Beispiel: die Klasse Brüche

```
class Fraction {
    int num;        // numerator
    Int denom;      // denominator

    Fraction (int n, int d) {
        num = n; denom = d;
    }

    void add (Fraction f) {
        num = num * f.denom + f.num * denom;
        denom = denom * f.denom;
    }

    void mult (Fraction f) {
        num = num * f.num;
        denom = denom * f.denom;
    }

    void reduce () {
        int t = ggt (num, denom);
        num = num / t;
        denom = denom / t;
    }

    static int ggt (int n, int d) { … }

    public String toString () {
        return num + “/“ + denom;
    }

    public static void main (String[] a) {
        Fraction p = new Fraction (5,6);
        Fraction q = new Fraction (3,4);
        p.add(q);    System.out.println (p);
        p.reduce(); System.out.println (p);
        p.mult(q);   System.out.println (p);
    }
}
```


Klassenmethoden vs. (Objekt-)Methoden

- Jedesmal, wenn ein Fraction-Objekt erzeugt wird, erhöht der entsprechende Objekt-Konstruktor das Klassenattribut **fractionCounter** um eins.
- Die Klassenmethode **resetFractionCounter** setzt das Klassenattribut fractionCounter auf 0 zurück.
- Jede Klasse hat höchstens einen Klassenkonstruktor (ohne Namen, ohne Parameter).

```
class Fraction {
    // ----- class field
    static int fractionCounter = 0;

    // ----- object fields
    int n = 1;
    int d = 1;

    // ----- class method
    static void resetFractionCounter () {
        fractionCounter = 0;
    }

    // ----- constructor and object methods
    Fraction (int n, int d) {
        this.n = n; this.d = d;
        fractionCounter++;  /* new object created =>
                               increase counter */
    }

    void add (Fraction f) {
        n = n * f.d + f.n * d;
        d = d * f.d;
    }
    ...

}
```

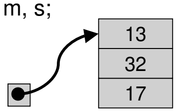

Klassenmethoden vs. (Objekt-) Methoden

```
class Fraction {
    ...
    Fraction (int n, int d) { // object constructor
        this.n = n; this.d = d;
        fractionCounter++; // new object created => increase counter
    }

    static { // class constructor
        fractionCounter = 0;
    }
    ...
}
```

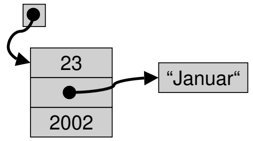

Klassen als Datenstrukturen

- Zusammenfassung verschiedener Attribute zu einem Objekt

```
class Time {
  int h, m, s;
}

Time t;
```




```
class Date {
  int day;
  String month;
  int year;
}
```



- Beispiel: Rückgabe mehrerer Funktionsergebnisse auf einmal

  Realisiert als Rückgabe eines einzigen komplexen Ergebnisobjekts:

```
static Time convert (int sec) {
  Time t = new Time();
  t.h = sec / 3600; t.m = (sec%3600) / 60; t.s = sec % 60;
  return t;
}
```

Klassen vs. Arrays

Klassen

- Bestehen im allgemeinen aus *verschiedenartigen* Elementen:
  class c {String s; int i;}
- Jedes Element hat einen eigenen Namen: c.s, c.i
- Anzahl der Elemente wird *statisch* bei der Deklaration der Klasse festgelegt.

Arrays

- Bestehen immer aus mehreren *gleichartigen* Elementen: int[]
- Elemente haben keine eigenen Namen, sondern werden über Indizes angesprochen: a[i]
- Anzahl der Elemente wird *dynamisch* bei der Erzeugung des Arrays festgelegt:
  new int [*n*]

Beispiel: Punkte im 2D

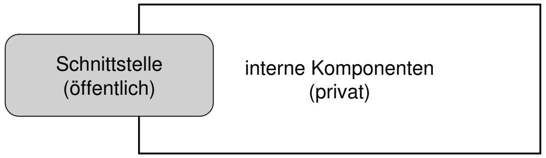
```
class Point2 {
    /** Cartesian Coordinates */
    double x, y;

    Point2 (double x, double y) {
        this.x = x; this.y = y;
    }
    /** add a vector */
    void add (Point2 p) {
        x += p.x; y += p.y;
    }
    /** multiply with a scalar */
    void scale (double s) {
        x *= s; y *= s;
    }
    /** return scalar product */
    double scalar (Point2 p) {
        return x * p.x + y * p.y;
    }
    /** return string representation */
    public String toString () {
        return "(" + x + ", " + y + ")";
    }

    /** test the methods */
    public static void main (String[] a) {
        Point2 p = new Point2 (…);
        Point2 q = new Point2 (…);
        …
        p.add (q);                  // vgl. p += q
        p.scale (5.0);              // vgl. p *= 5.0
        double s = p.scalar (q);    // vgl. s = p * q
    }
}
```


Datenabstraktion

- Ein Grundprinzip der Objektorientierung
  - Trennung von Spezifikation (*was*) und Realisierung (*wie*) von Typen.
- Spezifikation von Typen: „WAS“
  - beschreibt, *was* Objekte leisten können und wie sie verwendet werden.
  - wird durch öffentlich sichtbare Schnittstelle beschrieben.
  - Syntax ergibt sich aus Signatur, Semantik aus Kommentaren (bei Java).
- Realisierung (Implementierung) von Typen: „WIE“
  - legt fest, *wie* das spezifizierte Verhalten der Objekte erreicht wird.
  - zur Implementierung der sichtbaren Schnittstelle können zusätzliche private Komponenten eingeführt werden.

Datenabstraktion (2)

- Schema der Datenkapselung



- Vorteile der Datenabstraktion
  - Schaffung von Verantwortlichkeitsbereichen für die Programmentwicklung im Team   höhere Produktqualität
  - Austausch der Implementierung einer Schnittstelle ohne Änderung der verwendenden Anwendungsprogramme möglich.

Datenabstraktion in Java

- Datenkapselung
  - **public** kennzeichnet öffentliche Klassen und öffentliche Komponenten.
  - **private** kennzeichnet nicht-öffentliche Komponenten.
  - **protected** kennzeichnet eingeschränkten Zugriff (für Unterklassen).
  - „package“ (= kein Schlüsselwort, default) kennzeichnet eingeschränkten Zugriff (für Pakete).

- Strikte Datenkapselung
  - gesamter Objektzustand (= alle Attribute) wird als **private** deklariert.
  - Zustandsänderungen werden nur über öffentliche Methoden erlaubt.
  - Beispiel:

```
private int denom;
public int getDenom ()  {return denom;}
public void setDenom (int d)  {if (d != 0) denom = d; else /*Fehler*/}
```


Abstrakter Datentyp (ADT)

- Definition eines **interface**
  - Name des neuen Typs.
  - Signaturen der Methoden.
  - keine Methodenrümpfe.
  - keine Attribute.

- Eigenschaften
  - Instantiierung:
    nicht unmittelbar möglich.
    Implementierung durch andere Klassen nötig.
  - Verwendung:
    wie jeder andere Objekttyp.
    Objektstruktur nicht bekannt.

- Beispiel: Liste ganzzahliger Werte

```
public interface IntList {
    public void insert (int value);
    public boolean contains (int value);
    public void delete (int value);
}
```

- Implementierungsbeispiel

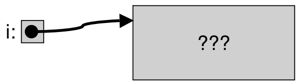
```
class MyIntList implements IntList {
    … (( insert, contains, delete ))
}
```


Abstrakte Datentypen: Verwendung

- Verwendungsbeispiel

```
class IntListTest {
    static void doTheTest (IntList l) {
        l.insert (4);
        l.insert (5);
        …
        if (l.contains (3)) …
    }

    public static void main (String[] a) {
        IntList l = new MyIntList ();
        doTheTest (l);
    }
}
```

- Bemerkungen
  - Methode *doTheTest* kennt weder die Struktur noch die inneren Abläufe einer aktuellen *IntList*.
  - Beim Aufruf einer Methode wird die die zum aktuellen Objekt gehörige Implementierung ermittelt und ausgeführt (*dynamisches Binden*).
  - Ein Aufrufer weiß also, **was** er mit einem Objekt anfangen kann, aber nicht, **wie** es implementiert ist.
  - Dasselbe **interface** kann unterschiedlich implementiert sein.



Implementierung als Array

```
class IntArray implements IntList {
    int[] elems;
    int count = 0;

    public IntArray (int maxNum) {
        elems = new int [maxNum];
    }

    /** unsortiertes Einfügen am Arrayende */
    public void insert (int value) {
        if (count < elems.length) {
            elems [count] = value;
            ++ count;
        } else
            System.err.println (“Array is full“);
    }

    public boolean contains (int value) {
        return getPosition (value) >= 0;
    }

    /** sequentielle Suche: linearer Aufwand */
    private int getPosition (int value) {
        for (int p = 0; p < count; ++p)
            if (elems[p] == value) return p;
        return –1;  /* indicates „not found“ */
    }

    public void delete (int value) {
        int p = getPosition (value);
        if (p >= 0) {
            -- count;
            while (p < count) {
                elems [p] = elems[p+1];
                ++p;
            }
        }
    }
}
```

Implementierung als sortiertes Array

```
class IntArraySorted implements IntList {
    int[] elems;
    int count = 0;

    public IntArraySorted (int maxNum) {
        elems = new int [maxNum];
    }

    public boolean contains (int value) {
        return getPosition (value) >= 0;
    }

    /** binäre Suche */
    private int getPosition (int value) {
        …
    }

    /** sortiertes Einfügen */
    public void insert (int value) {
        if (count < elems.length) {
            int i = count;
            while (i > 0 && elems[i-1] > value) {
                elems [i] = elems[i-1];
                --i;
            }
            elems [i] = value;
            ++ count;
        } else
            System.err.println (“Array is full“);
    }

    public void delete (int value) {
        … (( wie vorher ))
    }
}
```


Dynamische Erweiterung

- Nachteile der Implementierung durch Arrays
  - Arrays haben eine feste Länge (hier: maxNum).
  - Arrays können nicht dynamisch erweitert werden.
  - Problem beim „Überlauf“ des Arrays.
  - Bislang: System.err.println (“Array is full“);

- Lösungsmöglichkeit
  - Erzeugen eines neuen Arrays der doppelten Länge.
  - Alte Elemente an den Anfang kopieren.

```
int[] elemsNew = new int [2*elems.length];
System.arraycopy (elems, 0, elemsNew, 0, elems.length);
this.elems = elemsNew;
```