Objektorientierte Programmierung

Kapitel 11: Statische Methoden

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2013/14

http://www.informatik.uni-halle.de/~brass/oop13/

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 12 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 (Eingabewert, Argument) 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`.

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. `1`, 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. 1) gebunden.
- Dies wirkt wie eine Zuweisung: `n = 1;`
- Anschließend wird der Rumpf der Methode ausgeführt.
- Die Anweisung

  ```
  return n * n;
  ```

  beendet die Ausführung der Methode und legt den Ergebniswert (Rückgabewert) fest (im Beispiel: 1).

Beispiel, Grundbegriffe (5)

- Der Aufruf einer Methode ist ein Wertausdruck (Expression).
  - Die Eingabewerte (aktuelle Parameter) der Methode können mit beliebig komplexen Wertausdrücken berechnet werden.
  - Der Rückgabewert der Methode kann selbst in einem komplexen Wertausdruck weiter verwendet werden.
  - Im Beispiel ist der Rückgabewert Eingabe des Operators + (Stringkonkatenation bzw. zuerst Umwandlung nach `String`).

Beispiel, Grundbegriffe (6)

- Statt “Parameter” kann man auch “Argument” (der Methode) sagen.

  Je nach Kontext kann “Argument” entweder “aktueller Parameter” oder “formaler Parameter” bedeuten. Man sagt aber nicht “aktuelles/formales Argument”.

  Die Unterscheidung “aktuell” vs. “formal” macht man nur, solange man sich mit der Parameterübergabe beschäftigt. Später ergibt sich das immer aus dem Kontext.

- Statt “aktueller Parameter” sagt man auch “Eingabewert” (der Methode).

  Der mit `return` festgelegte Rückgabewert wäre entsprechend der Ausgabewert der Methode.

- Man darf Eingabewerte von Methoden (Parameterwerte) aber nicht mit Eingaben von der Tastatur verwechseln (s.u.).

Beispiel, Grundbegriffe (7)

- Sie müssen immer die Aufgabenstellung genau einhalten:
  - Es ist ein großer Unterschied, ob die Methode einen Parameterwert bekommt, oder selbst von Tastur 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.

Beispiel, Grundbegriffe (8)

- 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 *= 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 (9)

- 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`.

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 vorzeigt abzubrechen.
- Methoden mit anderen Ergebnistyp müssen immer (bis auf Exceptions) mit einer "`return` ⟨`Wert`⟩`;`" Anweisung enden.

  Diese müssen nicht ganz am Schluss stehen, aber der Compiler muss verstehen können, dass die Ausführung nicht das letzte Statement im Rumpf der Methode normal beenden kann. Jeder Ausführungspfad muss also mit einem "`return` ⟨`Wert`⟩`;`" oder "`throw` ⟨`Exception`⟩`;`" enden.

Aufruf von Methoden

- Statische Methoden werden normalerweise in der Form

  ⟨Klassenname⟩ . ⟨Methodenname⟩ (⟨Parameter⟩)

  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⟩ (⟨Parameter⟩)

- 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) }
```

Inhalt

Lokale Variablen (1)

- In Kapitel 8 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 12 (Klassen). Die Parameterlisten werden so auch kürzer.

Variablen gleichen Namens (1)

- 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.

- 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 gelöscht. Wenn es sich um einen geschachtelten Block gehandelt hat, kann man anschließend auch in der gleichen Prozedur eine neue Variable mit gleichem Namen deklarieren.

- Dies sind zwei unterschiedliche Variablen, die nichts miteinander zu tun haben.

Variablen gleichen Namens (2)

- Verschiedene Prozeduren/Methoden sollten sich möglichst wenig gegenseitig beeinflussen.

  Wenn man eine Methode anwendet, soll man nicht gezwungen sein, in den Methodenrumpf hineinzuschauen. Methoden sollten unabhängig von einander entwickelt werden können.

- Deswegen ist es ganz natürlich, dass es keine Namenskonflikte zwischen den innerhalb der Methoden deklarierten Variablen gibt.

- Die Parameter werden wie lokale Variablen behandelt: Sie sind auch nur innerhalb der jeweiligen Methode bekannt.

  Man kann dann nicht eine lokale 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. 12) 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 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(n(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) }
```

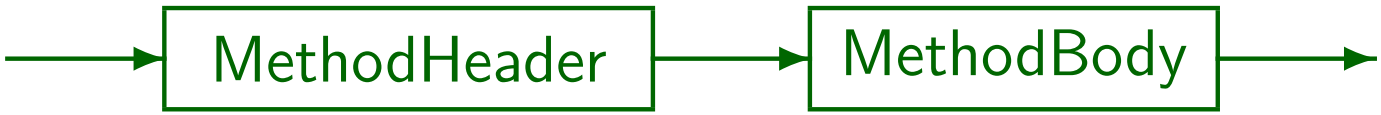
Hinweis zur Lebensdauer von Variablen

- 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.

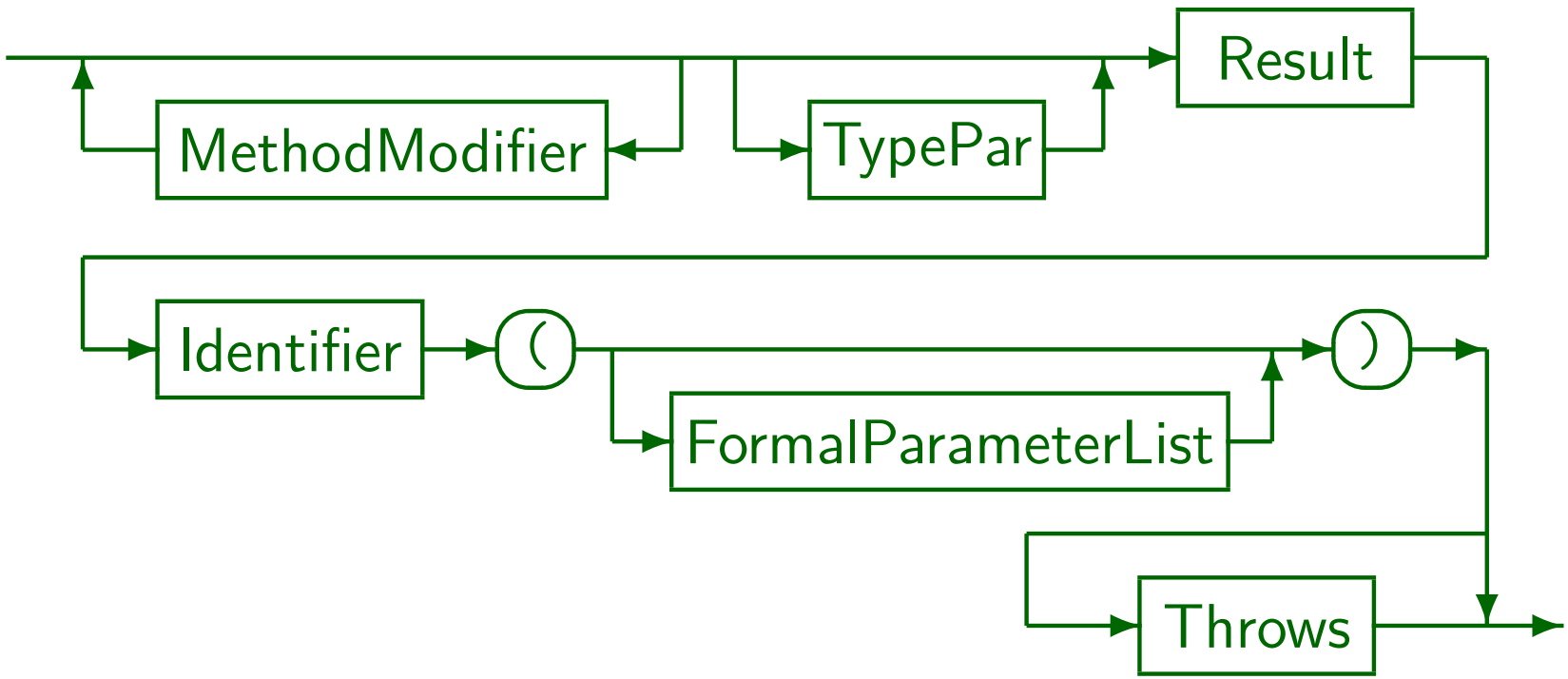
Inhalt

Methoden-Deklarationen: Syntax (1)

- MethodDeclaration:



- MethodHeader:



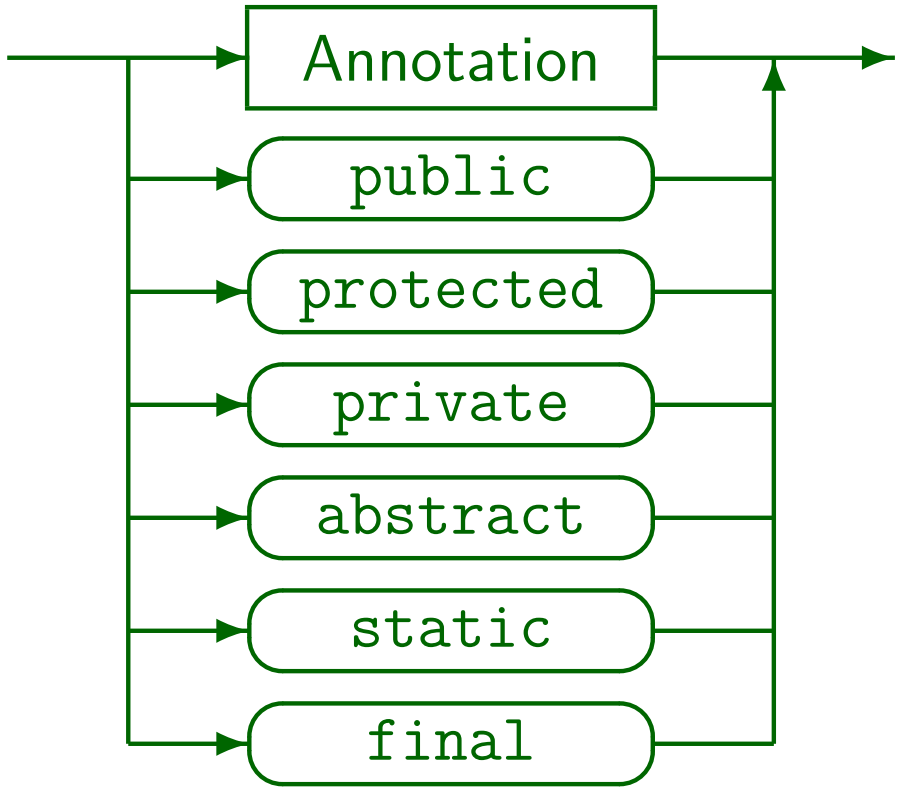
Methoden-Deklarationen: Syntax (2)

- Eine Methodendeklaration besteht also aus:
  - Modifizierern wie `public` und `static`,
  - optionalen Typ-Parametern (siehe Kapitel 21),
  - 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 13).

Methoden-Deklarationen: Syntax (3)

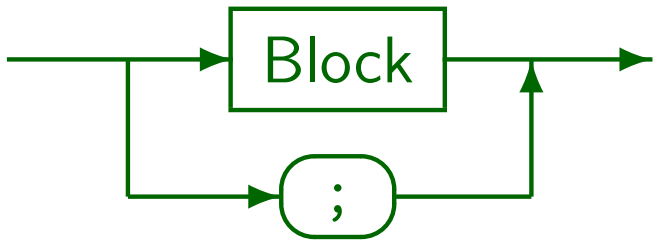
- MethodModifier:



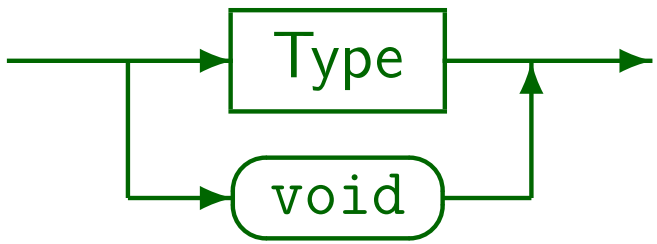
Bisher wurden nur die Modifier `public` und `static` verwendet. Die übrigen hier gezeigten Modifier werden in späteren Kapiteln besprochen. Es gibt noch weitere, die in der Vorlesung nicht behandelt werden: `synchronized`, `native`, `strictfp`.

Methoden-Deklarationen: Syntax (4)

- MethodBody:



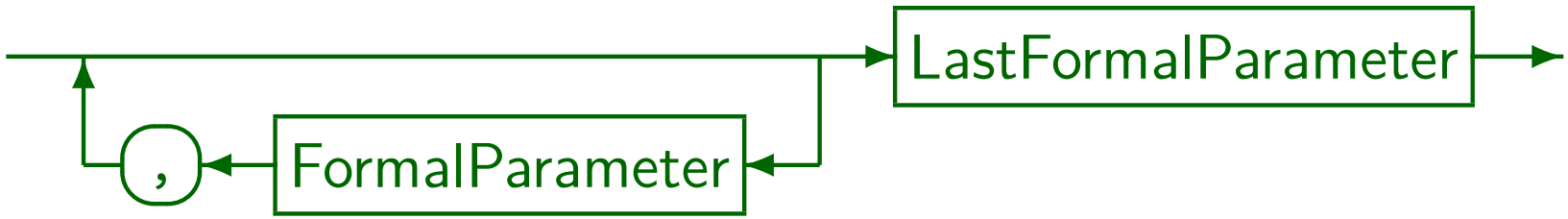
- Result:



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

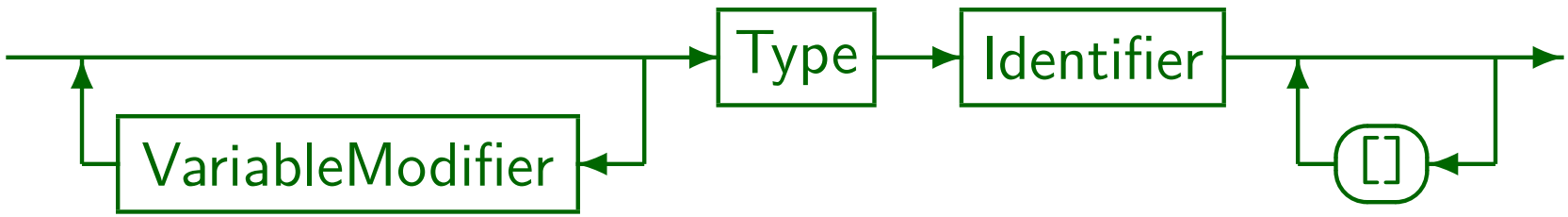
Methoden-Deklarationen: Syntax (5)

- FormalParameterList:



Der letzte formale Parameter wird getrennt behandelt, weil man Methoden mit variabler Argument-Anzahl deklarieren kann (siehe Kapitel 17), dann wird ihm ein Array mit allen übrigen Argumenten zugewiesen.

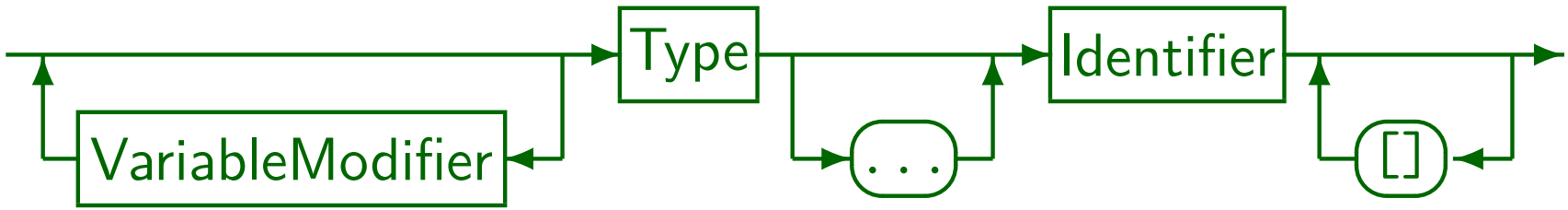
- FormalParameter:



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.

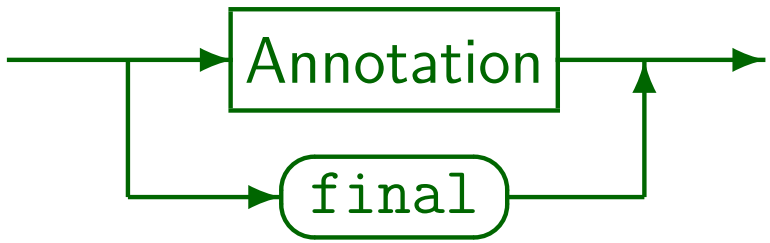
Methoden-Deklarationen: Syntax (6)

- LastFormalParameter:



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

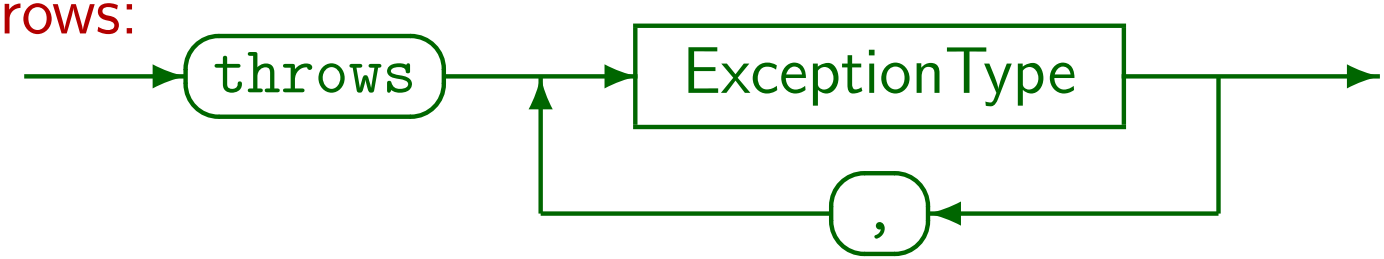
- VariableModifier:



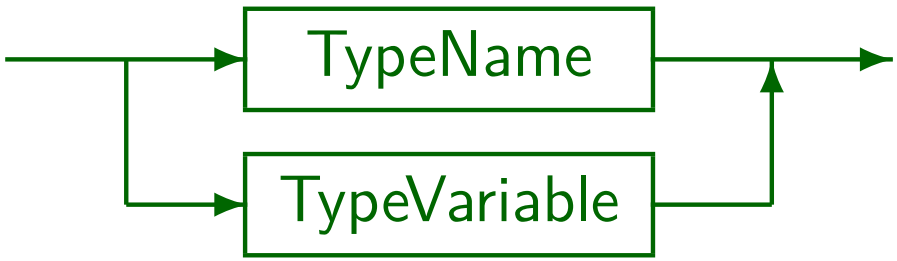
Die Angabe von `final` bewirkt, dass Zuweisungen an den Parameter im Methodenrumpf verboten sind. Annotationen werden in dieser Vorlesung nicht behandelt (außer "`@Override`" für Methoden in Kapitel 13).

Methoden-Deklarationen: Syntax (7)

- Throws:

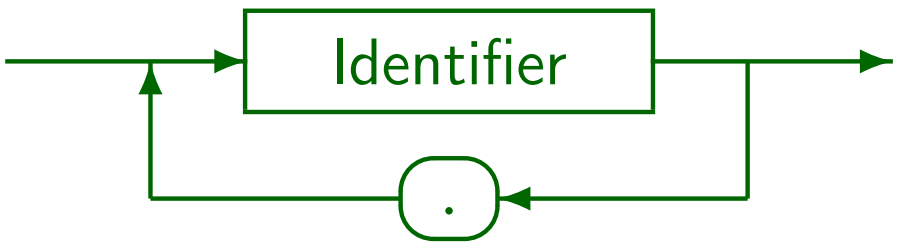


- ExceptionType:



Typ-Variablen werden erst in Kapitel 21 behandelt.

- TypeName:



Der Typ muss ein Untertyp von `Throwable` sein (siehe Kap. 9, 13, 19).

Inhalt

Parameterübergabe (1)

- Java benutzt "call by value" zur Parameterübergabe:
  - Selbst wenn der aktuelle Parameter eine Variable ist, wird der Wert dieser Variable an den formalen Parameter zugewiesen (d.h. kopiert).
  - Auf die Variable, die als aktueller Parameter angegeben ist, hat man in der Methode/Prozedur keinen Zugriff.
  - Auch eine Zuweisung an einen 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)
(3)      static void test(int[] a) {
(4)          a[0] = 2;
(5)      }
(6)
(7)      public static void main(String[] args) {
(8)          int[] a = new int[1];
(9)          a[0] = 1;
(10)         System.out.println(a[0]); // 1
(11)         test(a); // a[0] wird geaendert!
(12)         System.out.println(a[0]); // 2
(13)     }
(14) }
```

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 12).
  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.

Parameterübergabe (7)

- 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.

Parameter-Übergabemechanismen (1)

- In der Geschichte der Programmiersprachen wurden eine ganze Reihe verschiedener Parameter-Übergabemechanismen entwickelt.
- Die häufigsten sind (z.B. in Pascal vorhanden):
  - "Call by Value": Selbst wenn der aktuelle Parameter eine Variable ist, wird ihr Wert übergeben, d.h. in den formalen Parameter kopiert.
  - "Call by Reference" (`var`-Parameter in Pascal):
    Der aktuelle Parameter muss eine Variable sein, ihre Adresse wird übergeben.

    Da die Variable selbst übergeben wird, wirken sich Zuweisungen in der Prozedur auch auf den Variablenwert hinterher beim Aufrufer aus. Dies ist der klassische Ausgabe- oder Ein-/Ausgabe-Parameter.
    Es gibt aber noch mehr, z.B. "Call by Name", "Copy-in, Copy-out".

* Parameter-Übergabemechanismen (2) *

- Java hat nur "Call by Value". Weil man mit Objekten aber über Referenzen arbeitet, kann man "Call by Reference" simulieren.
- C hatte auch nur "Call by Value". Man konnte aber mit dem Operator "&" aber explizit die Adresse einer Variablen bestimmen und diese übergeben.

  Wenn in einem Prozeduraufruf "`&x`" stand, war klar, dass die aufgerufene Prozedur die Möglichkeit hatte, `x` zu ändern, und dies höchstwahrscheinlich auch tun würde.
- Bei C++ wurden dann Referenzen eingeführt, insbesondere auch für Parameter. Das war dann "Call by Reference".

  In C++ kann man Objekte aber auch "Call by Value" übergeben, d.h. das ganze Objekt kopieren. Das ist aber meistens ineffizient.

Inhalt

* Details: Rückkehr-Adressen (1) *

```
(1)  class ReturnTest {
(2)
(3)      static void printSquare(int n) {
(4)          System.out.print(n);
(5)          System.out.print(" zum Quadrat ist: ");
(6)          System.out.println(n * n);
(7)      }
(8)
(9)      public static void main(String[] args) {
(10)         System.out.println("Erster Aufruf:");
(11)         printSquare(15);
(12)         System.out.println("Zweiter Aufruf:");
(13)         printSquare(25);
(14)         System.out.println("Fertig!");
(15)     }
(16) }
```

* Details: Rückkehr-Adressen (2) *

- Eine Prozedur kann an verschiedenen Stellen in einem Programm aufgerufen werden.
- Sie kehrt jeweils zu der Stelle zurück, von der aus sie aufgerufen wurde (direkt hinter den Aufruf).
- Die CPU hat nur einen "Instruction Pointer" (IP) für die Adresse des aktuell abzuarbeitenden Befehls.

  Die Erklärungen zur internen Implementierung von Sprachkonstrukten auf Maschinenebene sind für manche Studierende hilfreich (die die Funktionsweise einer CPU schon gut kennen). Sonst können Sie diesen Teil ignorieren.

- Damit sie weiß, zu welcher Adresse sie am Ende der Prozedur zurückkehren soll, wird beim Prozeduraufruf die Adresse des nächsten Befehls in einen speziellen Speicherbereich, den "Stack" gelegt.

* Details: Rückkehr-Adressen (3) *

- Im Beispiel wird beim ersten Aufruf der Methode `printSquare` in Zeile 11 die Adresse des nächsten Maschinenbefehls nach dem Aufruf (also entsprechend Zeile 12) auf den Stack gelegt.

  Beim zweiten Aufruf (in Zeile 13) wird entsprechend die Adresse des Befehls von Zeile 14 auf den Stack gelegt.

- Am Ende des für die Methode erzeugten Maschinencodes steht ein Befehl, der den Instruktion Pointer in der CPU auf den (obersten) Wert im Stack setzt und diesen Wert vom Stack herunter nimmt ("Rücksprung").

* Details: Rückkehr-Adressen (4) *

- Nun kann aber auch innerhalb einer Methode eine weitere Methode aufgerufen werden.
- Deswegen reicht nicht eine einzelne Speicherstelle für die Rücksprungadresse.
- Es entsteht so ein ganzer Stapel (engl. “Stack”) von Rücksprungadressen:
  - Beim Methodenaufruf wird die Rücksprungadresse auf den Stack gelegt.

    Dies ist der aktuelle Inhalt des Instruction Pointers plus n, wobei n die Länge des Maschinenbefehls für den Unterprogramm-Aufruf ist.
  - Beim Rücksprung wird sie herunter genommen.

* Details: Rückkehr-Adressen (5) *

```
 (1)  class StackDemo {
 (2)
 (3)      static void f(int m) {
 (4)          System.out.println("f(" + m + ")");
 (5)      }
 (6)
 (7)      static void g(int n) {
 (8)          f(n+1);
 (9)          f(n+2);
(10)      }
(11)
(12)      public static void main(String[] args) {
(13)          g(10);
(14)          g(20);
(15)          System.out.println("Fertig!");
(16)      }
(17)  }
```

* Details: Rückkehr-Adressen (6) *

- Beim ersten Aufruf von g in Zeile 13 wird 14 (bzw. die entsprechende Adresse im Maschinenprogramm) auf den Stack gelegt.
  - Vor Aufruf von g:

  - Nach Aufruf von g (beginnt effektiv in Zeile 8):

* Details: Rückkehr-Adressen (7) *

- Innerhalb von `g` wird nun in Zeile 8 die Methode `f` aufgerufen.
- Also kommt die Adresse von Zeile 9 (nachfolgender Befehl) auf den Stack, und der Instruction Pointer wird auf die Startadresse von `f` gesetzt (Zeile 4):

- Nun druckt `f` den Text "`f(11)`".

* Details: Rückkehr-Adressen (8) *

- Anschließend (Ende von `f`) kehrt die CPU zu der obersten Adresse auf dem Stack zurück (Zeile 9) und nimmt diese Adresse vom Stack:

- Dann ruft g das zweite Mal `f` auf:

In Zeile 10 steht sozusagen noch implizit der Befehl "`return;`". Deswegen ist in der Prozedur g nach Rückkehr des zweiten Aufrufs von `f` noch etwas zu tun, wenn es auch nur die eigene Rückkehr ist.

* Details: Rückkehr-Adressen (9) *

- Bei diesem Aufruf druckt `f` den Text "`f(12)`".
- Anschließend kehrt die CPU zu Zeile 10 zurück (oberste Adresse auf dem Stack):

- Dann kehrt auch `g` zu Zeile 14 (in `main`) zurück, auch diese Adresse wird vom Stack genommen:

* Details: Rückkehr-Adressen (10) *

- Anschließend ruft `main` die Methode `g` zum zweiten Mal auf:

- Dies ruft wieder `f` auf:

- Jetzt wird "`f(21)`" gedruckt. u.s.w.

* Aufruf-Schachtelung *

- Druckt man am Anfang und am Ende jeder Methode einen kurzen Text mit einer öffnenden bzw. schließenden Klammer aus, so erhält man immer eine korrekte Klammerschachtelung:

```
(Anfang von main
    (Anfang von g: n=10
        (Anfang von f: m=11
         Ende von f)
        (Anfang von f: m=12
         Ende von f)
     Ende von g)
    ...
 Ende von main)
```

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) }
```

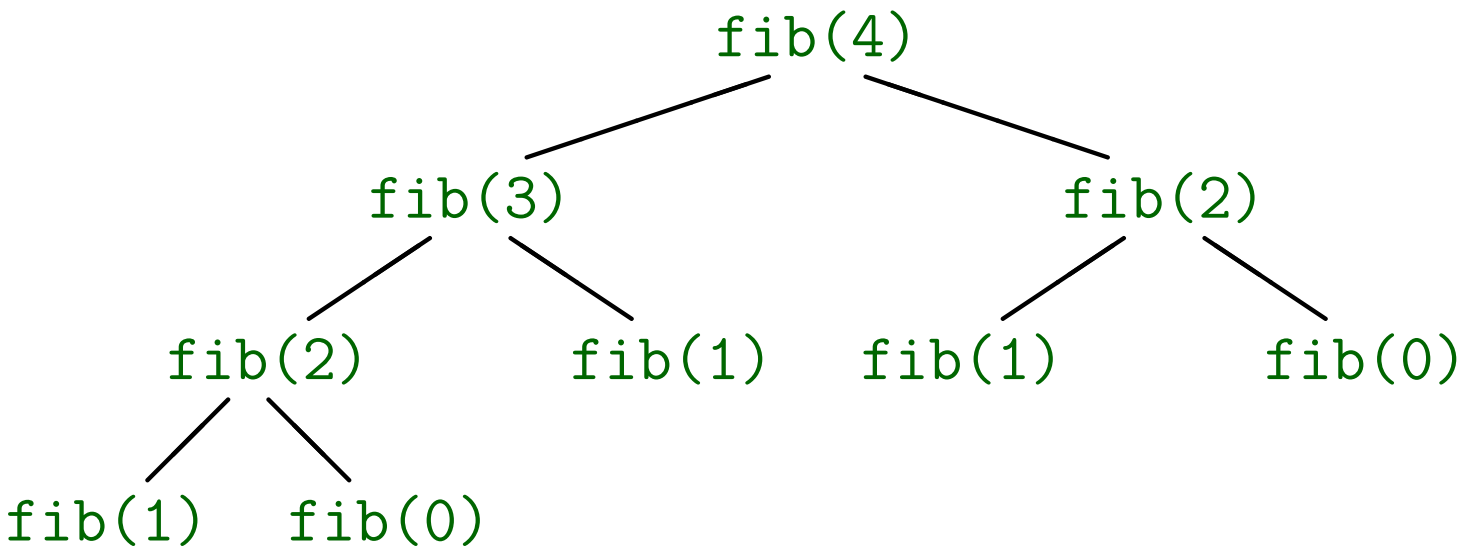
Man könnte unter `else` auch einfach "`return fib(n-1) + fib(n-2)`" schreiben, und braucht `f1` und `f2` dann nicht (normalerweise besser). Hier für Erklärung aber so.

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.

* Details: Stackframes (1) *

- Der Compiler kann bei rekursiven Methoden den Variablen also keine festen Adressen zuordnen.

  Bei den meisten Sprachen/Compilern werden rekursive und nicht-rekursive Prozeduren nicht unterschieden. Alle Prozeduren werden als potentiell rekursiv behandelt. In der Sprache Fortran war Rekursion ausgeschlossen, dort können Variablen feste Adressen haben. Bei vielen Prozeduren, die nicht gleichzeitig aktiv sind, wäre das aber eine Verschwendung von Speicherplatz.

- Jeder Methodenaufruf (Methoden-Aktivierung, "method invocation") benötigt seinen eigenen, frischen Satz von lokalen Variablen.

- Daher bietet es sich an, die lokalen Variablen ebenfalls auf dem Stack abzulegen.

* Details: Stackframes (2) *

- Jeder Methodenaufruf hat auf dem Stack einen "Stack Frame" oder "Activation Record".

  Aktivierungs-Records auf dem Stack existieren für jeden Methoden-Aufruf, der noch nicht beendet wurde.

- Dieser enthält insbesondere:
  - Speicherplatz für die Parameter
  - Speicherplatz für die lokalen Variablen

    In der JVM gibt es für Parameter und lokale Variablen ein gemeinsames Array, in dem zuerst die Parameter gespeichert sind und dann die lokalen Variablen. Bei nicht-statischen Methoden ist der erste Parameter (an Position 0) eine Referenz auf das Objekt, für das die Methode aufgerufen wurde.

  - Die Rücksprung-Adresse

* Details: Stackframes (3) *

- Wenn `fib` für `n=4` aufgerufen wurde, und seinen zweiten rekursiven Aufruf macht, sieht der oberste Teil des Stack ungefähr so aus: