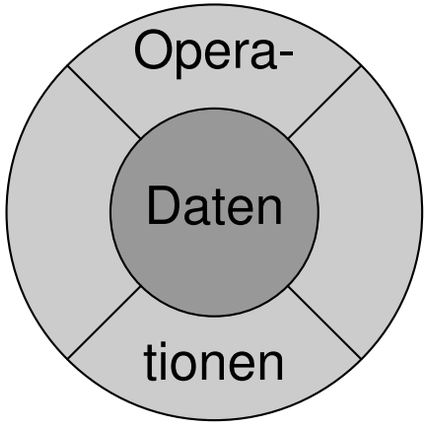
4 Klassen in Java

4.1 Grundsätzliches

- Eine Klasse definiert einerseits eine Datenstruktur (in Form von *Feldern*) und andererseits die zugehörigen Operationen (in Form von *Methoden*).

- Häufig sind die Datenfelder *privat*, d. h. „von außen“ nicht direkt zugänglich, während die Methoden *öffentlich* sind, d. h. von „Klienten“ aufgerufen werden können (*Kapselung*, *Geheimnisprinzip*). Dadurch können Implementierungsdetails der Klasse geändert werden, ohne dass dies Auswirkungen auf Klientencode hat.



- Klassen sind *Referenztypen* (vgl. § 3.2.2), d. h. Variablen (und Ausdrücke), deren Typ eine Klasse ist, besitzen als Werte *Referenzen auf Objekte* dieser Klasse (oder die Nullreferenz `null`).

4.2 Aufbau einer Klasse

❒ Eine Klasse kann folgende Elemente in beliebiger Reihenfolge enthalten:

	statisch	nicht-statisch	
Felder	Klassenvariablen	Objektvariablen	§ 4.5
Methoden	Klassenmethoden	Objektmethoden	§ 4.6
Konstruktoren	–	+	§ 4.7
Initialisierungsblöcke	Klasseninitialisierer	Objektinitialisierer	§ 4.8
geschachtelte Klassen	+	innere Klassen	
geschachtelte Schnittstellen	+	–	

❒ Anders als in C, wo striktes „declare before use“ gilt, sind alle Elemente einer Klasse an jeder Stelle der Klasse sichtbar und (mit gewissen Einschränkungen bei Initialisierungen) verwendbar.

❒ Für die Lesbarkeit einer Klasse ist es aber trotzdem hilfreich, Elemente erst nach ihrer Deklaration zu verwenden (obwohl manche Programmierer es bewusst anders machen und private Elemente grundsätzlich erst nach öffentlichen Elementen definieren).

4.3 Zugriffskontrolle

- Elemente einer Klasse (mit Ausnahme von Initialisierungsblöcken) können mit folgenden Zugriffsbeschränkungen deklariert werden:
  - `private` (privat)
    Zugriff nur innerhalb der Klasse selbst erlaubt
  - *keine Angabe* (paketöffentlich)
    Zugriff im gesamten Paket erlaubt, zu dem die Klasse gehört (vgl. Kap. 9)
  - `protected` (unterklassenöffentlich)
    Zugriff im gesamten Paket sowie (mit gewissen Einschränkungen) in allen Unterklassen der Klasse erlaubt (vgl. Kap. 5.2)
  - `public` (öffentlich)
    Zugriff überall erlaubt
- Eine Klasse selbst kann entweder öffentlich (Schlüsselwort `public`) oder paketöffentlich (keine Angabe) sein (vgl. Kap. 9).

4.4 Beispiel: Bankkonten (vgl. § 2.4)

```
// Klasse: Konto.
class Account {
  // Private Klassenvariable:
  // Nächste zu vergebende Kontonummer.
  private static int nextNumber = 1;

  // Private Objektvariablen:
  private final int number = nextNumber++;
                                  // Kontonummer (unveränderlich).
  private String holder;          // Kontoinhaber.
  private int balance = 0;        // Kontostand.

  // Öffentliche Konstruktoren: Konto mit Inhaber h, ggf.
  // Anfangsbetrag b und eindeutiger Nummer initialisieren.
  public Account (String h) {
    holder = h;
  }
  public Account (String h, int b) {
    this(h);                      // Den anderen Konstruktor aufrufen.
    balance = b;
  }
```

```
  // Öffentliche Objektmethoden:
  // Kontonummer/-inhaber/-stand abfragen.
  public int number () { return number; }
  public String holder () { return holder; }
  public int balance () { return balance; }

  // Öffentliche Objektmethoden:
  // Betrag amount einzahlen/abheben/überweisen.
  public void deposit (int amount) {
    balance += amount;
  }
  public void withdraw (int amount) {
    balance -= amount;
  }
  public void transfer (int amount, Account that) {
    withdraw(amount);
    that.deposit(amount);
  }

  // Öffentliche Klassenmethode:
  // Anzahl bereits erzeugter Konten abfragen.
  public static int numberOfAccounts () { return nextNumber - 1; }
}
```

4.5 Felder

4.5.1 Objektvariablen (nicht-statische Felder)

- Objektvariablen entsprechen Strukturkomponenten in C.
- Jedes Objekt einer Klasse besitzt eigene Ausprägungen der Objektvariablen der Klasse.
- Objektvariablen können Initialisierungsausdrücke besitzen, die bei jeder Erzeugung eines Objekts der Klasse ausgewertet werden (vgl. § 4.9).
- Nicht explizit initialisierte Objektvariablen erhalten bei der Erzeugung eines Objekts der Klasse einen typabhängigen Standardwert (vgl. § 3.2.3).
- Beispiele:

```
private final int number = nextNumber++;
private String holder;
private int balance = 0;
```

- Der Zugriff auf Objektvariablen erfolgt mit dem Punktoperator: `object.objvar`
  Dabei kann `object` ein beliebiger (ggf. geklammerter) Ausdruck sein, dessen Resultat ein Objekt der Klasse darstellt.

- Beispiele:

```
a.number
(a = new Account("Heinlein")).balance
```

- Die Objektvariablen des aktuellen Objekts (vgl. § 4.6.1) können auch direkt über ihren Namen angesprochen werden, d. h. `objvar` ist äquivalent zu `this.objvar`.

4.5.2 Klassenvariablen (statische Felder)

- Klassenvariablen entsprechen globalen Variablen in C, deren Zugriff jedoch beschränkt werden kann (vgl. § 4.3).

- Alle Objekte einer Klasse „teilen sich“ die Klassenvariablen der Klasse, d. h. verwenden sie gemeinsam.

- Klassenvariablen können ebenfalls Initialisierungsausdrücke besitzen, die einmalig beim Laden/Initialisieren der Klasse ausgewertet werden (d. h. normalerweise zu Beginn der Programmausführung).

- Nicht explizit initialisierte Klassenvariablen besitzen analog zu Objektvariablen typabhängige Standardwerte.

- Beispiel:

```
private static int nextNumber = 1;
```

- Der Zugriff auf Klassenvariablen erfolgt normalerweise über den Klassennamen: `classname.classvar`

- Die Klassenvariablen der aktuellen Klasse können auch direkt über ihren Namen angesprochen werden.

- Prinzipiell können Klassenvariablen auch über irgendein Objekt der Klasse angesprochen werden: `object.classvar`
  Das ist allerdings nicht üblich und wird von manchen Übersetzern mit einer Warnung quittiert. Das Objekt `object` wird hierbei gar nicht verwendet.

- Beispiele:

```
Account.nextNumber      // Zugriff über Klassenname
nextNumber              // Direkter Zugriff über Variablenname
a.nextNumber            // Zugriff über ein Objekt der Klasse
```

4.5.3 Unveränderliche Felder

- Objekt- bzw. Klassenvariablen, die `final` deklariert sind, müssen während der Initialisierung eines Objekts bzw. der Klasse (und vor ihrer ersten Verwendung) initialisiert werden und dürfen anschließend nicht mehr verändert werden.
  (Die Initialisierung eines Objekts besteht aus den in § 4.9 beschriebenen Schritten 4 und 5.)

- Beispiel:

```
private final int number = nextNumber++;
```

  (Alternativ kann die Initialisierung `number = nextNumber++` in einem Konstruktor oder Objektinitialisierer der Klasse erfolgen.)

- Unveränderliche Klassenvariablen, deren Wert bereits vom Übersetzer bestimmt werden kann, belegen zur Laufzeit des Programms keinen Speicherplatz und werden an jeder Verwendungsstelle direkt durch ihren Wert ersetzt (ähnlich wie Makros in C).

- Beispiel:

```
public static final int FIRST_NUMBER = 1;
```

  Entspricht in C:

```
#define FIRST_NUMBER 1
```

4.6 Methoden

4.6.1 Objektmethoden (nicht-statische Methoden)

- Objektmethoden entsprechen Funktionen in C, die einen zusätzlichen impliziten Parameter `this` besitzen, dessen Typ die umgebende Klasse ist und der das *aktuelle Objekt* der Klasse bezeichnet.
- Beispiele:

```
public int number () { return this.number; }

public void transfer (int amount, Account that) {
  withdraw(amount);      // Äquivalent zu: this.withdraw(amount);
  that.deposit(amount);
}
```

- Objektmethoden werden – analog zu Objektvariablen – immer für ein bestimmtes Objekt aufgerufen: `object.objmeth([arguments])`
- Innerhalb einer Objektmethode steht das Aufrufobjekt `object` dann als aktuelles Objekt `this` zur Verfügung.
- Die Objektmethoden des aktuellen Objekts können auch direkt über ihren Namen angesprochen werden, d. h. `objmeth` ist äquivalent zu `this.objmeth`.

4.6.2 Klassenmethoden (statische Methoden)

- Klassenmethoden entsprechen globalen Funktionen in C, deren Zugriff jedoch beschränkt werden kann (vgl. § 4.3).
- Beispiel:

```
public static int numberOfAccounts () { return nextNumber - 1; }
```

- Klassenmethoden werden normalerweise über den Klassennamen aufgerufen:
  `classname.classmeth([arguments])`
- Die Klassenmethoden der aktuellen Klasse können auch direkt über ihren Namen angesprochen und aufgerufen werden.
- Prinzipiell können Klassenmethoden auch über irgendein Objekt der Klasse aufgerufen werden: `object.classmeth([arguments])`
  Das ist allerdings, ebenso wie bei Klassenvariablen, nicht üblich und wird von manchen Übersetzern ebenfalls mit einer Warnung quittiert. Auch hier wird das Objekt `object` gar nicht verwendet.

- Beispiele:

```
Account.numberOfAccounts()      // Aufruf über Klasssenname
a.numberOfAccounts()            // Aufruf über Objekt der Klasse
```

- Innerhalb von Klassenmethoden gibt es kein aktuelles Objekt `this` (selbst wenn eine Klassenmethode formal über ein Objekt der Klasse aufgerufen wird), d. h. die explizite oder implizite Verwendung von `this` in Klassenmethoden führt zu einem Fehler.

4.6.3 Hauptmethode `main`

- Wenn eine Klasse eine öffentliche statische Methode mit dem Namen `main`, einem einzelnen Parameter des Typs `String []` und Resultattyp `void` besitzt, kann diese Klasse als Programm ausgeführt werden.
- Beim Aufruf des Programms wird die Klasse zunächst geladen und initialisiert (vgl. § 4.8) und anschließend die o. g. Methode `main` aufgerufen.
- Die beim Aufruf des Programms eventuell angegebenen Kommandoargumente sind über den Parameter der o. g. Methode `main` verfügbar.
- Der Name `main` kann wie jeder andere Methodenname verwendet und ggf. auch überladen werden, d. h. eine Klasse kann auch Methoden mit dem Namen `main` und anderen Signaturen besitzen (die dann aber nicht, wie oben beschrieben, als Programm ausgeführt werden können).

4.6.4 Resultatwert von Methoden

- Eine Methode, deren Resultattyp nicht `void` ist, muss unter allen Umständen mit `return` einen passenden Resultatwert zurückliefern (oder eine Ausnahme werfen).
- Das heißt, eine solche Methode darf unter keinen Umständen das normale Ende ihres Rumpfs erreichen.
- Wenn der Übersetzer dies (mit seinem begrenzten Wissen) nicht zweifelsfrei beweisen kann, signalisiert er einen Fehler.
- Beispiel:

```
public static String signAsString (double x) {
  // Math.signum liefert -1.0/0.0/+1.0, wenn x>0/x==0/x<0 ist.
  switch ((int)Math.signum(x)) {
  case -1: return "-";
  case  0: return "";
  case +1: return "+";
  }
}
```

Obwohl der kontrollierende Ausdruck der `switch`-Anweisung tatsächlich nur die Werte `-1`, `0` und `+1` besitzen kann, ist die Methode fehlerhaft, weil ein Ausdruck mit Typ `int` aus Sicht des Übersetzers natürlich auch andere Werte besitzen kann und in diesen Fällen dann keine `return`-Anweisung ausgeführt werden würde.

4.6.5 Variadische Parameter

- Der letzte Parameter einer Methode oder eines Konstruktors kann durch Auslassungspunkte (`...`) zwischen seinem Typ `T` und seinem Namen *variadisch* deklariert werden. (Die Methode oder der Konstruktor wird dann auch als variadisch bezeichnet.)

- Beim Aufruf der Methode oder des Konstruktors können dann beliebig viele (auch gar keine) Werte des Typs `T` übergeben werden, die vom Übersetzer automatisch zu einer Reihe mit Typ `T []` zusammengefasst werden.

- Dementsprechend besitzt der variadische Parameter tatsächlich den Typ `T []` und kann deshalb wie eine Reihe verwendet werden, um auf die übergebenen Werte zuzugreifen.

- Deshalb kann bei einem Aufruf anstelle von beliebig vielen Werten des Typs `T` auch eine einzelne Reihe des Typs `T []` übergeben werden.
  (Damit kann eine variadische Methode ihren variadischen Parameter direkt an eine andere variadische Methode weitergeben.)

- Der Parameter der Hauptmethode `main` (vgl. § 4.6.3) kann dementsprechend entweder als normaler Parameter mit Typ `String []` oder als variadischer Parameter mit Typ `String` und Auslassungspunkten definiert werden.

Beispiel

```
// Summe aller Werte xs berechnen.
public static double sum (double ... xs) {
  double s = 0;
  for (double x : xs) s += x;
  return s;
}

// Durchschnittswert aller Werte x und xs berechnen.
public static double avg (double x, double ... xs) {
  // Weitergabe des variadischen Parameters xs mit Typ double []
  // an die variadische Methode sum.
  return (x + sum(xs)) / (1 + xs.length);
}

// Mögliche Aufrufe von sum und avg:
sum();
sum(1);                    avg(1);
sum(1, 2);                 avg(1, 2);
sum(1, 2, 3);              avg(1, 2, 3);
// Usw.
double [] a = { 4, 5, 6 };
sum(a);                    avg(0, a);
```

4.7 Konstruktoren

- Konstruktoren sind spezielle Objektmethoden, die zur Initialisierung neuer Objekte aufgerufen werden (vgl. § 4.9).
- Der Name eines Konstruktors muss mit dem Namen der Klasse übereinstimmen, zu der er gehört.
- Konstruktoren besitzen keinen Resultattyp (nicht einmal `void`).
- Innerhalb eines Konstruktors kann das zu initialisierende Objekt wie in einer gewöhnlichen Objektmethode über das Schlüsselwort `this` angesprochen werden. (Das heißt, das Objekt ist bereits erzeugt und muss nur noch initialisiert werden.)
- Konstruktoren können – wie andere Methoden – überladen werden.
- Die *erste* Anweisung eines Konstruktorrumpfs kann ein expliziter Aufruf eines anderen Konstruktors der Klasse sein, wobei hier als „Methodenname“ nicht der Klassenname, sondern das Schlüsselwort `this` verwendet wird:
  `this([arguments])`
  Der so aufgerufene Konstruktor kann auf die gleiche Weise einen weiteren Konstruktor der Klasse aufrufen, solange dadurch keine zyklischen Aufrufe entstehen.
  In der Argumentliste eines solchen Konstruktoraufrufs darf das aktuelle Objekt `this` weder direkt noch indirekt verwendet werden, da es noch nicht initialisiert ist.

- Ansonsten können Konstruktoren nur im Zusammenhang mit Objekterzeugungsausdrücken aufgerufen werden (vgl. § 4.9).
- Beispiele:

```
public Account (String h) {
  holder = h;
}

public Account (String h, int b) {
  this(h);
  balance = b;
}
```

- Ein Konstruktorrumpf kann `return`-Anweisungen ohne Ausdruck enthalten, um die Ausführung vorzeitig zu beenden.
- Wenn eine Klasse keinen expliziten Konstruktor enthält, besitzt sie implizit einen öffentlichen parameterlosen Konstruktor mit leerem Rumpf.
- Um dies zu verhindern, kann man einen privaten parameterlosen Dummy-Konstruktor definieren.

4.8 Initialisierungsblöcke

4.8.1 Objektinitialisierer (nicht-statische Initialisierungsblöcke)

- Objektinitialisierer sind Anweisungsblöcke innerhalb einer Klasse, die bei jeder Initialisierung eines neuen Objekts der Klasse ausgeführt werden (vgl. § 4.9).

- Sie sind nützlich, um Initialisierungscode zu formulieren, der von allen Konstruktoren einer Klasse ausgeführt werden soll, nicht von Konstruktorparametern abhängt und nicht (vernünftig) mit Initialisierungsausdrücken von Objektvariablen ausgedrückt werden kann.

- Wie in einem Konstruktor, kann das zu initialisierende Objekt über das Schlüsselwort `this` angesprochen werden.

- Beispiel:

```
{
  System.out.println("Initialisierung eines neuen Kontos");
}
```

4.8.2 Klasseninitialisierer (statische Initialisierungsblöcke)

- Klasseninitialisierer sind Anweisungsblöcke innerhalb einer Klasse, die einmalig beim Laden/Initialisieren der Klasse ausgeführt werden (d. h. normalerweise zu Beginn der Programmausführung).

- Sie sind nützlich, um Initialisierungscode zu formulieren, der nicht (vernünftig) mit Initialisierungsausdrücken von Klassenvariablen ausgedrückt werden kann.

- Beim Laden/Initialisieren einer Klasse werden die Klasseninitialisierer und die Initialisierungsausdrücke von Klassenvariablen der Reihe nach ausgeführt bzw. ausgewertet.

- Beispiel:

```
static {
  System.out.println("Initialisierung der Klasse Account");
}
```

4.9 Erzeugung und Initialisierung von Objekten einer Klasse

- Objekte einer Klasse werden durch Ausführung eines Objekterzeugungsausdrucks erzeugt, der aus dem Schlüsselwort `new`, dem Namen der Klasse und einer (eventuell leeren) Konstruktorargumentliste besteht: `new classname ([arguments])`

- Zum Beispiel:

```
Account a = new Account("Heinlein");
a = new Account("Heinlein", 1000);
```

- Ein Objekterzeugungsausdruck wird wie folgt ausgewertet:
  1. Es wird Platz für das neue Objekt beschafft.
     Falls dies nicht möglich ist, wird eine Ausnahme des Typs `OutOfMemoryError` geworfen.
  2. Alle Objektvariablen des Objekts werden mit typabhängigen Standardwerten vorinitialisiert (vgl. § 3.2.3).
  3. Die Konstruktorargumente werden (von links nach rechts) ausgewertet.
  4. Die Objektinitialisierer der Klasse und die Initialisierungsausdrücke von Objektvariablen werden der Reihe nach ausgeführt bzw. ausgewertet.
  5. Der passende Konstruktor wird ausgeführt.
     - Wenn er als erstes via `this` einen anderen Konstruktor aufruft, wird dieser ausgeführt, nachdem seine Argumente (von links nach rechts) ausgewertet wurden.
     - Andernfalls bzw. anschließend wird der (verbleibende) Konstruktorrumpf ausgeführt.
  6. Als Resultat des Objekterzeugungsausdrucks wird eine Referenz auf das neue Objekt geliefert.

4.10 Freigabe von Objekten

- Mit `new` erzeugte (Klassen- und Reihen-) Objekte können und müssen nicht explizit freigegeben werden.

- Objekte, die vom Programm nicht mehr benötigt werden, weil sie nicht mehr *erreichbar* sind, werden von Zeit zu Zeit automatisch freigegeben (automatische Speicherbereinigung, garbage collection).

- Ein Objekt ist *erreichbar*, wenn es direkt oder indirekt erreichbar ist.

- Ein Objekt ist *direkt erreichbar*, wenn es von einer Klassenvariablen, einer lokalen Variablen (vgl. § 4.11) oder einem Parameter eines Konstruktors, einer Methode oder eines `catch`-Blocks (vgl. § 8.6.2) referenziert wird.

- Ein Objekt ist *indirekt erreichbar*, wenn es von einer Objektvariablen eines erreichbaren Klassenobjekts oder einem Element eines erreichbaren Reihenobjekts referenziert wird.

4.11 Lokale Variablen und Parameter

- Lokale Variablen sind Variablen, die in einem Anweisungsblock, im Initialisierungsteil einer `for`-Schleife (vgl. § 3.4.4) oder im Ressourcenteil einer `try`-Anweisung (vgl. § 8.6) deklariert werden, d. h. direkt oder indirekt in einer Methode, einem Konstruktor oder einem Initialisierungsblock.

- Eine lokale Variable oder ein Parameter *verdeckt* (engl. *shadows*) eine gleichnamige Objekt- oder Klassenvariable der umgebenden Klasse (die ggf. mittels `this.objvar` bzw. `classname.classvar` angesprochen werden kann).

- Eine lokale Variable (oder ein Parameter eines `catch`-Blocks; vgl. § 8.6.2) darf aber keine weiter außen deklarierten lokalen Variablen oder Parameter verdecken.

- Anders als Objekt- und Klassenvariablen (vgl. § 4.5.1 und § 4.5.2), werden lokale Variablen *nicht* mit Standardwerten vorinitialisiert, sondern müssen unter allen Umständen vor ihrer ersten Verwendung einen Wert erhalten. Wenn der Übersetzer dies (mit seinem begrenzten Wissen) nicht zweifelsfrei beweisen kann, signalisiert er einen Fehler (den man z. B. durch explizite Initialisierung mit einem Dummywert vermeiden kann).

- Wenn bei der Deklaration einer lokalen Variablen anstelle ihres Typs das Schlüsselwort `var` verwendet wird, wird ihr Typ automatisch aus dem Typ ihrer Initialisierung ermittelt, die in diesem Fall vorhanden sein muss.

Unveränderliche lokale Variablen und Parameter

- Wenn eine lokale Variable `final` deklariert ist, muss der Übersetzer zweifelsfrei beweisen können, dass sie höchstens einmal – entweder durch ihre Initialisierung oder durch eine Zuweisung – einen Wert erhält. (Außerdem gilt die oben genannte Regel, dass sie garantiert vor ihrer ersten Verwendung ihren Wert erhalten muss.)

- Da ein Parameter immer automatisch durch ein Aufrufargument initialisiert wird, darf an einen `final` deklarierten Parameter niemals zugewiesen werden.

- Wenn eine lokale Variable nicht `final` deklariert ist, aber trotzdem die zuvor genannten Bedingungen erfüllt, gilt sie als *faktisch unveränderlich* (effectively final). (Das heißt: Durch das Entfernen bzw. Hinzufügen des Schlüsselworts `final` wird aus einer unveränderlichen eine faktisch unveränderliche Variable und umgekehrt.) Dasselbe gilt für Parameter.

- In bestimmten Kontexten (z. B. in anonymen Klassen und Lambda-Ausdrücken) dürfen nur (faktisch) unveränderliche Variablen und Parameter verwendet werden.

4.12 Namenskonventionen

- In der „Java Community“ gelten die folgenden Namenskonventionen, d. h. Regeln, die üblicherweise befolgt werden, obwohl sie nicht verpflichtend sind:
  - Namen von Klassen und Schnittstellen (vgl. § 6.2) beginnen mit einem Großbuchstaben (z. B. `Account`), alle anderen Namen mit einem Kleinbuchstaben (z. B. `number`, `deposit`, `i`).
  - Bei Namen, die aus mehreren Wörtern zusammengesetzt sind, beginnen das zweite und alle folgenden Wörter mit einem Großbuchstaben (z. B. `LimitedAccount`, `nextNumber`).
- Für Konstanten, d. h. unveränderliche Klassenvariablen, gelten abweichend folgende Regeln:
  - Die Namen bestehen komplett aus Großbuchstaben (z. B. `PI`).
  - Bei zusammengesetzten Namen werden die einzelnen Wörter durch Unterstriche getrennt (z. B. `FIRST_NUMBER`).
- Klassen und Schnittstellen werden normalerweise mit Substantiven bezeichnet (z. B. `Account`), Methoden mit Verben (und ggf. zugehörigen Substantiven, z. B. `deposit`, `printName`), Variablen mit Substantiven (z. B. `number`) oder Adjektiven (z. B. `empty`).

4.13 Beispiel: Einfach verkettete Liste

```
class List {
  // Unterklassenöffentliche Objektvariablen
  // (protected statt private, damit die Variablen später
  // in Unterklassen verwendet werden können):
  protected int head;   // Erstes Element (Kopf).
  protected List tail;  // Restliste (Schwanz).

  // Öffentliche Konstruktoren:
  // Liste mit erstem Element h und ggf. Restliste t initialisieren.
  public List (int h, List t) {
    head = h;
    tail = t;
  }
  public List (int h) {
    this(h, null);
  }
```

```
  // Öffentliche Objektmethoden:
  // Erstes Element und Restliste abfragen.
  // (Eine Methode kann den gleichen Namen wie ein Feld besitzen.)
  public int head () { return head; }
  public List tail () { return tail; }

  // Öffentliche Objektmethode:
  // Länge (d. h. Anzahl der Elemente) ermitteln.
  public int length () {
    int n = 1;
    for (List p = tail; p != null; p = p.tail) n++;
    return n;
  }

  // Öffentliche Objektmethode: Liste ausgeben.
  public void print () {
    for (List p = this; p != null; p = p.tail) {
      System.out.println(p.head);
    }
  }
}
```

```
class Test {
  // Testprogramm.
  public static void main (String [] args) {
    // Liste ls erzeugen, die als Elemente die Werte der
    // Kommandoargumente in umgekehrter Reihenfolge enthält.
    List ls = null;
    for (int i = 0; i < args.length; i++) {
      ls = new List(Integer.parseInt(args[i]), ls);
    }

    // Liste ausgeben.
    System.out.println(ls.length());
    ls.print();
  }
}
```

Beispielaufruf: `java Test 1 2 3`