Programmierkurs

Vorlesung 06

M.Sc. Laslo Hunhold

Department Mathematik/Informatik
Abteilung Informatik
Universität zu Köln

14. Dezember 2020

Letzte Vorlesung

- Standardwerte für Typen
- Vertiefung zu Objekten (Klasse, Speicherstruktur)
- Vertiefung zu mehrdimensionalen Arrays
- Zeichen und Zeichenketten (Unicode, String, Stringwerkzeuge)

Methoden

Motivation

```
1  double[] arr = { 1.0, 5.2, 8.3, 33.5, 22.2 };
2  double d;
3
4  /* Durchschnitt berechnen */
5  d = 0.0;
6  for (int i = 0; i < arr.length; i++) {
7          d += arr[i];
8  }
9  d /= arr.length;
```

```
1  double[] arr = { 1.0, 5.2, 8.3, 33.5, 22.2 };
2  double d = durchschnitt(arr);
```

- Abstrahierung/Strukturierung von Arbeitsschritten (bessere Lesbarkeit, Bibliotheken)
- Verhinderung von Wiederholungen („Deduplikation“, weniger Quelltext, geringere Fehleranfälligkeit)
- Durchschnittsberechnung: Eingabe: `double`-Array, Ausgabe: `double`

Beschreibung

```
double[] arr = { 1.0, 5.2, 8.3, 33.5, 22.2 };
double d = durchschnitt(arr);
```

- `durchschnitt(arr)` ist ein „Methodenaufruf“
- Methode `durchschnitt` erhält den „Parameter“ `arr` und gibt einen Wert („Rückgabewert“) zurück, den wir in `d` speichern
- Programmausführung „springt“ in die Funktion und kommt wieder

```
konsolenausgabe("OK");
```

- Methode mit Rückgabewert: „Funktion“
- Methode ohne Rückgabewert: „Prozedur“
- Rückgabewert kann „verworfen“ werden

```
double[] arr = { 1.0, 5.2, 8.3, 33.5, 22.2 };
durchschnitt(arr);
```

Deklaration

```
public static void main(String[] args) { ... }
```

```
static double durchschnitt(double[] a) { ... }
```

```
void f() { ... }
```

- Deklaration in einer Klasse (Sichtbarkeitsbereich)
- Allgemeine Form des „Methodenkopfs“ gefolgt vom „Methodenrumpf“ (`{ ... }`):
  `MODIFIKATOREN RÜCKGABETYP NAME(PARAMETER) { ... }`
- `MODIFIKATOREN`: Optionale, durch Leerzeichen getrennte Liste aus `static`, `final`, `public`, `private` (später)
- `RÜCKGABETYP`: Typ des zurückgegebenen Werts, z.B. `double`; oder `void`, falls kein Wert zurückgegeben wird
- `NAME`: Bezeichner der Methode, z.B. `durchschnitt`, Konvention: Klein geschrieben
- `PARAMETER`: Optionale, durch Kommata getrennte Liste aus (Eingangs-)Variablen, z.B. „`int[] arr, double d`“

Beispiel

```
1  class Main {
2          static void ausgabe(String name, double w) {
3                  System.out.println("Wert von " + name + ": " + w);
4          }
5
6          static double durchschnitt(double[] a) {
7                  double d = 0.0;
8
9                  for (int i = 0; i < a.length; i++) {
10                         d += a[i];
11                 }
12                 d /= a.length;
13
14                 return d;
15         }
16
17         public static void main(String[] args) {
18                 double[] arr = { 1.0, 5.2, 8.3, 33.5, 22.2 };
19                 double d = durchschnitt(arr);
20                 ausgabe("Durchschnitt", d);
21         }
22 }
```

Methodenparameter

- Parameter einer Methode sind „vorgefüllte“ lokale Variablen; haben nur in der Methode Gültigkeit

```
static void ausgabe(String name, double w) {
        System.out.println("Wert von " + name + ": " + w);
}
```

```
ausgabe("Durchschnitt", 12.5);
```

- Werte werden implizit gecastet (wie bei einer Zuweisung)

```
ausgabe("Durchschnitt", 12);
```

- Werte werden in die Parameter *kopiert* (Bei Zeigern nur die *Adressen*; es werden keine neuen Objekte erzeugt!)

Beispiel

```
 1  class Main {
 2          static void mod(int a, int[] b) {
 3                  a = 0;
 4                  b[0] = 0;
 5          }
 6
 7          public static void main(String[] args) {
 8                  int m = 1;
 9                  int[] n = { 1 };
10
11                  /* m = 1, n[0] = 1 */
12
13                  mod(m, n);
14
15                  /* m = 1, n[0] = 0 */
16          }
17  }
```

Lokale Variablen in Methoden

- Lokale Variablen wie gewohnt benutzbar
- Nicht von außen sichtbar
- Dürfen nicht den Namen eines Parameters haben (offensichtlich)
- Parameter einer Methode sind also im selben Namensraum wie ihre lokalen Variablen

```
1  static double durchschnitt(double[] a) {
2          double d = 0.0;
3
4          for (int i = 0; i < a.length; i++) {
5                  d += a[i];
6          }
7          d /= a.length;
8
9          return d;
10 }
```

`return`-Anweisung

- `return` steigt aus der Methode aus („kehrt zurück“)
- Bei Prozeduren *ohne* Rückgabewert

```
if (n < 0) {
        return;
}
```

- Prozedur ohne `return` wird einfach bis zum Methodenrumpfende (i.e. Blockende) ausgeführt
- Bei Funktionen `return` mit Rückgabewert (impliziter Cast auf Rückgabetyp)

```
return 19;
```

- Funktionen brauchen zwingend ein `return`, meist am Ende des Methodenrumpfs

Instanz- und Klassenmethoden (`static`-Modifikator)

- Methoden greifen eventuell auf Instanzvariablen einer Klasse zu

```
class Bruch {
        int zaehler;
        int nenner;

        void multipliziere(int f) {
                zaehler *= f;
        }

        static void test(int x) {
                System.out.println("Hallo Welt! " + x);
        }
}
```

- Die Methode `multipliziere` kann nur auf *Instanzen* der Klasse `Bruch` verwendet werden („Instanzmethode“)
- Die Methode `test` kann auch ohne Instanz der Klasse `Bruch` verwendet werden („Klassenmethode“), angezeigt durch `static`-Modifikator

Aufruf

- Innerhalb der definierenden Klasse: Einfach über den Namen

```
class Main {
        static void hallo() {
                System.out.println("Hallo Welt!");
        }
        public static void main(String[] args) {
                hallo();
        }
}
```

- Außerhalb der definierenden Klasse: Analog zu Instanz- und Klassenvariablen mit dem Punktoperator
- Instanzmethoden werden über die Instanz aufgerufen

```
Bruch b = new Bruch();
b.multipliziere(2);
```

- Klassenmethoden werden über die Klasse aufgerufen (Instanz geht auch, aber gibt eine Warnung vom Compiler)

```
Bruch.test(2);
```

Beispiel

```
class Main {
        static Bruch erzeuge_bruch(int z, int n) {
                Bruch b = new Bruch();

                b.zaehler = z;
                b.nenner = n;

                return b;
        }

        public static void main(String[] args) {
                Bruch b = erzeuge_bruch(1, 2);
        }
}
```

Überladung

Motivation

```
class Bruch {
        int zaehler;
        int nenner;

        void multipliziere(int f) {
                zaehler *= f;
        }
}
```

- Können einen `Bruch` mit einem Integer multiplizieren
- Wollen aber auch mit einem anderen `Bruch` multiplizieren

```
        void multiplziereBruch(Bruch b) {
                zaehler *= b.zaehler;
                nenner *= b.nenner;
        }
```

- Viele Namen für den gleichen Effekt. Geht das eleganter?

Signatur/Schnittstelle

- Name und Parameterliste einer Methode ist die „Signatur“ bzw. „Schnittstelle“
- Nicht dazu gehört der Rückgabetyp, die Parameternamen und Modifikatoren
- Die Funktionen

```
static int max(int x, int y) { ... }
```

und

```
double max(int a, int b) { ... }
```

haben dieselbe Signatur „`max(int, int)`“

Überladung

- Im selben Namensraum müssen sich Methoden nur in ihrer Signatur, nicht in ihrem Namen, unterscheiden
- Kommt ein Methodenname mehr als einmal vor (mit unterschiedlichen Signaturen) ist dieser „überladen“
- Compiler wählt bei Verwendung automatisch die passende Methode, je nach Eingabeparameter(n)

```
class Bruch {
        int zaehler;
        int nenner;

        void multipliziere(int f) {
                zaehler *= f;
        }

        void multipliziere(Bruch b) {
                zaehler *= b.zaehler;
                nenner *= b.nenner;
        }
}
```

Konstruktoren

Motivation

```
class Bruch {
        int zaehler;
        int nenner;
}
```

```
Bruch b = new Bruch();
/* b.zaehler = 0, b.nenner = 0 */
```

- Instanzvariablen werden bei Instanziierung auf Standardwerte gesetzt
- Standardwerte aber nicht immer gut; Bruch 0/0 ist mathematisch nicht definiert

```
Bruch b = new Bruch();
b.zaehler = 1;
b.nenner = 2;
```

- Werte von Hand zu setzen ist umständlich
- Bemerkung: `Bruch()` sieht aus wie ein Methodenaufruf (!)

Konstruktor

- Nach der Speicherallokation für die Instanz wird eine Methode aufgerufen, die die Instanz mit Werten füllt (der „Konstruktor“)
- Ist kein Konstruktor definiert (also wie bisher) wird der sogenannte „Default-Konstruktor“ aufgerufen, der alle Instanzvariablen mit Standardwerten füllt

Deklaration

- Deklaration wie eine Methode, aber ohne Modifikatoren und Rückgabetyp; kann überladen werden
- Der Name entspricht dem Klassennamen
- Ist ein Konstruktor definiert, gibt es keinen Default-Konstruktor mehr

```
1  class Test {
2          int wert;
3
4          Test(int w) {
5                  wert = w;
6          }
7  }
```

```
1  Test t1 = new Test(); /* Fehlermeldung */
2  Test t2 = new Test(42);
```

Beispiel

```
 1  class Bruch {
 2          int zaehler;
 3          int nenner;
 4
 5          Bruch() {
 6                  zaehler = 0;
 7                  nenner = 1;
 8          }
 9
10          Bruch(int z, int n) {
11                  zaehler = z;
12                  nenner = n;
13          }
14  }
```

```
Bruch b = new Bruch();
Bruch c = new Bruch(1, 2);
```

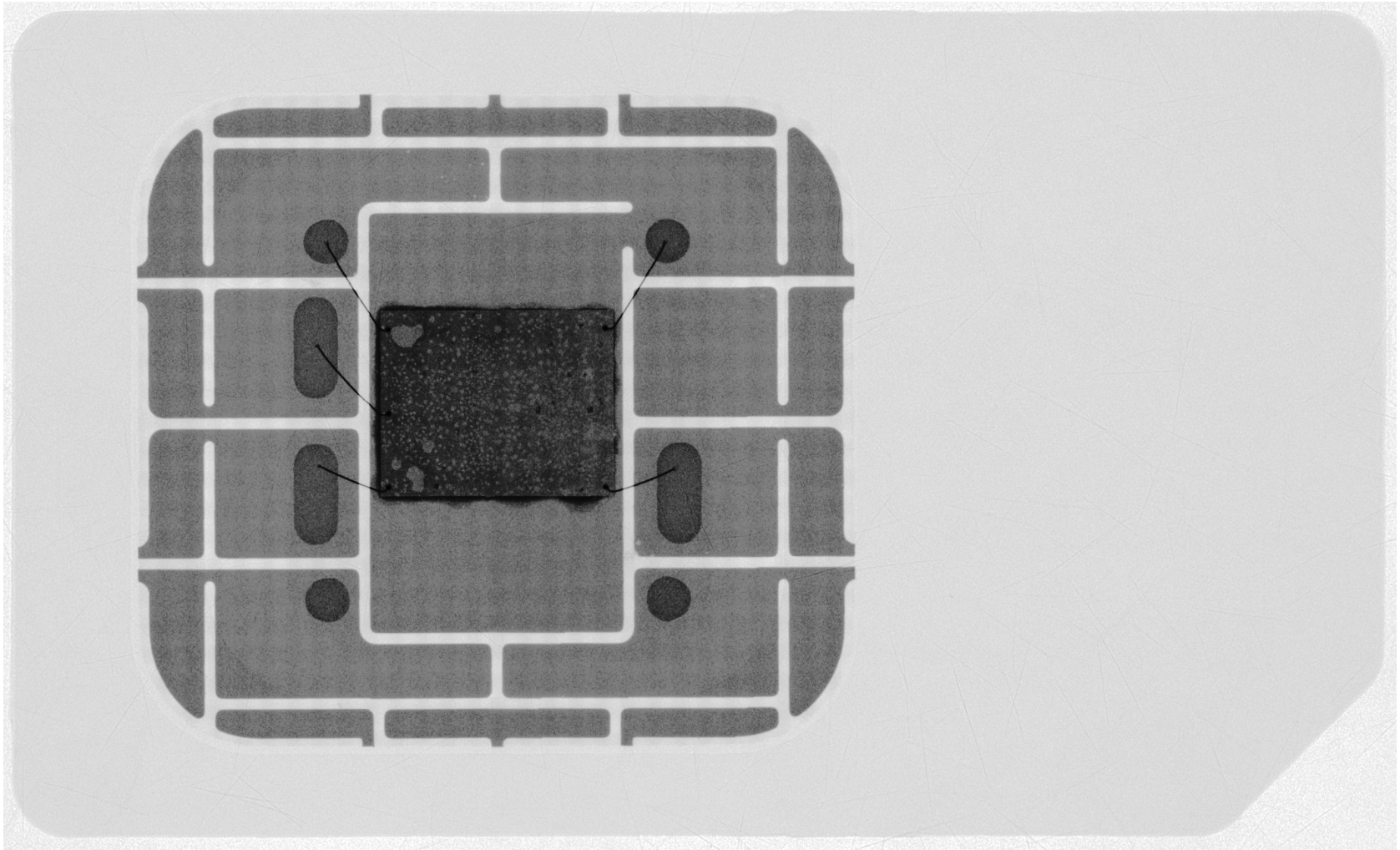

Abbildung: Mini-SIM-Karte (2FF) (Quelle: SecretDisc, CC BY-SA 3.0)