Methoden

Adrian Schüpbach
adrian_laurent.schuepbach@alumni.ethz.ch

Worum geht es?

- Funktionen/Methoden
- Überladung
- Rekursion

Was sind Funktionen?

- Funktion hat ...
  - ... einen Namen
  - ... einen “Körper” mit Anweisungen
- Funktion kann immer wieder verwendet werden
  - Führt immer die gleichen Anweisungen aus
  - Kann somit Code vereinfachen und sparen

Was sind Funktionen?

- Funktion hat ...
  - ... einen Namen
  - ... einen “Körper” mit Anweisungen
- Funktion kann immer wieder verwendet werden
  - Führt immer die gleichen Anweisungen aus
  - Kann somit Code vereinfachen und sparen

- Funktionen bekannt aus der Mathematik
  - $f(x) = ...$
  - $sin(x)...$

Parameterlose Funktionen

- Beispiel:
  - Programm schreibt zweidimensionales Array auf den Bildschirm
  - Zur besseren Lesbarkeit soll zwischen zwei Zeilen eine Markierung ausgegeben werden
    - ```
      "**************************************"
      "******** Naechste Zeile ********"
      "**************************************"
      ```
  - Könnte jedesmal dreimal `System.out.println()` benutzen
  - Besser: Eine Funktion `zeilenTrennung()` implementieren
      ```

Parameterlose Funktionen

Funktion zeilenTrennung()

```
public class Programm14 {
  public static void zeilenTrennung() {
    System.out.println("**********************");
    System.out.println("*** Naechste Zeile ***");
    System.out.println("**********************");
  }
}
```

In jeder Schleife 3 System.out... + am Anfang, für erste Zeile

```
public class Programm15 {
  public static void main(String[] args) {
    int[][] matrix = new int[7][12];

    System.out.println("**********************");
    System.out.println("*** Naechste Zeile ***");
    System.out.println("**********************");
    for (int i = 0; i < matrix.length; i++) {
      for (int j = 0; j < matrix[i].length; j++) {
        System.out.print(matrix[i][j] + "\t");
      }
      System.out.println("**********************");
      System.out.println("*** Naechste Zeile ***");
      System.out.println("**********************");
    }
  }
}
```

Funktion: Code wird übersichtlicher

```
public class Programm16 {
  public static void zeilenTrennung() {
      System.out.println("**************************");
      System.out.println("*** Naechste Zeile ***");
      System.out.println("**************************");
  }
  public static void main(String[] args) {
    int[][] matrix = new int[7][12];

    zeilenTrennung();
    for (int i = 0; i < matrix.length; i++) {
      for (int j = 0; j < matrix[i].length; j++) {
        System.out.print(matrix[i][j] + "\t");
      }
      zeilenTrennung();
    }
  }
}
```

Funktionen: Vorteile

- Code übersichtlicher
- Funktionalität verständlicher
  - Funktion genau für eine Aufgabe verantwortlich
  - Wenn man weiss, was Funktion kann, muss man ihren Inhalt nicht genau verstehen
- Muss das Gleiche nur einmal schreiben
  - Viel kleinere Fehleranfälligkeit

Funktionen: Vorteile

Kleinere Fehleranfälligkeit

- Beispiel: Text `"******** Naechste Zeile ********"` ändern in `"******** weitere Linie ********"`

- Ohne Funktion
  - Muss Programm durchsuchen
  - Muss bei jedem `System.out.println()` ändern
  - Ev. hin und wieder Tippfehler...
- Mit Funktion
  - Muss Text nur einmal in Funktion ändern
  - Jeder Aufruf der Funktion `zeilenTrennung();` schreibt neuen Text

Funktionen mit Parametern

- Mathematik: Funktionen haben Parameter
  - $sin(x)$:
    - $sin()$ ist Funktion
    - x ist Parameter
    - $sin(x)$ nimmt Eingabe x, transformiert x und gibt Resultat zurück
    - Zuweisung Resultat: $y = sin(x)$

- Informatik: Funktionen haben Parameter
  - `zeilenTrennung(int zeilenNummer);`
    - `zeilenTrennung()` ist Funktion
    - `zeilenNummer` ist Parameter des Typs `int`

Funktionen mit Parametern

- Zelentrennung soll aktuelle Zeilennummer anzeigen
  - ```
    "**************************************"
    "********** Zeile 13 **********"
    "**************************************"
    ```
    ```

Funktion mit Parameter

```
public class Programm17 {
  public static void zeilenTrennung(int zNr) {
    System.out.println("*****************");
    System.out.println("*** Zeile " + zNr + " ***");
    System.out.println("*****************");
  }
  public static void main(String[] args) {
    int[][] matrix = new int[7][12];

    for (int i = 0; i < matrix.length; i++) {
      zeilenTrennung(i); //i als Uebergabeparameter
      for (int j = 0; j < matrix[i].length; j++) {
        System.out.print(matrix[i][j] + "\t");
      }
    }
  }
}
```

Funktionen/Methoden

- Funktionen heissen Methoden in objektorientierter Sprache

Methodensignatur

- Jede Methode besteht aus ...
  - ... einem Methodennamen
  - ... einer Anzahl Parameter
  - ... einem Datentyp pro Parameter
  - ... einer bestimmten definierten Reihenfolge der Parameter

Methodensignatur

- Jede Methode besteht aus ...
  - ... einem Methodennamen
  - ... einer Anzahl Parameter
  - ... einem Datentyp pro Parameter
  - ... einer bestimmten definierten Reihenfolge der Parameter

- Die Kombination aus dem Methodennamen und der Liste der typisierten Parameter ist die Signatur

Parametertypen müssen stimmen

- Methode hat bestimmte Signatur
  - Legt fest, welche Argument-Datentypen an welcher Stelle akzeptiert werden
- Methodenaufruf
  - System prüft, ob Anzahl übergebener Argumente mit Anzahl erwarteter Parameter stimmt
  - System vergleicht übergebene Argumente mit Methodensignatur
    - Stimmen alle Datentypen?
    - Können sie sonst automatisch konvertiert werden?

Parametertypen müssen stimmen

Parameter stimmen

```
public class Programm19 {
  public static void m1(int z1, int z2) {}

  public static void main(String[] args) {
    int zahl1 = 2; int zahl2 = 17;
    m1(zahl1, zahl2);
  }
}
```

Parametertypen müssen stimmen

Parameter stimmen nicht

```
public class Programm20 {
  public static void m2(boolean b1, boolean b2) {}

  public static void main(String[] args) {
    int zahl1 = 2; int zahl2 = 17;
    m2(zahl1, zahl2);
  }
}
```

Parametertypen müssen stimmen

Parameter werden konvertiert

```
public class Programm21 {
  public static void m3(long z1, long z2) {}

  public static void main(String[] args) {
    int zahl1 = 2; int zahl2 = 17;
    m3(zahl1, zahl2);
  }
}
```

Parametertypen müssen stimmen

Übergabe eines Arrays

```
public class Programm23 {
  public static void m4(int[] zahlen, int add) {}

  public static void main(String[] args) {
    int[] z = {4,1,2}; int addieren = 9;
    m4(z, addieren);
  }
}
```

Parameter: “Pass-by-Value”

- Parameter werden bei Übergabe an Methode kopiert
  - Methode kann originale Werte *nicht* ändern
- Bei Arrays wird *Zeiger* kopiert
  - Inhalt wird nicht kopiert
  - → Erinnerung: Zuweisung von Array-Variablen

Parameter: “Pass-by-Value”

Methode kann originale Werte nicht ändern

```
public class Programm22 {
  public static void aendere(int zahl) {
    zahl = 3;
  }

  public static void main(String[] args) {
    int zahl = 20;
    aendere(zahl);
    System.out.println("Zahl = " + zahl);
  }
}
```

Ausgabe: `Zahl = 20`

Parameter: “Pass-by-Value”

- Zur Erinnerung: Zuweisung von `null` zu Arrayvariable
  - “Pfeil”/Zeiger gelöscht → System räumt Speicher auf

Pass-by-Value

```
public class Programm51 {
  public static void loescheArray(int[] array) {
    array = null; // NUETZT NICHTS!!
  }

  public static void main(String[] args) {
    int[] zahlen = new int[200];
    loescheArray(zahlen);
  }
}
```

- Pass-by-Value = Kopie ⇒ Zwei Pfeile

Rückgabewerte

- Gesehen: $y = sin(x)$: Gibt Resultat zurück, wird y zugewiesen

- Rückgabewerte bei Methoden
  - Rückgabetyp *vor* Methodenname
    - `int`, `float`, `double`, `boolean`
    - `int[]`, ...
  - Methode gibt nichts zurück
    - `void`

Rückgabewerte

- Methode mit Rückgabewert (!=`void`) *muss* explizit Wert zurückgeben

- `return(WERT);`

Rückgabewerte

Methode: Addition

```
public class Programm18 {
  public static int addieren(int z1, int z2) {
    return (z1 + z2);
  }
  public static void main(String[] args) {
    int x = 13;
    int resultat = addieren(5, x);
    System.out.println("Resultat = " + resultat);
  }
}
```

Überladen von Methoden

Überladen von Methoden

- Gelernt: Methode hat eindeutigen Namen
- Manchmal braucht man eine Reihe ähnlicher Methoden

Überladen von Methoden

- Methode, die zwei Werte addiert
  - Methode heisst addiere2
  - `int addiere2(int z1, int z2)`
- Methode, die drei Werte addiert
  - Methode heisst addiere3
  - `int addiere3(int z1, int z2, int z3)`

- Was, wenn ich 4 oder 5 Werte addieren will?
- Was, wenn ich 2, 3, 4 oder 5 `float`-Werte addieren will?

Überladen von Methoden

- Compiler schaut sich nicht nur Methodennamen an
- Relevant für Compiler ist die Signatur
  - Signatur muss eindeutig sein

Überladen von Methoden

- Compiler schaut sich nicht nur Methodennamen an
- Relevant für Compiler ist die Signatur
  - Signatur muss eindeutig sein

- Methoden dürfen gleich heissen, wenn Signatur eindeutig ist
  - Andere Parametertypen
  - Unterschiedliche Anzahl Parameter

Überladen von Methoden

Überladen: Methoden haben gleichen Namen

```
public class Programm24 {
  public static int add(int z1, int z2) {}
  public static int add(int z1, int z2, int z3) {}
  public static float add(float z1, float z2) {}
  public static float add(float z1, float z2,
                          float z3) {}

  public static void main(String[] args) {
    int x = 13; float f1 = 13.2f;
    int resultat = add(5, x);
    System.out.println("Resultat = " + resultat);
    float resultatf = add(5 /* wird float */, f1);
    System.out.println("Resultat = " + resultatf);
  }
}
```

Lösen eines Problems

Lösen eines Problems

- Gegeben: Schwierige Aufgabe
- Gesucht: Algorithmus zur Lösung der Aufgabe

Lösen eines Problems

- Gegeben: Schwierige Aufgabe
- Gesucht: Algorithmus zur Lösung der Aufgabe

- Ansatz 1: Iterativer Algorithmus
  - Wie kann Problem mit Schleifen und Verzweigungen gelöst werden?
  - Algorithmus für ganzes Problem programmieren

Lösen eines Problems

- Gegeben: Schwierige Aufgabe
- Gesucht: Algorithmus zur Lösung der Aufgabe

- Ansatz 1: Iterativer Algorithmus
  - Wie kann Problem mit Schleifen und Verzweigungen gelöst werden?
  - Algorithmus für ganzes Problem programmieren
- Ansatz 2: Rekursiver Algorithmus
  - Annahmen:
    - Kann Problem in Teilprobleme unterteilen
    - Teilproblem ist gelöst
    - Ich weiss, wie man kleinen Lösungsschritt macht
  - Algorithmus für kleinen Schritt programmieren
  - Algorithmus mehrfach aufrufen

Rekursion

Beispiel

- Turm von Hanoi

Rekursion

Beispiel

- Turm von Hanoi
  - Drei Stangen A, B, C
  - Ein Turm aus der Grösse nach geordneter Holzscheiben
  - Ziel: Turm von Stange A nach Stange C verschieben

Rekursion

Beispiel

- Turm von Hanoi
  - Drei Stangen A, B, C
  - Ein Turm aus der Grösse nach geordneter Holzscheiben
  - Ziel: Turm von Stange A nach Stange C verschieben
- Regeln
  - Grössere Platte darf nicht auf kleinerer Platte sein
  - Kann nur eine Platte auf einmal verschieben

Rekursion

Beispiel

- Kann einen Turm der Grösse 1 verschieben (Rekursionsende)

Rekursion

Beispiel

- Kann einen Turm der Grösse 1 verschieben (Rekursionsende)
- Grösse 2:
  - Verschiebe oberen Teil von A nach B (Grösse 1)
  - Verschiebe dann unterste Platte nach C
  - Verschiebe schliesslich oberen Teil von B nach C

Rekursion

Beispiel

- Kann einen Turm der Grösse 1 verschieben (Rekursionsende)
- Grösse 2:
  - Verschiebe oberen Teil von A nach B (Grösse 1)
  - Verschiebe dann unterste Platte nach C
  - Verschiebe schliesslich oberen Teil von B nach C
- Grösse 3:
  - Verschiebe oberen Teil von A nach B (Grösse 2)
  - Verschiebe dann unterste Platte nach C
  - Verschiebe schliesslich oberen Teil von B nach C

Rekursion

Beispiel

- Beispiel: Fakultät von n berechnen

Rekursion

Beispiel

- Beispiel: Fakultät von n berechnen

- Iterativer Algorithmus
  1. Neue Variable mit Wert 1 $\rightarrow$ (Zwischen)resultat
  2. Neue Variable mit Wert 1 $\rightarrow$ Zählvariable
  3. Schleife, solange Zählvariable $< n$
  4. Zwischenresultat mit Zählvariable multiplizieren

Rekursion

Beispiel

- Beispiel: Fakultät von n berechnen

- Iterativer Algorithmus
  1. Neue Variable mit Wert 1 $\rightarrow$ (Zwischen)resultat
  2. Neue Variable mit Wert 1 $\rightarrow$ Zählvariable
  3. Schleife, solange Zählvariable $< n$
  4. Zwischenresultat mit Zählvariable multiplizieren
- Rekursiver Algorithmus
  1. Fakultät n = n mal Fakultät $n - 1$
     - 1 Schritt
     - $\rightarrow$ Rufe also zuerst Fakultät $n - 1$ auf
  2. Fakultät 1 ist einfach zu berechnen

Iterativ

```
public class Programm50 {
  public static int fakultaet_it(int n) {
    int res = 1;
    for (int i = 1; i <= n; i++) {
      res *= i;
    }
    return (res);
  }
}
```

Rekursiv

```
public class Programm50 {
  public static int fakultaet_rek(int n) {
    return((n > 1 ? n * fakultaet_rek(n - 1) : 1));
  }
}
```

Rekursion

Wichtig

Das Wichtigste ist das Rekursionsende!

Rekursion

Wichtig

- Rekursionsmethode ruft sich selber auf
  - Dann wieder sich selber
    - Dann wieder sich selber
    - ...

Rekursion

Wichtig

- Rekursionsmethode ruft sich selber auf
  - Dann wieder sich selber
    - Dann wieder sich selber
    - ...
- Muss irgendwann ein Ende haben und aufhören, sich selbst aufzurufen

Rekursion

Rekursionsende Fakultät

- Rekursionsende Fakultät: Fakultät von 1 ist 1
  - Das weiss ich, muss mich nicht selbst aufrufen, um Fakultät von 1 zu berechnen
- Rekursionsende möglichst am Anfang hinschreiben

- 1. Schritt: Rekursionsende hinschreiben

Rekursionsende

```
public class Programm50 {
  public static int fakultaet_rek(int n) {
    if (n == 1) {
      return (1); //kein Aufruf mehr, direkt return
    }
    ...
  }
}
```

Kompakte Schreibweise

```
public class Programm50 {
  public static int fakultaet_rek(int n) {
    return((n > 1 ? n * fakultaet_rek(n - 1) : 1));
  }
}
```

- 2. Schritt: Rekursion: Sich selbst mit einfacherem Wert aufrufen

Rekursionsende

```
public class Programm50 {
  public static int fakultaet_rek(int n) {
    if (n == 1) {
      return (1); //kein Aufruf mehr, direkt return
    }
    // einfacherer Wert: n - 1 und
    // naechster Schritt: Multiplikation mit n
    return (n * fakultaet_rek(n - 1));
  }
}
```

Rekursion

Vor- und Nachteile

Rekursion

Vor- und Nachteile

- Vorteile
  - Einfach, einen Schritt zu programmieren
  - Einfach, Elemente in Datenstrukturen einzuhängen (→ später)
  - Aktueller Zustand implizit durch Programm/Rekursion gespeichert

Rekursion

Vor- und Nachteile

- Vorteile
  - Einfach, einen Schritt zu programmieren
  - Einfach, Elemente in Datenstrukturen einzuhängen (→ später)
  - Aktueller Zustand implizit durch Programm/Rekursion gespeichert
- Nachteile
  - Muss richtiges Rekursionsende finden
  - Grosser Speicherverbrauch bei grosser Rekursionstiefe
    - → Aktueller Zustand

Rekursion

Vor- und Nachteile

- Vorteile
  - Einfach, einen Schritt zu programmieren
  - Einfach, Elemente in Datenstrukturen einzuhängen (→ später)
  - Aktueller Zustand implizit durch Programm/Rekursion gespeichert
- Nachteile
  - Muss richtiges Rekursionsende finden
  - Grosser Speicherverbrauch bei grosser Rekursionstiefe
    - → Aktueller Zustand

- Demo: Fakultät von 10000

Rekursion

Fazit

Rekursion

Fazit

- Angenehme Lösungsstrategie
- “Merkt” sich aktuellen Zustand
- Vorsicht beim Rekursionsende