {  
    // int op1 = "Wert des ersten aktuellen Parameters";  
    // int op2 = "Wert des zweiten aktuellen Parameters";  
    return op1 + op2;  
}  
  
public static void main(String[] args) {  
    int s1 = summe(2, 3*4);  
    int s2 = summe(s1, -5);  
}
```



## Parameter / Übergabe (2)

```
class Beispiel {
    static int fak(int zahl) {
        int erg = 1;
        for(int zaehler=2; zaehler<=zahl; zaehler++)
            erg = erg * zaehler;
        return erg;
    }
    static int ueber(int n, int m) {
        return fak(n) / (fak(m) * fak(n-m));
    }
    public static void main(String[] args) {
        int zahl = fak(3);
        zahl = fak(fak(2));
        zahl = ueber(fak(zahl)+1,zahl-1);
    } }
```

```
class Beispiel {  
  
    static int funk(int n1, int n2, int n3, int n4) {  
        return n1 + n2 + n3 + n4;  
    }  
  
    public static void main(String[] args) {  
        int n = 2;  
        int zahl = funk(n++, n = n+2, n--, n);  
        // entspricht hier: funk(2, 5, 5, 4)  
        // Grund: Parameterauswertung von links nach rechts  
    }  
}
```

# Parameter / varargs / Motivation

```
static int summe2(int zahl1, int zahl2) {  
    return zahl1 + zahl2;  
}
```

```
static int summe3(int zahl1, int zahl2, int zahl3) {  
    return zahl1 + zahl2 + zahl3;  
}
```

```
static int summe4(int zahl1, int zahl2, int zahl3, int zahl4) {  
    return zahl1 + zahl2 + zahl3 + zahl4;  
}
```

```
public static void main(String[] args) {  
    int s1 = summe2(3, 5);  
    int s2 = summe3(66, 5, s1);  
    s1      = summe4(67, s1, -3, s2);  
    s2      = summe(4, 7, s1, s2 - 2, 88);    // Fehler  
}
```

```
static int summe(int... zahlen) { ← varargs-Parameter
    int ergebnis = 0;
    for (int n=0; n < zahlen.length; n++) {
        ergebnis += zahlen[n] ;
    }
    return ergebnis;
}
```

Anzahl aktueller Parameter

n-ter aktueller Parameter (ab 0)

```
public static void main(String[] args) {
    int s1 = summe(3, 5);
    int s2 = summe(66, 5, s1);
    s2      = summe(1, 2, s1, 4, 5, 6, s2, 8, 9, 10);
}
```

- Seit Java 5.0
- Eine Funktion darf maximal einen varargs-Parameter besitzen
- Dieser muss an letzter Stelle der formalen Parameterliste stehen

# Parameter / Zusammenfassung

---

- Parameter sind spezielle funktionslokale Variablen
- Formale Parameter werden zur Laufzeit mit aktuellen Parameter(werten) initialisiert
- in Java:
  - nur Werteparameter (call-by-value)
  - die Typen der Parameter müssen bei der Funktionsdefinition festgelegt werden (keine flexiblen Parametertypen)
  - als Parametertypen können (bisher nur) Standarddatentypen verwendet werden (später auch Klassentypen)
  - es ist nicht möglich, Typen von Funktionen zu definieren
  - der Funktionstyp muss bei der Definition angegeben werden (keine flexiblen Funktionstypen)
  - Funktionen können nur einen einfachen Wert liefern (insbesondere kein Kreuzprodukt von Werten)
  - aktuelle Funktionsparameter werden von links nach rechts ausgewertet

## ➤ **Gültigkeitsbereich ("Scope"):**

- der Gültigkeitsbereich einer Variablen ist zur Compilezeit relevant
- der Gültigkeitsbereich eines Variablennamens ist der Block, in dem die Variable definiert wird, sowie aller inneren Blöcke, und zwar **nach** der Stelle seiner Definition
- Variablennamen müssen innerhalb eines Blocks und aller inneren Blöcke eindeutig sein
- Variablennamen sind nur innerhalb ihres Gültigkeitsbereichs anwendbar

## ➤ **Lebensdauer:**

- die Lebensdauer einer Variablen ist zur Laufzeit relevant
- die Lebensdauer einer Variablen beginnt bei der Ausführung der Variablendefinition (→ Reservierung von Speicherplatz)
- die Lebensdauer einer Variablen endet nach der vollständigen Abarbeitung des Blockes, in dem sie definiert wird, bzw. nach dem Verlassen des Blockes mittels einer return-Anweisung (→ Freigabe von Speicherplatz)
- die beiden Definitionen werden später noch erweitert

## Gültigkeitsbereich / Lebensdauer von Variablen (2)

```
class GueltigeBeispiele {
    static void p(int x) {
        int i = x, y = 2; ...
    }
    static void q(int x) {
        float y = 2, w = 0; ...
    }
    public static void main(String[] args) {
        int i = 3, j = i;
        {
            ...
            int x = i;
            p(i);
            ...
        }
        j = i;
        {
            int w = 0, x = j;
            ...
            w = 4;
            ...
        }
    }
}
```

## Gültigkeitsbereich / Lebensdauer von Variablen (3)

```
class UngueltigeBeispiele {
    static void P(int x) {
        int i = z; // Fehler: z nicht gültig
    }
    static void Q(int x) {
        float x = 2.0F; // Fehler: x doppelt definiert
        int j = i; // Fehler: i nicht gültig
    }
    public static void main(String[] args) {
        int i = 3 * i; // Fehler: i noch nicht definiert
        {
            ...
            int v = i;
            ...
        }
        i = v; // Fehler: v nicht mehr gültig
    } }
```



# Gültigkeitsbereich / Lebensdauer von Variablen (4)

```
class UngueltigeBeispiele2 {
    static int eingabe(int min, int max) {
        int zahl = IO.readInt ("Zahl eingeben:");
        while ((zahl < min) || (zahl > max)) {
            IO.println("Fehler!");
            zahl = IO.readInt("Zahl eingeben:");
        }
        return zahl;
    }

    static int eingabeF(int min, int max) {
        int zahl = IO.readInt ("Zahl eingeben:");
        while ((zahl < min) || (zahl > max)) {
            IO.println("Fehler!");
            int zahl = IO.readInt("Zahl eingeben:"); // Fehler!
        }
        return zahl;
    }
}
```

# Globale und lokale Variablen

```
class Variablen {
```

```
    static int i;
```

globale Variable

```
    static void f1() {  
        IO.println(i);  
    }
```

```
    static void f2() {  
        i++;  
    }
```

```
    static void f3(int i) {  
        int j = 3;  
        IO.println(i+j);  
    }
```

lokale Variable

```
    public static void main(String[] args) {  
        i = 4711;  
        f1();  
        f2();  
        f3(i);  
    } }
```

## ➤ **Gültigkeitsbereich ("Scope"):**

- der Gültigkeitsbereich eines Funktionsnamens ist die gesamte Klasse, in der die Funktion definiert wird
- Funktionsnamen müssen innerhalb eines Blocks und aller inneren Blöcke eindeutig sein; Ausnahme: Überladen von Funktionen!
- Funktionsnamen sind nur innerhalb ihres Gültigkeitsbereichs anwendbar

## ➤ **Überladen von Funktionen:**

- zwei oder mehrere Funktionen können innerhalb eines Gültigkeitsbereichs denselben Namen besitzen, wenn
  - sie eine unterschiedliche Anzahl an Parametern besitzen oder
  - wenn sich die Parametertypen an entsprechender Stelle unterscheiden
- die Definition wird später noch erweitert

# Gültigkeitsbereich / Überladen von Funktionen (2)

```
class Beispiel {
    static float summe(float op1, float op2) {
        return op1 + op2;
    }
    static int summe(int op1, int op2) {
        return op1 + op2;
    }
    static float summe(int op1, int op2) // Fehler!

    public static void main(String[] args) {
        int s1      = summe(2, 3);          // int-summe
        float s2     = summe(2.0F, 3.0F);    // float-summe
        float s3     = summe(2, 3);          // int-summe
        float s4     = summe(2.0F, 3);       // float-summe
    }
}
```

# Beispiel 1

```
class Dual {
    static int dual(int dezZahl) {
        if (dezZahl < 0) return -1; // Fehlerfall
        int dualZahl = 0;
        int faktor = 1;
        while (dezZahl > 0) {
            dualZahl += dezZahl % 2 * faktor;
            dezZahl /= 2;
            faktor *= 10;
        }
        return dualZahl;
    }
    public static void main(String[] args) {
        int zahl = IO.readInt ("Dezimalzahl:");
        IO.println(dual(zahl));
    }
}
```

## Beispiel 2

```
class Umrechnung {

    static int umrechnung(int dezZahl, int nach) {
        if ((dezZahl < 0) || (nach < 2) || (nach > 10))
            return -1; // Fehlerfall
        int zahl = 0;
        int faktor = 1;
        while (dezZahl > 0) {
            zahl += dezZahl % nach * faktor;
            dezZahl /= nach;
            faktor *= 10;
        }
        return zahl;
    }

    public static void main(String[] args) {
        int zahl = IO.readInt ("Dezimalzahl:");
        int system = IO.readInt ("System:");
        IO.println(umrechnung(zahl, system));
    }
}
```

## Beispiel 3

```
class Reverse {

    static int abs(int zahl) {
        return (zahl < 0) ? -zahl : zahl;
    }

    static int reverse(int zahl) {
        zahl = abs(zahl);
        int ergebnis = 0;
        while (zahl > 0) {
            ergebnis = ergebnis * 10 + zahl % 10;
            zahl /= 10;
        }
        return ergebnis;
    }

    public static void main(String[] args) {
        int zahl = IO.readInt ("Zahl eingeben:");
        IO.println(reverse(zahl));
    }
}
```

## Beispiel 4

```
class Check {  
  
    static boolean isDigit(char zeichen) {  
        return (zeichen >= '0') && (zeichen <= '9');  
    }  
  
    static int getValue(char zeichen) {  
        if (isDigit(zeichen))  
            return zeichen - '0';  
        else  
            return -1; // Fehlerfall  
    }  
  
    public static void main(String[] args) {  
        char zeichen = IO.readChar ("Zeichen eingeben:");  
        IO.println(getValue(zeichen));  
    }  
}
```



## Beispiel 5 (1)

Schreiben Sie ein Programm "*Rechteck*", das zunächst eine Zahl *hoehe* einliest, die größer als 1 ist, dann eine Zahl *breite* einliest, die größer als 1 und kleiner als 10 ist, und anschließend ein "Rechteck" mit der Höhe *hoehe* und der Breite *breite* in folgender Gestalt auf den Bildschirm ausgibt:

Beispiel:

```
$ java Rechteck
Hoehe eingeben:
4<CR>
Breite eingeben:
5<CR>
+----+
|    |
|    |
|    |
+----+
$
```

## Beispielprogramm 5 (2)

```
class Rechteck {

    public static void main(String[] a) {
        int hoehe = eingabe(2);
        int breite = eingabe(2, 9);
        zeilenausgabe('+', '-', breite);
        for (int j=2; j<hoehe; j++) {
            zeilenausgabe('|', ' ', breite);
        }
        zeilenausgabe('+', '-', breite);
    }

    static int eingabe(int min) {
        int zahl = IO.readInt ("Zahl eingeben:");
        while (zahl < min) {
            IO.println("Fehlerhafte Eingabe!");
            zahl = IO.readInt("Zahl eingeben:");
        }
        return zahl;
    }
}
```

## Beispielprogramm 5 (3)

```
static int eingabe(int min, int max) {
    int zahl = IO.readInt ("Zahl eingeben:");
    while ((zahl < min) || (zahl > max)) {
        IO.println("Fehlerhafte Eingabe!");
        zahl = IO.readInt("Zahl eingeben:");
    }
    return zahl;
}

static void ausgabe(char zeichen, int anzahl) {
    for (int i=0; i<anzahl; i++)
        IO.print(zeichen);
}

static void zeilenausgabe(char start, char mitt,
                           int anzahl) {

    ausgabe(start, 1);
    ausgabe(mitt, anzahl-2);
    ausgabe(start, 1);
    IO.println();
}
}
```

## Beispiel 6

```
class Mathematik {
    static int min(int x, int y) {
        return (x <= y) ? x : y;
    }
    static int max(int x, int y) {
        return (x >= y) ? x : y;
    }
    static int abs(int x) {
        return (x < 0) ? -x : x;
    }
    static float pow(float zahl, int pot) {
        float ergebnis = 1.0F;
        for (int i=0; i<abs(pot); i++)
            ergebnis = ergebnis * zahl;
        return (pot >= 0) ? ergebnis : 1.0F/ergebnis;
    }
    static int round(float x) {
        return (x < 0) ?
            -((int)(-x + 0.5F)) : (int)(x + 0.5F);
    }
}
```

# Aufgabe 1

Die Ulam-Zahlenfolge beginnt mit einer beliebigen natürlichen Zahl  $a$ . Ist  $a$  gerade, so erhält man die nächste Zahl  $b$  durch Halbierung. Ist  $a$  ungerade, so erhält man die nächste Zahl  $b$  durch eine Verdreifachung von  $a$  und dem Addieren von 1. Die Folge endet, wenn ein Glied der Folge den Wert 1 erreicht.

Schreiben Sie ein Java-Programm `UlamFolge`, das zunächst die Eingabe eines positiven `int`-Wertes erwartet. Anschließend soll die Ulam-Folge zu dieser Zahl auf den Bildschirm ausgegeben werden. Definieren und nutzen Sie dabei eine Funktion `ulamWert`, die zu einem als Parameter übergebenen `int`-Wert den nächsten Wert der Ulam-Folge liefert.

Beispiel:

```
$ java UlamFolge
Eingabe: 3
Ausgabe: 3 10 5 16 8 4 2 1
```

Schreiben Sie ein Programm `Sortieren`, das die Ziffern einer eingelesenen positiven Zahl sortiert nach deren Größe auf den Bildschirm ausgibt. Es dürfen keine Arrays verwendet werden!

Beispiel:

```
$ java Sortieren  
Eingabe: 37734207  
Ausgabe: 02334777
```

Hinweise: Definieren und nutzen Sie folgende Funktionen:

- `length`: Liefert Anzahl an Ziffern einer Zahl  
`length(12345) → 5`
- `kleinsteZiffer`: Liefert die kleinste Ziffer einer Zahl  
`kleinsteZiffer(342667) → 2`
- `liefereZiffer`: Liefert Ziffer einer Zahl an einer bestimmten Position  
`liefereZiffer(147335, 2) → 3`
- `loescheZiffer`: Liefert Zahl, bei der eine Ziffer einer Zahl an einer bestimmten Position gelöscht ist  
`loescheZiffer(147335, 2) → 14735`