Objektorientierte Programmierung

Kapitel 8: Statische Methoden

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2018/19

http://www.informatik.uni-halle.de/~brass/oop18/

Inhalt

Motivation (1)

Beherrschung von Komplexität:

- Programme können lang und kompliziert werden.
- Ein Programm sollte aus kleineren Einheiten aufgebaut sein, die man unabhängig verstehen kann.
- Dazu müssen diese kleineren Einheiten eine gut dokumentierte und möglichst kleine/einfache Schnittstelle zum Rest des Programms haben.

  D.h. die Interaktion mit dem Rest des Programms muss auf wenige, explizit benannte Programmelemente beschränkt sein.
- Es gibt eine solche Strukturierung des Programms auf unterschiedlichen Ebenen. Methoden/Prozeduren bilden die erste Ebene oberhalb der einzelnen Anweisungen.

  Weitere Ebenen sind z.B. Klassen und Packages.

Motivation (2)

Vermeidung von "Copy&Paste Programmierung":

- Oft braucht man in einem Programm an verschiedenen Stellen ein gleiches Stück Programmcode.
- Um die Lesbarkeit und Änderbarkeit des Programms zu verbessern, sollte dieser Programmcode nur einmal aufgeschrieben werden.
- Das ist mit Methoden/Prozeduren möglich:
  - Man kann einem Stück Programmcode einen Namen geben, und
  - es dann an verschiedenen Stellen im Programm aufrufen (ausführen lassen).

Motivation (3)

Methoden/Prozeduren als Mittel zur Abstraktion:

- Methoden/Prozeduren sind ein Mittel der Abstraktion (man interessiert sich nicht mehr für bestimmte Details, vereinfacht die Realität). Hier
  - spielt es für den Benutzer/Aufrufer keine Rolle, wie die Methode/Prozedur ihre Aufgabe löst, sondern nur
  - was sie genau macht.
- Dadurch, dass man sich Methoden/Prozeduren definiert, erweitert man gewissermaßen die Sprache:
  - Man kann sich die Methoden/Prozedur-Aufrufe wie neue, mächtigere Befehle vorstellen.
  - Dadurch kann man Algorithmen auf höherer Abstraktionsebene beschreiben.

Motivation (4)

Methoden zur Dokumentation der Programmentwicklung:

- In Methoden kann man wieder Methoden aufrufen.
- "Bottom-up" Konstruktion eines Programms:
  von unten (Java) nach oben (Aufgabe).

  Beginnend mit dem von Java vorgegebenen Sprachumfang definiert man sich nach und nach mächtigere Methoden, die der vorgegebenen Aufgabenstellung immer näher kommen, bis die Methode "`main`" sie schließlich löst.
- "Top-down" Konstruktion eines Programms:
  von oben (Aufgabe) nach unten (Java).

  Man unterteilt die Aufgabenstellung immer weiter in Teilaufgaben, bis diese so klein sind, dass sie sich direkt lösen lassen (schrittweise Verfeinerung).
- Es ist günstig, wenn die Denkstrukturen bei der Entwicklung sich auch im fertigen Programm wiederfinden.

Motivation (5)

Methoden als Mittel zur Wiederverwendung von Programmcode:

- Methoden werden selbst wieder zu Klassen, Packages, und Bibliotheken zusammengefasst.
- Im Rahmen dieser größeren Einheiten werden Methoden zum Teil auch in anderen Programmen wiederverwendet:
  - Es muss nicht jeder das Rad neu erfinden.
  - Man kann Zeit und Geld sparen, indem man von Anderen entwickelten Programmcode verwendet.

    Sofern er gut dokumentiert und möglichst fehlerfrei ist.
- Wiederverwendung (engl. “Reuse”) von Programmcode war eines der großen Ziele der Objektorientierung.

  Und sollte zur Lösung der Softwarekrise beitragen.

Methoden, Prozeduren, Funktionen

- Im wesentlichen sind “Methode”, “Prozedur”, “Funktion”, “Unterprogramm”, “Subroutine” synonym.

  In Pascal gibt es Prozeduren und Funktionen: Prozeduren liefern keinen Wert zurück, der Prozeduraufruf ist ein Statement. Funktionen liefern einen Wert und werden in Expressions benutzt. In C/C++ gibt es nur Funktionen, die aber den leeren Typ `void` als Ergebnistyp haben können (wenn sie keinen Rückgabewert liefern).

- Methoden sind in Klassen deklarierte Funktionen/Prozeduren. Java hat nur Methoden.

  C++ hat auch Funktionen außerhalb von Klassen (zur Kompatibilität mit C).

- Statische Methoden entsprechen klassischen Funktionen.

  Z.B. ist `sin` eine statische Methode in der Klasse `Math`. Im Kapitel 11 werden wir auch nicht-statische Methoden besprechen. Sie haben Zugriff auf die Attribute eines “aktuellen Objektes”, für das sie aufgerufen wurden.

Beispiel, Grundbegriffe (1)

```
(1)  class Quadratzahlen {
(2)
(3)      // square(n) liefert n^2
(4)      static int square(int n) {
(5)          return n * n;
(6)      }
(7)
(8)      // Hauptprogramm:
(9)      public static void main(String[] args) {
(10)         for(int i = 1; i <= 20; i++) {
(11)             System.out.println(i
(12)                 + " zum Quadrat ist "
(13)                 + square(i));
(14)         }
(15)     }
(16) }
```

Beispiel, Grundbegriffe (2)

- Der Abschnitt:

```
static int square(int n)  // Methodenkopf
{ return n * n; }         // Methodenrumpf
```

  ist die Definition einer (statischen) Methode:

  - Die Methode heißt `square`.
  - Sie hat einen Parameter vom Typ `int`, der `n` heißt.
  - Sie liefert einen `int`-Wert.

    Wie bei Variablen steht der Typ (hier Ergebnistyp, Rückgabetyp) vor dem Namen (der Methode).

  - Der Ergebniswert berechnet sich als `n * n`.

    Die Anweisung heisst `return`, weil man zu der Stelle im Programm zurückkehrt, an der die Methode aufgerufen wurde. Die Ausführung des Methodenrumpfes wird damit beendet.

Beispiel, Grundbegriffe (3)

- Die Definition einer Methode alleine tut nichts.

  Der Compiler erzeugt natürlich Maschinencode.

- Erst durch den Aufruf

  ```
  square(i)
  ```

  wird der Rumpf der Methode ausgeführt.

  Ein guter Compiler sollte eine Warnung ausgeben, wenn Methoden definiert werden, die nicht ausgeführt werden ("toter Code"). In Java würde das aber höchsten für private Methoden gelten — der Compiler weiß ja nichts über die Verwendung der erzeugten `class`-Datei.

- Der Wert von `i`, z.B. 2, ist der aktuelle Parameter.

  In der Wikipedia steht, dass "aktueller Parameter" eine falsche Übersetzung von "actual parameter" ist. Korrekt wäre "tatsächlicher Parameter". "actual" kann aber auch "gegenwärtig" bedeuten.

Beispiel, Grundbegriffe (4)

- Beim Aufruf findet die Parameterübergabe statt.
  Dabei wird der formale Parameter `n` an den aktuellen Parameter (z.B. 2) gebunden.
- Dies wirkt wie eine Zuweisung: `n = 2;`
- Anschließend wird der Rumpf der Methode ausgeführt.
- Die Anweisung

  ```
  return n * n;
  ```

  beendet die Ausführung der Methode und legt den Ergebniswert (Rückgabewert, Funktionswert, Ausgabewert) fest.
- Im Beispiel liefert die Methode 4 (Rückgabewert).

Beispiel, Grundbegriffe (5)

- Der Aufruf einer Methode ist ein Wertausdruck (Expression).
  - Die Eingabewerte (aktuelle Parameter) der Methode können mit beliebig komplexen Wertausdrücken berechnet werden.

    Die Wertausdrücke selbst nennt man Argumente (des Methodenaufrufs). Die Sprechweise ist aber nicht ganz einheitlich.
  - Der Rückgabewert der Methode kann selbst in einem komplexen Wertausdruck weiter verwendet werden.

    Im Beispiel ist der Rückgabewert selbst Eingabe des Operators + (Stringkonkatenation bzw. zuerst Umwandlung nach `String`).
- Parameter (ohne Zusatz) meint "formaler Parameter".

  Statt "aktueller Parameter" kann man auch Parameterwert sagen.

Beispiel, Grundbegriffe (6)

- Eine Methode kann mehr als einen Parameter haben:

```
static double power(double x, int n)
{
        double result = 1.0;
        for(i = 1; i <= n; i++)
                result = result * x;
        return result;
}
```

- Aktuelle und formale Parameter werden über ihre Position verknüpft, beim Aufruf `power(2.0, 3)` bekommt
  - `x` (1. Parameter) den Wert `2.0` und
  - `n` (2. Parameter) den Wert 3.

Beispiel, Grundbegriffe (7)

- Anzahl und Typ der aktuellen Parameter muss zu den deklarierten formalen Parametern passen.
- Zum Beispiel wäre folgender Aufruf falsch:

  ```
  square(2, 4)
  ```

  `square` ist mit nur einem Parameter deklariert.

  Selbstverständlich erkennt der Compiler diesen Fehler und gibt eine entsprechende Fehlermeldung aus: "`square(int) in Quadratzahlen cannot be applied to (int,int)`".
- Bei der Parameterübergabe finden fast die gleichen Typ-Umwandlungen wie bei einer Zuweisung statt.

  Mit Ausnahme der Spezialbehandlung von konstanten Ausdrücken. Die Zahl `0` als Argument (Typ `int`) kann also (ohne explizite Typumwandlung) nur für einen Parameter vom Typ `int`, `long`, `float`, `double` verwendet werden (sowie auch "Wrapper-Klassen" wie `Integer`), aber z.B. nicht für `byte`.

Wichtiger Hinweis

- Sie müssen immer die Aufgabenstellung genau einhalten:
  - Es ist ein großer Unterschied, ob die Methode einen Parameterwert bekommt, oder selbst von Tastatur liest.
  - Entsprechend ist es wesentlich, ob die Methode das Ergebnis selbst ausdrucken soll, oder als Funktionswert zurückliefern.
  - Es ist auch wichtig, ob die Methode (z.B. im Fehlerfall) nur sich selbst beenden soll, oder das ganze Programm mit `System.exit(`*n*`)` abbrechen.

    Entsprechend müssen auch Exceptions abgesprochen sein.
  - Entsprechend (s.u.): Globale Variable vs. Parameter.
- Der Aufrufer ist sozusagen der Kunde (Auftraggeber), der Programmierer der Methode der Auftragnehmer.

Methoden ohne Rückgabewert: `void` (1)

- Methoden müssen nicht unbedingt einen Wert zurückgeben.
- Dann schreibt man anstelle des Ergebnistyps das Schlüsselwort "`void`":

```
static void printHello(String name) {
        System.out.println("hello, " + name);
}
```

- Formal zählt in Java "`void`" nicht als Typ.

  Aber es kann als Rückgabetyp von Methoden verwendet werden. Dagegen kann man keine Variablen vom Typ `void` deklarieren (macht auch keinen Sinn).

- Natürlich kann man den leeren Ergebniswert nicht verwenden:

```
printHello("world"); // ok
int n = printHello("Stefan"); // falsch
```

Methoden ohne Rückgabewert: `void` (2)

- Eine Methode mit "Ergebnistyp" `void` braucht keine `return`-Anweisung:
  - Falls die Ausführung bis zur schließenden Klammer `}` durchläuft, kehrt sie automatisch zum Aufrufer zurück.

    Ganz am Ende steht sozusagen implizit eine `return;`-Anweisung.
  - Man darf allerdings `return;` verwenden, um die Ausführung vorzeitig abzubrechen.
- Z.B. ist `main` eine Methode mit "Ergebnistyp" `void`. Deswegen war die `return`-Anweisung bisher nicht nötig.
- Aber praktisch: Fehler-/Sonderfälle am Anfang der Methode behandeln und Ausführung jeweils mit `return;` beenden.

  Anschließend hat man dann den Kopf frei für die eigentliche Aufgabe, und steckt nicht in einer tiefen `if`/`else`-Struktur.

Methoden ohne Rückgabewert: void (3)

```
(1)  import java.util.Scanner;
(2)  class Quadratzahlen {
(3)      public static void main(String[] args) {
(4)          Scanner s = new Scanner(System.in);
(5)          System.out.print("Quadratzahlen bis: ");
(6)          n = s.nextInt();
(7)          if(n <= 0) {
(8)              System.out.println("Fehler: ...");
(9)              return;
(10)         }
(11)         // Hierher kommt Ausfuehrung nur bei n>0
(12)         for(int i = 1; i <= n; i++) {
(13)             System.out.println(i * i);
(14)         }
(15)     }
(16) }
```

return in Methoden mit Ergebnis (1)

- Methoden mit anderen Ergebnistyp als `void` müssen immer mit einer "`return` ⟨`Wert`⟩`;`" Anweisung enden.

  Genauer: Jeder Ausführungspfad muss mit "`return` ⟨`Wert`⟩`;`" oder "`throw` ⟨`Exception`⟩`;`" enden. Durch das Auslösen einer Exception (Fehlerfall, siehe Kap. 9) wird die Ausführung der Methode abgebrochen.

- Die `return`-Anweisung kann auch in anderen Anweisungen (z.B. `if`/`else`) geschachtelt sein:

```
(1)  class Maximum {
(2)      public static int max(int n, int m) {
(3)          if(n > m)
(4)              return n;
(5)          else
(6)              return m;
(7)      }
(8)      ...
```

return in Methoden mit Ergebnis (2)

- Der Compiler muss verstehen können, dass die Ausführung der Methode auf jeden Fall mit einem return ⟨Wert⟩; endet. Dies geht z.B. nicht:

```
(1)  class Maximum {
(2)      public static int max(int n, int m) {
(3)          if(n > m)
(4)              return n;
(5)          if(m >= n)
(6)              return m;
(7)      }
(8)      ...
```

Der Compiler denkt beim `if` ohne `else` immer, dass es übersprungen werden könnte. Die Bedingungen schaut er sich nicht an. Deswegen vermutet er, dass die Ausführung bis Zeile (7) kommen kann und die Ausführung der Methode ohne definierten Rückgabewert endet.

return in Methoden mit Ergebnis (3)

- Aufgabe: Berechnet diese Methode korrekt das Maximum?

```
(1)  class Maximum {
(2)      public static int max(int n, int m) {
(3)          if(n > m)
(4)              return n;
(5)          return m;
(6)      }
(7)      ...
```

- Hinweis: Denken Sie daran, das `return` die Ausführung der Methode beendet (man kehrt zum Aufrufer zurück).

return in Methoden mit Ergebnis (4)

- Der Typ des gelieferten Wertes muss zum Ergebnistyp der Methode passen. Bei `void` kann man nur "nichts" liefern:
  - Ein einfaches `return;` ist in einer Methode mit echtem Rückgabetyp (nicht `void`) immer ein Syntaxfehler.
  - Entsprechend ist "`return` ⟨`Wert`⟩`;`" in einer Methode mit Ergebnis `void` immer falsch.
- Beispiel: Man kann nicht eine Zeichenkette liefern, wenn man als Ergebnistyp `int` angegeben hat:

```
(1)  class Fehler {
(2)      public static int test() {
(3)          return "abc"; // Fehler!
(4)      }
(5)      ...
```

Unerreichbarer Code

- Befehle, die niemals ausgeführt werden können, heißen "unerreichbarer Code". Sie sind offensichtlich sinnlos.
- Wenn der Compiler das bemerkt, gibt er eine Fehlermeldung aus: "`unreachable statement`".

  Wieder schaut er sich nicht die genauen Bedingungen an. Auch analysiert er z.B. nicht, ob bestimmte Methoden das Programm beenden.

- Beispiel:

```
(1)  class Fehler {
(2)      public static void main(String[] args) {
(3)          return;
(4)          System.out.println("Impossible!");
(5)      }
(6)  }
```

Aufruf von Methoden

- Statische Methoden werden normalerweise in der Form

  ⟨Klassenname⟩ . ⟨Methodenname⟩ (⟨Argumente⟩)

  aufgerufen.

  Es ist auch möglich, statt des Klassennamens einen Wertausdruck zu schreiben, der ein Objekt der Klasse liefert (das ist aber eher verwirrend und schlechter Stil). Der Wertausdruck wird ausgewertet (wichtig, falls er Seiteneffekte hat). Sein Ergebnis wird aber ignoriert, nur der Typ ist wichtig (genauer der statische Typ, der zur Compilezeit bestimmt wird, und nicht der eventuell speziellere Typ des tatsächlich berechneten Objektes).

- Für statische Methoden der eigenen Klasse reicht

  ⟨Methodenname⟩ (⟨Argumente⟩)

- Daher konnten wir in den obigen Beispielen den Klassennamen weglassen.

Gültigkeitsbereich von Methoden (1)

- Eine deklarierte Methode ist in der ganzen Klasse bekannt.
- Im Gegensatz zu lokalen Variablen ist also nicht verlangt, dass eine Methode vor ihrer Verwendung deklariert ist.
- Die Reihenfolge der Komponenten in einer Klassendeklaration spielt also keine Rolle.
- Die Anordnung ist eine Frage des Programmierstils:
  - Z.B. öffentlich zugreifbare Methoden wie `main` zuerst, und dann die privaten Hilfsfunktionen.

    Dann braucht jemand, der die Klasse von außen benutzen will, nur den Anfang zu lesen. Vielleicht nutzt er aber ohnehin eher `javadoc`.
  - Oder umgekehrt.

    Falls man systematisch Funktionen vor ihrer Verwendung definieren will.

Gültigkeitsbereich von Methoden (2)

- Folgendes ist also erlaubt (Aufruf vor Deklaration):

```
(1)  class FunktionswertTabelle {
(2)     public static void main(String[] args){
(3)        for(double x = 0.0; x < 1.05;
(4)              x += 0.1)
(5)           System.out.println(
(6)              x + ": " + f(x));
(7)     }
(8)
(9)     static double f(double x) {
(10)       return x * x + 2 * x + 1;
(11)    }
(12) }
```

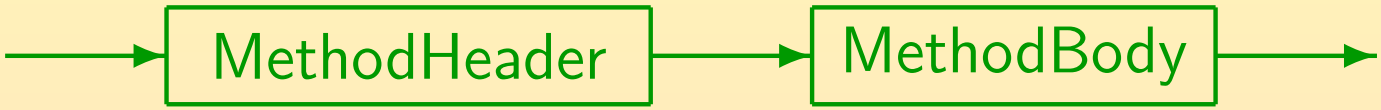
Methoden-Deklarationen: Übersicht

- Eine Methodendeklaration besteht aus:
  - Modifizierern wie `public` und `static`,
  - optionalen Typ-Parametern (siehe Kapitel 20),
  - dem Ergebnis-Datentyp der Methode (Rückgabe-Typ),
  - dem Namen der Methode (ein Identifier/Bezeichner),
  - eine "`(`"
  - der Liste formaler Parameter (kann auch leer sein),
  - eine "`)`"
  - optional einer `throws`-Klausel (siehe Kapitel 9), und
  - dem Rumpf der Methode, einem Block.

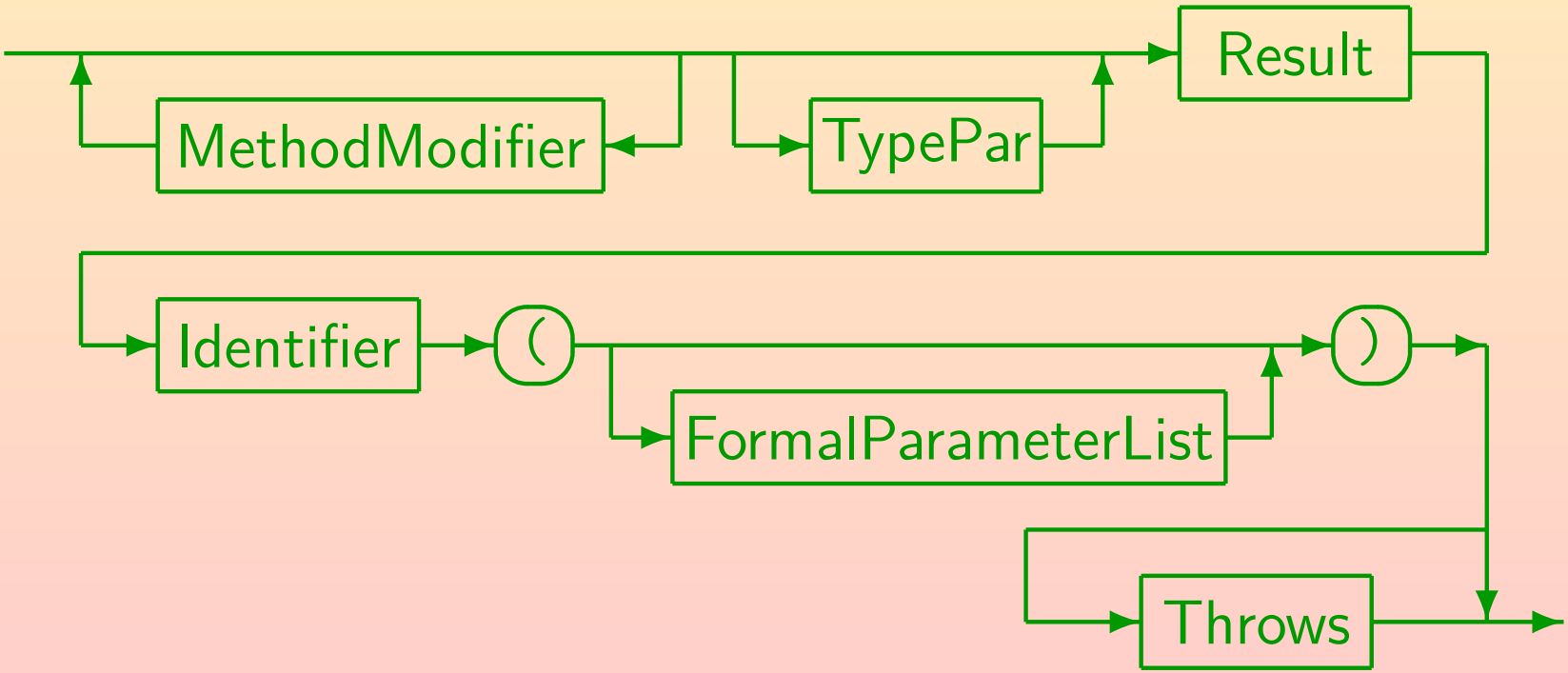
    Oder einfach "`;`" bei abstrakten Methoden (siehe Kapitel 12).

Methoden-Deklarationen: Syntax (1)

- MethodDeclaration:

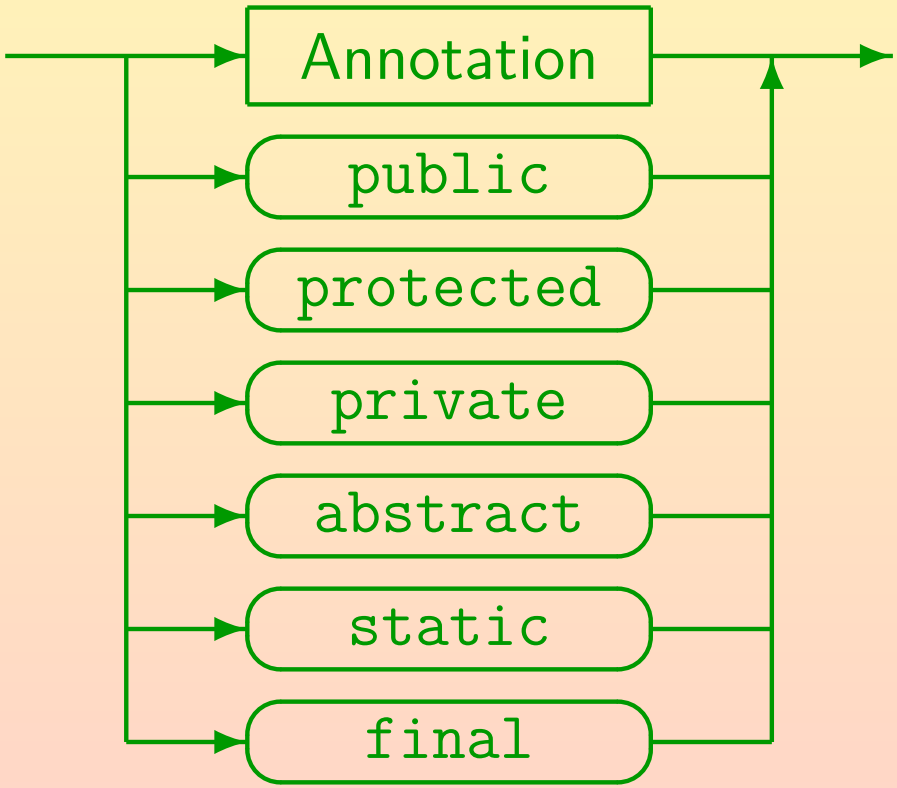


- MethodHeader:



Methoden-Deklarationen: Syntax (2)

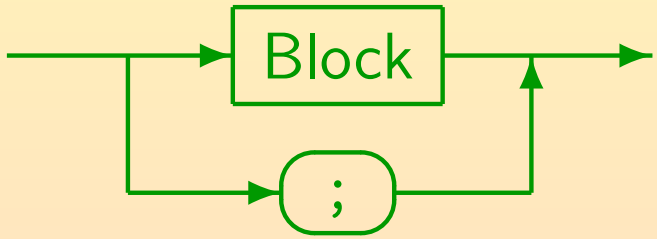
- MethodModifier:



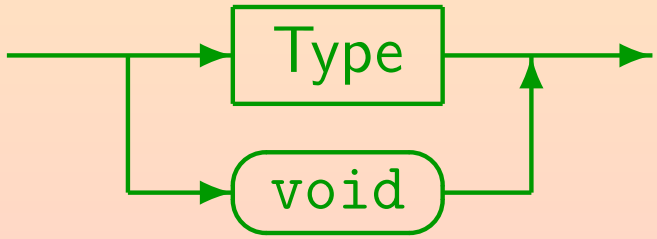
Bisher wurden nur die Modifier `public` und `static` verwendet. Die übrigen hier gezeigten Modifier werden in späteren Kapiteln besprochen. Es gibt noch weitere, die in der Vorlesung nicht behandelt werden: `synchronized`, `native`, `strictfp`.

Methoden-Deklarationen: Syntax (3)

- MethodBody:



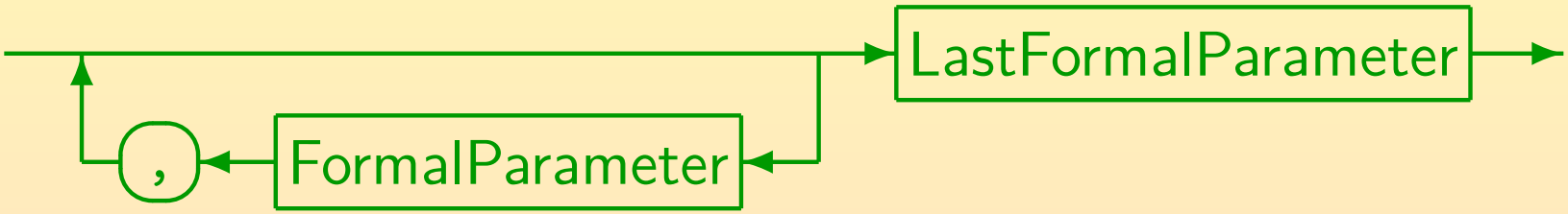
- Result ("ResultType"):



Formal ist `void` kein Typ, kann aber hier anstelle eines Typs verwendet werden, um anzuzeigen, dass die Methode keinen Ergebniswert liefert.

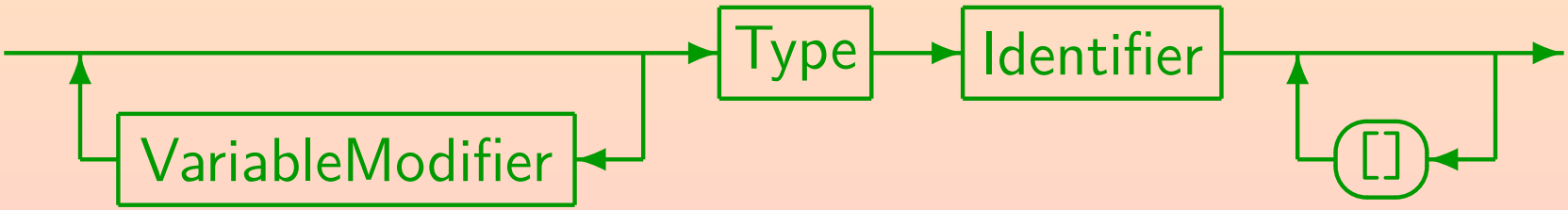
Methoden-Deklarationen: Syntax (4)

- FormalParameterList:



Der letzte formale Parameter wird getrennt behandelt, weil man Methoden mit variabler Argument-Anzahl deklarieren kann (siehe Kapitel 16), dann wird ihm ein Array mit allen übrigen Argumenten zugewiesen.

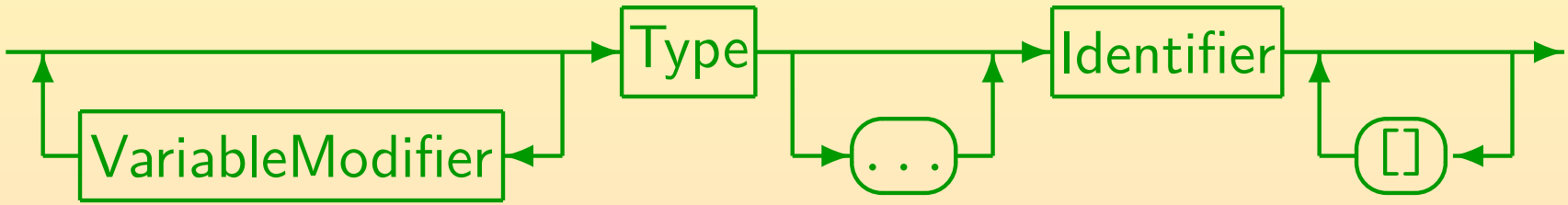
- FormalParameter:



Die Möglichkeit, den Arraytyp mit "[]" noch nach dem Parameternamen anzugeben, ist sollte Umsteigern von C++ entgegenkommen, wird in normalen Java-Programmen aber eher nicht verwendet.

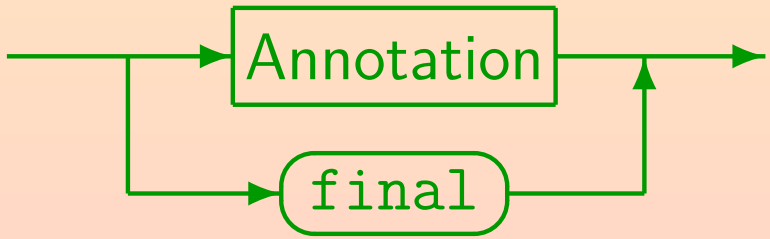
Methoden-Deklarationen: Syntax (5)

- LastFormalParameter:



Der Unterschied ist nur das optionale "`...`" nach dem Typ.

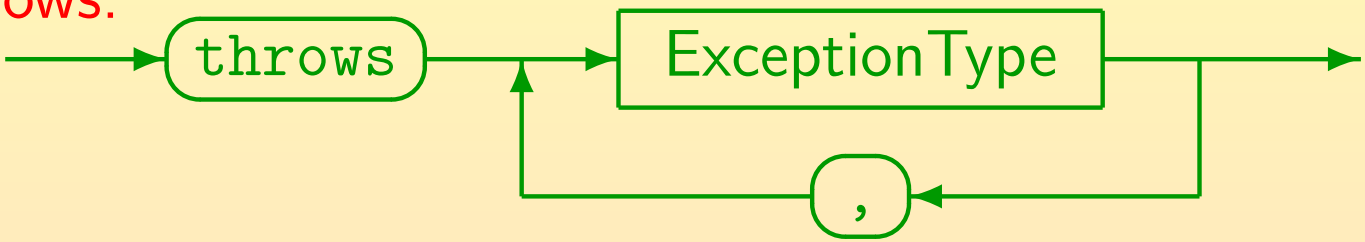
- VariableModifier:



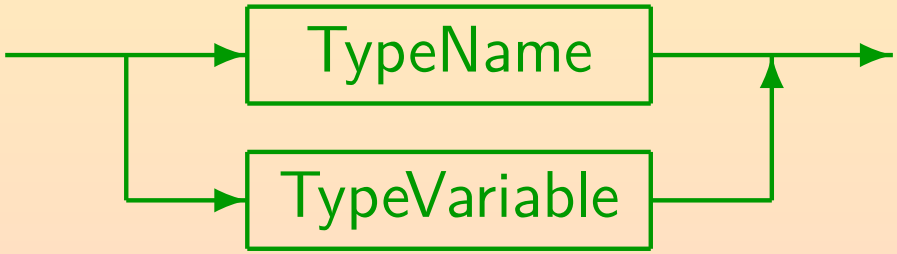
Die Angabe von `final` bewirkt, dass Zuweisungen an den Parameter im Methodenrumpf verboten sind. Annotationen werden in dieser Vorlesung nicht behandelt (außer "`@Override`" für Methoden in Kapitel 12).

Methoden-Deklarationen: Syntax (6)

- Throws:

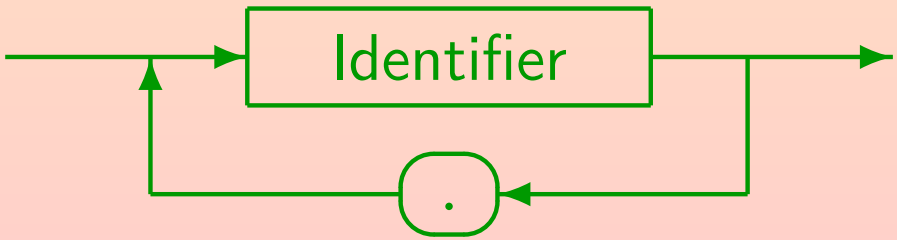


- ExceptionType:



Typ-Variablen werden erst in Kapitel 20 behandelt.

- TypeName:



Der Typ muss ein Untertyp von `Throwable` sein (siehe Kap. 9, 12, 18).

Inhalt

Parameterübergabe (1)

- Java benutzt "call by value" zur Parameterübergabe:
  - Selbst wenn das Argument im Methodenaufruf eine Variable ist, wird der Wert dieser Variable an den formalen Parameter zugewiesen (d.h. kopiert).
  - Auf die Variable, die als Argument angegeben wurde, hat man in der Methode/Prozedur keinen Zugriff.
  - Auch eine Zuweisung an den Parameter bewirkt nach außen nichts (siehe Beispiel, nächste Folie).
- Das ist passend für Eingabeparameter, bei denen also ein Wert vom Aufrufer in die Methode/Prozedur hinein fließt.
- Zur Ausgabe (Datenfluss von der Methode zum Aufrufer) dient in erster Linie der Rückgabewert (siehe aber unten).

Parameterübergabe (2)

```
(1)  class CallTest {
(2)
(3)      static int square(int n) {
(4)          int s = n * n;
(5)          n = 1000; // Ändert n, aber nicht i
(6)          return s;
(7)      }
(8)
(9)      public static void main(String[] args) {
(10)         for(int i = 1; i <= 20; i++) {
(11)             int q = square(i);
(12)             // Schleife läuft von 1 bis 20.
(13)             // Aufruf ändert i nicht.
(14)             System.out.println(i + ": " + q);
(15)         }
(16)     }
(17) }
```

Parameterübergabe (3)

- Wenn Objekte übergeben werden, wird nur die Referenz kopiert, aber nicht das Objekt selbst.
- Wenn die Methode Zugriff auf Attribute des Objektes hat, kann sie das Objekt ändern.
- Diese Änderung bleibt nach Rückkehr der Methode erhalten. Wenn der Aufrufer das Attribut abfragen kann, ist das eine Möglichkeit für zusätzliche "Ausgabeparameter".
- Da Arrays spezielle Objekte sind, gilt auch für Arrays:
  - Nur die Referenz wird bei der Parameterübergabe kopiert, nicht das Array selbst.
  - Ändert man das Array (einen Eintrag) in der Methode, so ist das Array hinterher auch für den Aufrufer verändert.

Parameterübergabe (4)

```
(1)  class CallTest2 {
(2)      static void test(int[] a) {
(3)          a[0] = 2;
(4)      }
(5)
(6)      public static void main(String[] args) {
(7)          int[] a = new int[1];
(8)          a[0] = 1;
(9)          System.out.println(a[0]); // 1
(10)         test(a); // a[0] wird geaendert!
(11)         System.out.println(a[0]); // 2
(12)     }
(13) }
```

Die Möglichkeit zur Änderung besteht natürlich nur bei Übergabe des ganzen Arrays, wie hier an `test`. Beim Aufruf `System.out.println(a[0])` wird normal ein `int` übergeben, so dass `a[0]` von der Methode nicht geändert werden kann.

Parameterübergabe (5)

- Selbstverständlich sollte die Dokumentation einer Methode sehr klar machen, wenn ein übergebenes Objekt oder Array geändert wird.
- Stilistisch wäre ein Rückgabewert (Funktionswert) meist besser.

  Die Änderung übergebener Objekte oder Arrays ist normalerweise nur interessant, wenn man mehrere Werte liefern muss (oder wenn es von der Anwendung her "sehr natürlich" ist).
- Wenn man Klassen entwirft, sollte man darüber nachdenken, ob man die Änderbarkeit von Attributen wirklich benötigt.

  Falls nicht, gibt es Möglichkeiten, das zu verhindern (siehe Kapitel 11). Solche (nicht änderbaren) Objekte kann man gefahrlos einer Methode übergeben, ohne Sorge, dass die Methode sie vielleicht unerwartet verändert.

Parameterübergabe (6)

- Es ist möglich, Parameter als "`final`" zu deklarieren, aber das bedeutet nur, dass direkte Zuweisungen an den Parameter verboten sind:

```
(1)  class CallTest3 {
(2)
(3)      static void test(final int[] a) {
(4)          a = new int[10]; // verboten!
(5)          a[0] = 2; // erlaubt
(6)      }
```

- Es ist zu empfehlen, Parameter in Methoden normalerweise nicht zu ändern.

  Es könnte günstig sein, den originalen Parameterwert auch am Ende der Methode noch zu haben. In Ausnahmefällen kann aber auch die Änderung des Parameterwertes sehr elegant sein.

Rückgabe mehrerer Werte: Übersicht

- Oft hat eine Methode nur einen Ausgabewert, den kann man dann als Rückgabewert verwenden.
- Falls es mehrere Ausgabewerte gibt, kann man der Methode änderbare Objekte (oder Arrays) übergeben, in die die Methode ihr Ergebnis speichert.
- Es ist auch möglich, dass die Methode ein Objekt/Array mit `new` erzeugt, und dieses als Rückgabewert liefert: Darin können dann beliebig viele Werte gespeichert werden.
- Exceptions erweitern auch die Möglichkeiten, mit der eine Methode Information zum Aufrufer übermitteln kann.

  Sie sind allerdings hauptsächlich für Fehlerfälle gedacht. Siehe Kap. 9.

Inhalt

Lokale Variablen (1)

- In Kapitel 7 wurde bereits erläutert, dass eine in einem Block deklarierte Variable von der Deklaration bis zum Ende dieses Blockes bekannt ist.

  Die in einem `for`-Statement deklarierte Variable ist nur dort bekannt.

- Der Rumpf einer Methode ist ein Block.
- In einer Methode deklarierte Variablen sind damit nur innerhalb der Methode bekannt.

  Daher der Name "lokale Variable": Lokal zu einer Methode/Block.

- Zwei verschiedene Methoden können zwei Variablen mit gleichem Namen deklarieren: Dies sind verschiedene Variablen, jede hat einen eigenen Wert.

  Eine Zuweisung an eine Variable würde den Wert der anderen nicht ändern, selbst wenn sie den gleichen Namen haben.

Lokale Variablen (2)

```
(1)  class Quadratzahlen {
(2)
(3)      public static void main(String[] args) {
(4)          int i;
(5)          for(i = 1; i <= 20; i++) {
(6)              int q = square(i);
(7)              System.out.println(i + "^2 = " + q);
(8)          }
(9)      } // Ende des Gültigkeitsbereiches von i
(10)
(11)     static int square(int n) {
(12)         return n * i; // Fehler: i unbekannt
(13)     } // Ende Gültigkeitsbereich von n
(14) }
```

Lokale Variablen (3)

- Im obigen Beispiel bekommt man die Fehlermeldung:

```
Quadratzahlen.java:12: cannot find symbol
symbol : variable i
location: class Quadratzahlen
        return i * n; // ...
               ^
1 error
```

- Der Gültigkeitsbereich von Parametern ist der ganze Rumpf der Methode.
- Gültigkeitsbereich heißt englisch “scope (of the declaration)”.

  Im Beispiel gibt es auch eine Variable “q”, die in einem geschachtelten Block deklariert ist: Ihr Gültigkeitsbereich endet entsprechend am Ende dieses Blockes, und nicht am Ende der Methode. Geschachtelte Blöcke werden unten noch näher diskutiert.

Lokale Variablen (4)

Aufgabe: Was gibt dieses Programm aus?

```
(1)  class Test {
(2)
(3)      public static void main(String[] args) {
(4)          int i = 3;
(5)          p(1);
(6)          System.out.println(i);
(7)      }
(8)
(9)      static void p(int n) {
(10)         int i;
(11)         i = 5 * n;
(12)         System.out.println(i);
(13)     }
(14) }
```

Statische Variablen/Attribute (1)

- Man kann Variablen auch außerhalb von Methoden deklarieren (aber natürlich innerhalb der Klasse).
- In diesem Kapitel gehen wir davon aus, dass diese Variablen mit dem Schlüsselwort "`static`" gekennzeichnet werden (so wie bisher auch Methoden):

```
private static int n = 5;
```

- Diese Variable ist dann von allen Methoden der Klasse aus zugreifbar.
- Durch das Schlüsselwort "`private`" ist sie von außerhalb der Klasse aus nicht zugreifbar.

  Der Zugriffsschutz wird später noch genauer besprochen. Man kann "`private`" weglassen (dann ist sie nur innerhalb des Paketes zugreifbar, aber auch von anderen Klassen) oder "`public`" schreiben (dann ist sie von überall zugreifbar).

Statische Variablen/Attribute (2)

```
(1)  class SVarDemo {
(2)      static int anzAufrufe = 0;
(3)
(4)      public static void main(String[] args) {
(5)          for(double x = 0; x < 1.05; x += 0.1)
(6)              System.out.println(poly(x));
(7)          System.out.println("Aufrufe: " +
(8)                             anzAufrufe);
(9)      }
(10)
(11)     static double poly(double x) {
(12)         anzAufrufe++;
(13)         return x * x + 2 * x + 1;
(14)     }
(15) }
```

Lokale vs. globale Variablen (1)

- In der nicht-objektorientierten Programmierung gab es (etwas vereinfacht):
  - lokale Variablen (innerhalb einer Prozedur/Funktion) und
  - globale Variablen (außerhalb von Prozeduren).
- Probleme globaler Variablen:
  - Namenskonflikte bei großen Programmen

    Zwei verschiedene Programmierer konnten zufällig eine Variable gleichen Namens einführen.
  - Die Interaktion von Prozeduren/Funktionen mit ihnen kann recht undurchschaubar werden.

    Globale Variablen sind von allen Prozeduren/Funktionen aus zugreifbar, ohne dass dies im Prozedurkopf deklariert werden muss. Man kann nur hoffen, dass der Programmierer es in der Dokumentation erwähnt.

Lokale vs. globale Variablen (2)

- In Java gibt es keine globalen Variablen.
- Die in Klassen deklarierten Variablen kommen ihnen aber nahe: Auch auf sie können Methoden zugreifen, ohne sie in der Parameterliste aufzuführen.
- Da Klassen meist nicht sehr groß sind, ist dies akzeptabel (und normal in der objektorientierten Programmierung).

  Statt Variablen, die global im ganzen Programm bekannt sind, hat man jetzt "globale Variablen" beschränkt auf die Klasse. Wenn man Zugriffe von außerhalb der Klasse zuläßt (weil man die Variable nicht als "`private`" deklariert), hat man aber Probleme, die denen der früheren globalen Variablen ähnlich sind. Außerdem sollte man in der Dokumentation von Methoden natürlich erwähnen, auf welche Variablen (Attribute) zugegriffen wird (insbesondere schreibend), sofern dies nicht offensichtlich ist.

Lokale vs. globale Variablen (3)

- Der Zweck von Prozeduren war es, ein in sich geschlossenes Programmstück mit möglichst kleiner Schnittstelle zum Rest des Programms als Einheit zu verpacken.

  Könnte man beliebig auf Variablen fremder Prozeduren zugreifen, wäre die Interaktion zwischen den Prozeduren völlig undurchschaubar. Lokale Variablen sind in klassischer wie objektorientierter Programmierung wichtig.

- In der klassischen prozeduralen Programmierung läuft idealerweise jeder Datenaustausch einer Prozedur mit dem Rest des Programms nur über Parameter ab.

- In der objektorientierten Programmierung gehört dagegen zum Konzept, dass Objekte einen Zustand haben, auf den Methoden der Klasse lesend und schreibend zugreifen können.

  Siehe Kapitel 11 (Klassen). Die Parameterlisten werden so auch kürzer.

Warnung vor Fehlern

- Während man für Variablen außerhalb von Methoden z.B. `public` und `private` verwenden kann, geht dies für lokale Variablen nicht (Syntaxfehler).

  In der letzten Klausur haben das mehrere Studenten gemacht. Sie können das nie ausprobiert haben. Lokale Variablen sind immer "super-privat", und nur innerhalb der einen Methode (bzw. dem Block) zugreifbar. Dagegen kann man auf `private`-Attribute von allen Methoden der Klasse aus zugreifen.

- Es ist ein schwerer Stilfehler, eine außerhalb von Methoden deklarierte ("globale") Variable zu verwenden, wenn eine lokale Variable gereicht hätte.

- Allgemein reduziert man die Fehlermöglichkeiten, wenn Variablen nur dort zugreifbar sind, wo man sie auch tatsächlich braucht.

Variablen gleichen Namens

- Wenn eine Variable mit Namen `x` schon deklariert ist, kann man nicht noch eine zweite Variable mit dem gleichen Namen `x` deklarieren.

  Der Compiler wüßte dann ja nicht mehr, was man meint, wenn man `x` schreibt.

- Wenn eine Methode einen Parameter `x` hat, kann man entsprechend nicht noch eine lokale Variable `x` deklarieren.

- Da die Deklaration einer lokalen Variable am Ende der Methode wieder vergessen wird, kann man anschließend (in einer anderen Methode) eine weitere Variable mit dem gleichen Namen deklarieren.

  Genauer wird die Deklaration am Ende des Blockes aus der entsprechenden Tabelle des Compilers gelöscht. Wenn es sich um einen geschachtelten Block gehandelt hat (nicht den ganzen Rumpf), kann man auch in der gleichen Prozedur in einem späteren Block eine Variable mit gleichem Namen deklarieren.

Verschattung (1)

- Eine Ausnahme von der Regel "niemals gleichzeitig zwei Variablen mit gleichem Namen" ist:
  - Man kann eine lokale Variable mit dem gleichen Namen wie eine statische Variable (bzw. Attribut, siehe Kap. 11) einführen.
  - Dann steht der Name innerhalb des entsprechenden Blockes für die lokale Variable. Sie ist "näher" deklariert.
  - Die statische Variable mit gleichem Namen ist innerhalb der Methode nicht direkt zugreifbar: Es ist durch die lokale Variable "verschattet".

    Eventuell wußte der Programmierer der Methode nicht, dass es schon eine statische Variable mit diesem Namen gab. Dann schadet es nichts, wenn er darauf nicht zugreifen kann. Ansonsten kann er Klassennamen und einen Punkt voranstellen (z.B. `SVarDemo.anzAufrufe`).

Verschattung (2)

```
(1)  class Verschattung {
(2)      static int n = 5;
(3)
(4)      public static void main(String[] args) {
(5)          System.out.println(n); // 5
(6)          int n = 3;
(7)          System.out.println(n); // 3
(8)          System.out.println(Verschattung.n); // 5
(9)      }
(10) }
```

Dies ist schlechter Stil, zeigt aber die Möglichkeiten: In Zeile (5), bevor die lokale Variable `n` deklariert wurde, bezieht sich `n` noch auf das statische Attribut. Anschliessend, in Zeile (7) auf die lokale Variable. Mit Angabe des Klassennamens kann auf das statische Attribut in Zeile (8) noch zugegriffen werden: Es ist unverändert. Besser: Verschattungen in ganzer Methode einheitlich.

Blockstruktur (1)

- Es gibt nicht nur
  - statische Variablen (außerhalb von Methoden deklariert) und
  - lokale Variablen und Parameter (innerhalb),

  sondern man kann auch in einer Methode Blöcke (Gültigkeitsbereiche) in einander schachteln.
- Die Regel ist immer dieselbe:
  - Weiter außen deklarierte Variablen gelten auch innen

    (sofern sie nicht verschattet d.h. von einer anderen Deklaration mit gleichen Namen verdeckt werden)
  - aber umgekehrt nicht.

    In Java ist es verboten, lokale Variablen durch andere lokale Variablen zu verschatten, in den meisten anderen Sprachen (z.B. C++) nicht.

Blockstruktur (2)

```
(1)  class Blockstruktur {
(2)      static int a = 2;
(3)
(4)      public static void main(String[] args) {
(5)          int b = 3;
(6)          for(int i = 10; i < 30; i += 10) {
(7)              int c = a + b + i;
(8)              System.out.println(f(c));
(9)          } // Ende Gültigkeitsbereich c, i
(10)     } // Ende Gültigkeitsbereich b, args
(11)
(12)     static int f(int n) {
(13)         return a * n;
(14)     } // Ende Gültigkeitsbereich n
(15) } // Ende Gültigkeitsbereich a
```

Hinweis zur Lebensdauer von Variablen

- Lokale Variablen werden am Ende des Blockes, in dem sie deklariert sind, wieder gelöscht.
- Änderungen von lokalen Variablen (und von Parametern, s.u.) wirken sich also nach Ende der Ausführung der jeweiligen Methode nicht mehr aus.
- Ändert man dagegen statische Variablen, so bleiben sie natürlich auch nach Ende der jeweiligen Methode verändert.
- Wenn man einen Wert nur vorübergehend (während der Ausführung einer Methode) benötigt, so sollte man dafür eine lokale Variable verwenden, keine statische Variable.
- Statische Variablen sind für Werte gedacht, die man sich zwischen Methodenaufrufen merken muss.

Inhalt

Rekursive Funktionen (1)

- Eine Funktion kann auch sich selbst aufrufen.
  Die Funktion und der Aufruf heißen dann “rekursiv”.
- Selbstverständlich geht das nicht immer so weiter,
  sonst erhält man das Gegenstück zu einer Endlosschleife,
  die Endlosrekursion.
- Endlosrekursionen führen normalerweise schnell zu einem
  “Stack Overflow”.

  Oft ist nur ein relativ kleiner Speicherbereich für den Stack reserviert,
  z.B. 64 KByte (die Größe kann meist mit einer Option gesteuert werden).
  Im besseren Fall bemerkt die CPU automatisch den Stack Overflow.
  Im schlechteren Fall werden einfach Hauptspeicher-Adressen überschrieben.
  Bei Java kann das aber nicht vorkommen: Die JVM überwacht den Stack
  und erzeugt ggf. eine Exception: `StackOverflowError` oder `OutOfMemoryError`.

Rekursive Funktionen (2)

- Beispiel einer rekursiven Funktion ist die Fibonacci-Funktion: Sie ist für $n \in \mathbb{N}_0$ definiert durch:

$$f(n) := \begin{cases} 1 & \text{für } n = 0,\ n = 1 \\ f(n-1) + f(n-2) & \text{sonst.} \end{cases}$$

Eine mögliche Veranschaulichung ist, dass $f(n)$ die Anzahl von Kaninchenpärchen zum Zeitpunkt n ist: Man nimmt an, dass keine Kaninchen sterben, deswegen hat man zum Zeitpunkt n mindestens $f(n-1)$ Pärchen. Außerdem haben die zum Zeitpunkt $n-2$ existierenden Kaninchenpärchen jeweils ein Pärchen als Nachwuchs bekommen (man nimmt an, dass sie eine Zeiteinheit brauchen, um geschlechtsreif zu werden).

Fibonacci-Zahlen werden in der Informatik aber z.B. auch bei der Analyse von AVL-Bäumen benötigt.

Rekursive Funktionen (3)

```
(1)  class Fib {
(2)      static int fib(int n) {
(3)          if(n < 2)
(4)              return 1;
(5)          else {
(6)              int f1 = fib(n-1);
(7)              int f2 = fib(n-2);
(8)              return f1 + f2;
(9)          }
(10)     }
(11)
(12)     public static void main(String[] args) {
(13)         System.out.println(fib(4));
(14)     }
(15) }
```

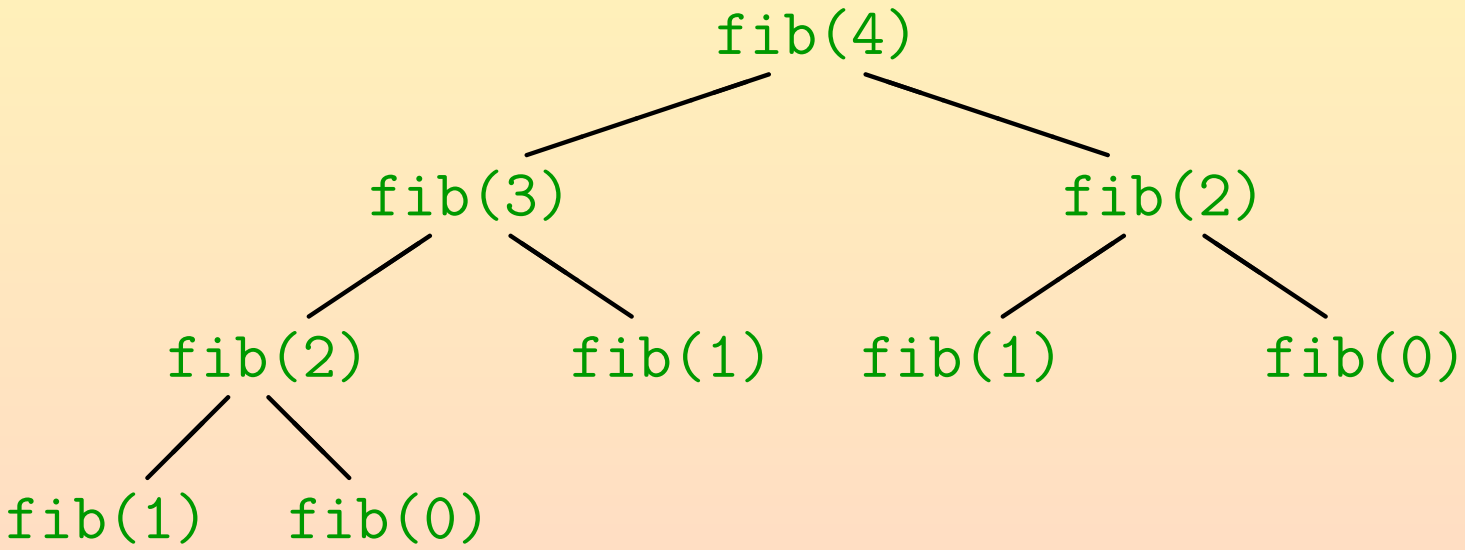
Man könnte unter `else` auch einfach "`return fib(n-1) + fib(n-2)`" schreiben, und braucht `f1` und `f2` dann nicht (normalerweise besser). Hier für Erklärung aber so.

Rekursive Funktionen (4)

- Die Terminierung ist hier garantiert, weil
  - der Parameter `n` bei jedem Aufruf um mindestens 1 kleiner wird, und
  - rekursive Aufrufe nur bei $\texttt{n} \geq 2$ stattfinden.
- Der gleiche Funktionswert wird mehrfach berechnet, man könnte durch eine Umwandlung in eine iterative Berechnung (mit einer Schleife) ohne Doppelung Laufzeit sparen.

  Das obige Programm hat aber den Vorteil, dass es der mathematischen Definition am nächsten kommt. Die iterative Lösung ist eine Übungsaufgabe.
  Richtig nützlich ist Rekursion für rekursive Datenstrukturen wie Bäume (siehe Vorlesung "Datenstrukturen und effiziente Algorithmen I").
  Dort ist eine iterative Lösung recht kompliziert, und eine rekursive sehr einfach.
  Es gibt dann auch keine Dopplung wie hier.

Rekursive Funktionen (5)



- Dieser Graph zeigt die Funktionsaufrufe: Z.B. führt der Aufruf `fib(4)` zu den Aufrufen `fib(3)` und `fib(2)`.

Rekursive Funktionen (6)

- Bei rekursiven Prozeduren braucht man gleichzeitig mehrere Kopien der Parameter und der lokalen Variablen:
  - Sei z.B. der Aufruf für `n` $= 2$ betrachtet.
  - Darin wird die Funktion für `n` $= 1$ aufgerufen.
  - Wenn dieser Aufruf zurückkehrt, hat `n` wieder den Wert 2 (so dass anschließend `fib(n-2)` zum Aufruf `fib(0)` führt).
  - Es gibt hier gleichzeitig zwei verschiedene Variablen `n` (eine mit dem Wert 2, eine mit dem Wert 1).

Rekursive Funktionen (7)

- Das gleiche passiert mit den lokalen Variablen `f1` und `f2`:
  - Sei z.B. der Aufruf `fib(4)` betrachtet.
  - Er weist `f1` den Wert `3` zu (`fib(3)`).
  - Anschließend wird `fib(2)` aufgerufen.
    Dies setzt seine Kopie von `f1` auf `1`.
  - Wenn der rekursive Aufruf zurückkehrt, hat `f1` wieder den Wert `3`: Es ist eine andere Variable mit gleichem Namen.

Rekursive Funktionen (8)

- Verschiedene Variablen mit gleichem Namen gibt es auch als lokale Variablen in unterschiedlichen Methoden (s.o.).
- Die Situation bei der Rekursion ist aber anders (und komplizierter):
  - Im Programm steht syntaktisch nur eine Deklaration, und
  - zur Compile-Zeit ist nicht bekannt, wie viele Kopien der Variablen später zur Laufzeit erforderlich sein werden.
- Für jeden Aufruf einer Methode/Prozedur wird ein neuer Satz lokaler Variablen angelegt.

Ausblick: Anwendungen der Rekursion

- Die Fibonacci-Funktion kann man auch relativ einfach iterativ (mit einer Schleife) berechnen.

  Immerhin nicht ganz so einfach wie die Fakultät (häufiges Beispiel).

- Es gibt aber auch Probleme, die sich viel einfacher rekursiv als iterativ lösen lassen, z.B.
  - Durchsuchen einer Ordner-Hierarchie nach einer Datei.

    Ein Dateiordner kann auch selbst Dateiordner enthalten, hier hat man also schon eine rekursiv definierte Datenstruktur. Allgemeiner Baumstrukturen (z.B. Operatorbäume), Schachtelungen, Hierarchien.
  - Vorausschau in Spielen, wenn man alternative Züge probiert, und dann entsprechend rekursiv für den Gegner (bis zu einer gewissen Tiefe).

    Allgemeiner Durchsuchen von Alternativen mittels Backtracking (z.B. Labyrinth).