Objektorientierte Programmierung

Kapitel 16: Arrays für Fortgeschrittene

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2018/19

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

Inhalt

Wiederholung: Grundlagen (1)

- Arrays geben die Möglichkeit, sich eine große (und ggf. vom Programm berechnete) Anzahl von Variablen zu beschaffen.

  Weil die genaue Anzahl erst zur Laufzeit festgelegt werden muss, kann sie von den Eingabedaten oder Optionen in einer Konfigurationsdatei abhängig sein.

- Die Variablen in einem Array müssen alle den gleichen Typ haben, den Elementtyp (oder Basistyp) des Arrays.
- In Java enthält der Array-Datentyp selbst nicht die Größe des Arrays. Man deklariert eine Variable `a` für ein Array mit Elementtyp `T` folgendermaßen: `T[] a;`
- Die Variable `a` enthält nur eine Referenz auf das Array, das Array selbst muss man mit `new` erzeugen, und dabei die Größe `n` angeben: `a = new T[n];`
- Oder Deklaration+Initialisierung: `T[] a = new T[n];`

Wiederholung: Grundlagen (2)

- Die einzelne Variable in einem Array `a` der Größe `n` wird durch einen Index ausgewählt, der ein `int`-Wert von `0` bis `n - 1` ist.
- Syntaktisch schreibt man den Array-Zugriff `a[i]`, wobei statt `i` ein beliebiger Wertausdruck mit Ergebnistyp `int` stehen kann, und statt `a` ein Ausdruck, der ein Array liefert.

  `a[i]` liefert eine Variable, kann also auch links von einer Zuweisung stehen (Schreibzugriff). Wie üblich wird bei Bedarf der in der Variablen gespeicherte Wert genommen (Lesezugriff).
- Wenn der berechnete Index nicht im Bereich `0` bis `n - 1` liegt, erhält man eine `ArrayIndexOutOfBoundsException`.

  Da das Auftreten von Exceptions von den Eingabedaten abhängt, wird es beim Testen nicht sicher gefunden. Man sollte also besonders sorgfältig überlegen, und beweisen, dass dieser Fall nicht auftreten kann.

Wiederholung: Grundlagen (3)

- Die Größe eines Arrays kann nicht nachträglich verändert werden.
- Die Variable a ist aber nicht ein eine bestimmte Größe gebunden, sie kann später auch auf andere Array-Objekte mit anderer Größe zeigen (aber dem gleichen Element-Typ).

  Wenn das ursprünglich angelegte Array später zu klein sein sollte, kann man ein größeres anlegen, den Inhalt des alten Arrays in das neue kopieren, und dann das neue Array der Variablen a zuweisen. Das alte Array wird dann vom Garbage Collector eingesammelt (sofern es keine anderen Verweise darauf gibt).
- Man kann die Größe abfragen mit dem Attribut `length`:

  ```
  System.out.println(a.length);
  ```

  Dieses Attribut ist `final`, d.h. es sind keine Zuweisungen daran möglich.

Wiederholung: Grundlagen (4)

- Wie bei Objekten werden auch die Komponenten eines Arrays automatisch initialisiert (auf 0, `null`, `false`, je nach Typ).

  Java stellt sicher, dass man nicht auf uninitialisierte Variablen zugreifen kann. Das kostet etwas Laufzeit, bringt aber mehr Sicherheit. Bei C++ werden Arrays nicht initialisiert (außer Arrays von Objekten).

- Wie bei Objekten vergleicht `==` nur die Referenz.

  Und nicht den Inhalt, d.h. die einzelnen Array-Elemente.

- Die Array-Größe `0` ist möglich, d.h. folgendes Statement zur Erzeugung eines leeren Arrays gibt keinen Fehler:

```
int[] a = new int[0];
```

  Manchmal muss man einer Methode ein Array übergeben, braucht bei einem speziellen Aufruf aber vielleicht keine Werte. Negative Größen geben einen Laufzeit-Fehler (`NegativeArraySizeException`).

Wiederholung: Grundlagen (5)

- Beispiel:

```
int[] a = new int[5];
a[0] = 27; a[1] = 42; ...; a[4] = 56;
```

- Mathematisch gesehen ist der Wert eines Arrays eine Abbildung vom Indexbereich auf den Wertebereich des Element-Datentyps, im Beispiel:

$$\{0,1,2,3,4\} \rightarrow \texttt{int}: \{-2^{31}, \ldots, 2^{31}-1\}$$

- Man kann sie als Tabelle darstellen:

Index	Inhalt
0	27
1	42
2	18
3	73
4	56

Arrays als Objekte

Ausblick für Experten:

- Arrays sind Objekte.

  Man kann Arrays in Variablen vom Typ `Object` speichern. `Object` (s. Kap. 12) ist die gemeinsame Oberklasse für alle Klassen und Array-Typen.

- Array-Typen sind Referenztypen, aber keine Klassen.

  Referenztypen sind Klassen, Interfaces (Kap. 13) und Array-Typen.

- Array-Typen erben einige Methoden von `Object`, und haben sonst keine eigenen Methoden.

  Z.B. erhält man mit `a.clone()` eine Kopie des Arrays `a` (also einen neuen Satz von Variablen, der mit den Werten aus `a` initialisiert ist).
  Die Methode `clone` wird für Arrays überschrieben, alle anderen Methoden haben die Standard-Implementierung aus `Object`. Arrays implementieren die Interfaces `Cloneable` und `Serializable` (s. Kap. 13).

Arrays: Bibliotheksfunktionen

- Es gibt eine Klasse `java.util.Arrays` [http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html] mit nützlichen Hilfsfunktionen (als statische Methoden), z.B.
  - `java.util.Arrays.equals(a, b)` vergleicht zwei Arrays.

    D.h. die Elemente im Array, nicht nur die Referenz auf das Array wie ==.
  - `java.util.Arrays.sort(a)` sortiert das Array.

    In numerischer Ordnung für Arrays von numerischem Typ.

    Man kann auch einen Comparator angeben.
  - `java.util.Arrays.toString(a)` erzeugt eine druckbare Repräsentation.

    Mit `java.util.Arrays.deepToString(a)` auch für geschachtelte Arrays.
  - Es gibt z.B. auch binäre Suche, Kopier- und Füllfunktionen, meist auch eine Version für Teilbereiche eines Arrays.

Inhalt

Implementierung (1)

- Für ein Array der Größe n reserviert der Compiler den n-fachen Speicherplatz wie für eine einzelne Variable des entsprechenden Typs.

  Das wäre so richtig in C und C++. Für Java ist es etwas vereinfacht, da noch die Längeninformation und eventuell Typ-Information hinzukommt. Der Anteil für die Elemente ist aber der Hauptteil, der das Wesen des Arrays ausmacht.

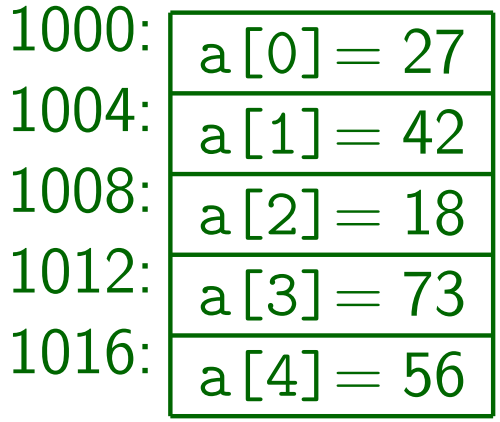
- Wenn z.B. ein `int` 4 Byte belegt, reserviert der Compiler für das Array `a`: $5 * 4 = 20$ Byte.

- Wenn das Array z.B. ab Adresse 1000 beginnt, steht an dieser Stelle der Wert von `a[0]`.

  Er belegt die vier Bytes mit den Adressen 1000 bis 1003.

Implementierung (2)

- Ab Adresse 1004 steht dann `a[1]`.
- Die fünf `int`-Werte stehen also direkt hintereinander im Speicher:



27, 42, 18, 73, 56 sind hier irgendwelche (sinnlosen) Beispiel-Inhalte des Arrays (Variablenwerte).

Implementierung (3)

- Sei allgemein g die Größe des Element-Datentyps.

  Bei `int` also $g = 4$.

- Hat der Compiler für a einen entsprechend großen Speicherbereich ab Adresse s belegt, so steht das Array-Element $a[i]$ an Adresse $s + i * g$.

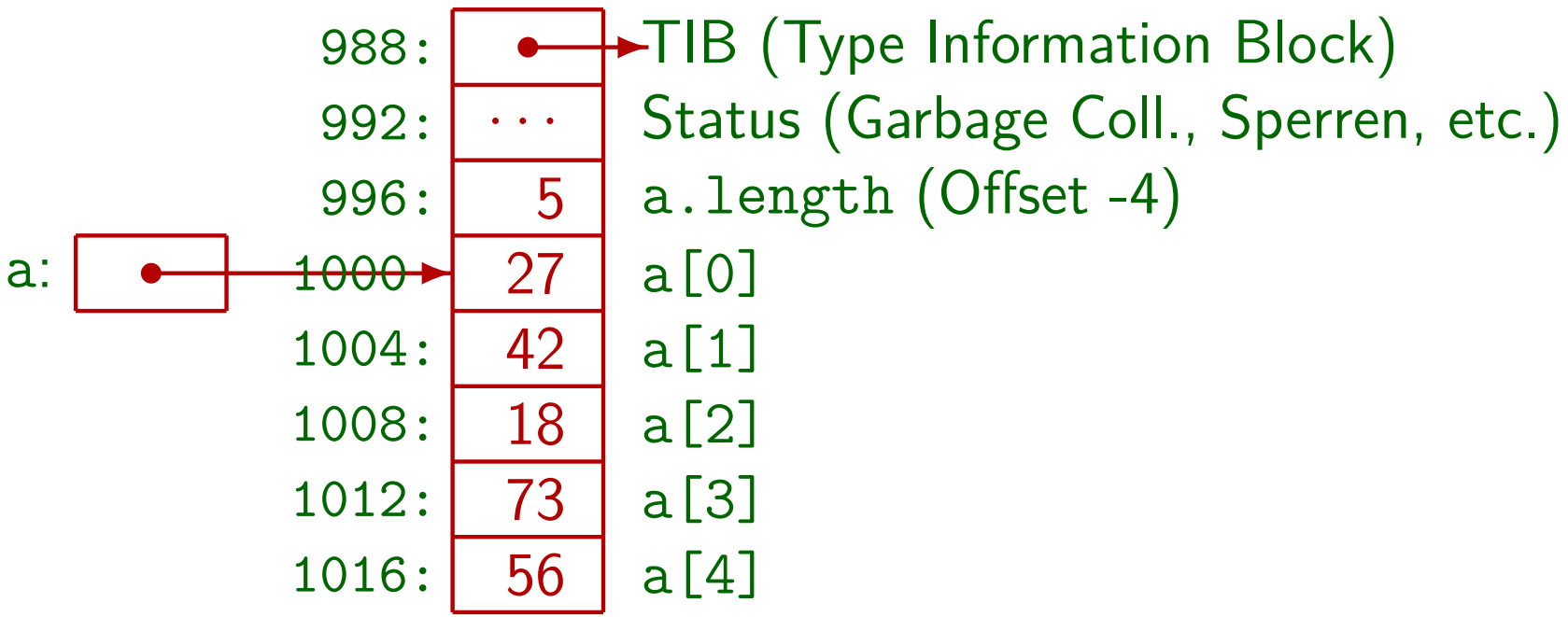
  Deswegen ist es einfacher, wenn der Indexbereich mit 0 beginnt.

- Der Compiler erzeugt zum Zugriff auf Arrayelemente Maschinenbefehle, die diese Adressberechnung zur Laufzeit vornehmen.

  Viele CPUs haben Adressierungsmodi, die die diese Berechnung etwas vereinfachen/beschleunigen. Die Multiplikation mit der Arraygröße muß aber wohl explizit durchgeführt werden. Bei Zweierpotenzen kann der Compiler natürlich einen Shift-Befehl verwenden.

Implementierung (4)

- Wie oben schon angedeutet, ist die obige Erklärung etwas vereinfacht und berücksichtigt nicht die Besonderheiten der Sprache Java. Eine reale Implementierung ist:



Siehe:
[http://pp.ipd.kit.edu/lehre/WS200910/compiler/05-jvm_bytecode.pdf]

Inhalt

Programm-Beispiele (1)

Ganzes Array ausgeben:

- Schleife über Array:

```
for(int i = 0; i < a.length; i++)
    System.out.println(a[i]);
```

- Spezielle `for`-Schleife für Collection-Typen:

```
for(int e : a)
    System.out.println(e);
```

  `e` ist hier eine Element des Arrays. Hätte das Array den Typ `float[]`, so würde man `e` als `float` deklarieren.

- Wenn man unbedingt eine `while`-Schleife will:

```
int i = 0;
while(i < a.length)
    System.out.println(a[i++]);
```

Programm-Beispiele (2)

Prüfen, ob Element in Array:

- Boolesche Variable vor der Schleife auf `false` setzen, falls gefunden, in der Schleife auf `true`:

```
boolean gefunden = false;
for(int i = 0; i < a.length; i++)
    if(a[i] == gesuchterWert)
        gefunden = true;
```

- Schleife abbrechen, falls gefunden:

```
int i;
for(i = 0; i < a.length; i++)
    if(a[i] == gesuchterWert)
        break;
if(i < a.length)
    System.out.println("Gefunden, Position " + i);
```

Programm-Beispiele (3)

Prüfen, ob Element in Array (Forts.):

- Natürlich kann man auch im ersten Beispiel-Programmstück die Schleife abbrechen, sobald es gefunden ist.
- In einer Methode kann man mit `return` die Ausführung beenden und ein Ergebnis definieren:

```
static boolean gefunden(int[] a,
                        int gesuchterWert) {
    for(int i = 0; i < a.length; i++)
        if(a[i] == gesuchterWert)
            return true;
    // Nach der Schleife:
    return false;
}
```

Programm-Beispiele (4)

Klasse zum Speichern von Werten in Array:

- Man braucht hier neben dem Array selbst eine Variable für den aktuellen Füllungsgrad des Arrays, also die nächste freie Index-Position.

  Diese Konstruktion ist ganz typisch: Arrays müssen nicht immer vollständig gefüllt sein.

- Die Klasse ist eine einfache Implementierung von Mengen von `int`-Werten:
  - Der Konstruktor initialisiert sie auf die leere Menge.
  - Die Methode `einfuegen` fügt eine Zahl hinzu.
  - Die Methode `element` prüft, ob eine Zahl enthalten ist.

Programm-Beispiele (5)

```
 (1)  public class intMenge {
 (2)
 (3)      // Begrenzung fuer Implementierung:
 (4)      private static final int MAX_ELEMENTE
 (5)                                   = 100;
 (6)
 (7)      // Attribute:
 (8)      private int anzElemente;
 (9)      private int elemente[];
(10)
(11)      // Konstruktor:
(12)      public intMenge() {
(13)          anzElemente = 0;
(14)          elemente = new int[MAX_ELEMENTE];
(15)      }
(16)
```

Programm-Beispiele (6)

```
(16)      // Methode zum Einfuegen eines Elementes:
(17)      public void einfuegen(int zahl) {
(18)         if(anzElemente == MAX_ELEMENTE) {
(19)            System.out.println("Menge voll");
(20)            System.exit(1); // -> naechste Folie
(21)         }
(22)         elemente[anzElemente++] = zahl;
(23)      }
(24)
(25)      // Methode zum Suchen eines Elementes:
(26)      public boolean element(int zahl) {
(27)         for(int i = 0; i < anzElemente; i++)
(28)            if(elemente[i] == zahl)
(29)               return true;
(30)         return false;
(31)      }
(32)   }
```

Programm-Beispiele (7)

Hinweis (Einhalten von Spezifikationen):

- Wenn man im Team zusammenarbeitet, oder auch nur eine Klasse schreiben will, die man später in anderen Projekten wiederverwenden kann, muss (gemeinsam) entschieden und dokumentiert werden, wie mit Fehlern umzugehen ist.
- Das obige Programm erzeugt eine Ausgabe und bricht das Programm ab. Das geht so nur für kleine Programme.
- Mit einer Exception (oder dem Aufruf einer Fehlermethode) würde man dem Benutzer der Klasse mehr Flexibilität geben.
- Spezifikationen/Absprachen müssen genau eingehalten werden.

  Auch bei der Klausur! Wenn dort steht, dass man die Methode abbrechen soll, ist `return` gemeint, nicht `exit` (beendet ganzes Programm). Notfalls fragen.

Programm-Beispiele (8)

Aufgaben:

- Ändern Sie die Methode `einfuegen` so, dass Zahlen, die bereits in der Menge enthalten sind, nicht nochmals eingefügt werden. Welche Vor- und Nachteile hat das?
- Heben Sie die Beschränkung der maximalen Element-Anzahl auf, indem Sie ggf. ein doppelt so großes Array anfordern, und die bisherigen Elemente umkopieren.
- Implementieren Sie die Klasse statt mit einem Array auch mit einer verketteten Liste.

  Sie müssen eine Hilfsklasse für die einzelnen Elemente in der verketteten Liste einführen. Beachten Sie, dass die Schnittstelle stabil bleibt — von außen kann die Änderung nicht bemerkt werden (außer über die Laufzeit). Was sind die Vor- und Nachteile?

Inhalt

Initialisierung von Arrays (1)

- Man kann ein Array nicht nur mit `new` erzeugen, und dann die Element-Werte setzen, sondern auch direkt alle Einträge auflisten:

```
int[] a = { 2, 3, 5, 7, 11 };
```

- Dies ist äquivalent zu:

```
int[] a = new int[5];
a[0] = 2;
a[1] = 3;
a[2] = 5;
a[3] = 7;
a[4] = 11;
```

Die Array-Größe wird dabei aus der Anzahl der angegebenen Wert bestimmt.

Es finden auch die üblichen Typ-Umwandlungen wie bei einer Zuweisung statt, z.B. kann man einen `int`-Wert in ein `double[]`-Array speichern.

Initialisierung von Arrays (2)

- Es geht selbstverständlich auch mit Objekten:

```
Datum[] termine = {
        new Datum(22, 1, 2013),
        new Datum(29, 1, 2013)
};
```

- Syntaktische Feinheit: Man darf ein Komma nach dem letzten Element schreiben.

  Das vereinfacht z.B. Programme, die die Initialisierungsdaten für ein Array erzeugen: Der letzte Eintrag muss nicht anders behandelt werden.

  Auch eine Umsortierung der Einträge ist einfach möglich.

- Natürlich kann man Elemente eines Arrays von einem Referenztyp auch mit `null` initialisieren.

  In C/C++ verwendet man öfters so eine Markierung am Schluss der Liste.

  In Java ist das nicht nötig, weil man da die Länge des Arrays abfragen kann.

Initialisierung von Arrays (3)

- Initialisierte Arrays sind besonders nützlich, wenn man eine Tabelle mit Daten im Programm angeben muss.

  Beispiele: Daten von Monstern in einem Rollenspiel-Programm, aus einer kontextfreien Grammatik erstellte Parser-Tabelle.
  Alternativ kann man die Daten auch zur Laufzeit von einer Datei lesen.
  Die Lösung mit einem initialisierten Array ist programmiertechnisch einfacher.
  Die Lösung mit der Datei bietet syntaktisch mehr Freiheiten und würde eine Änderung der Daten auch erlauben, wenn man den Quellcode nicht hat.

- Wenn das Array als `final` deklariert wird, heißt das nur, dass die Referenz auf das Array nicht geändert werden kann. Das Array selbst kann schon geändert werden.

  So ist es auch mit Objekten. Dort kann man aber in der Klasse einfach keine Änderungsmethoden vorsehen. Bei Arrays ist dagegen immer die Zuweisung an Array-Elemente möglich, das kann man nicht verhindern.

Initialisierung von Arrays (4)

- Man kann bei der Initialisierung auch den Typ mit angeben:

```
Datum[] termine = new Datum[] { d1, d2, d3 };
```

  Dies setzt voraus, dass `d1`, `d2`, `d3` Variablen vom Typ `Datum` sind (oder einem Subtyp). Im `new`-Ausdruck könnte auch eine Subklasse von `Datum` stehen. Dann kann man in das Array aber nur Objekte dieser Subklasse speichern (oder einer tieferen Subklasse), sonst bekommt man eine `ArrayStoreException`. Das wird unten noch genauer erläutert.

- Wenn Arrays auf diese Art initialisiert werden, darf man bei `new` keine Array-Größe angeben.

  Diese ergibt sich automatisch aus der Initialisierung.

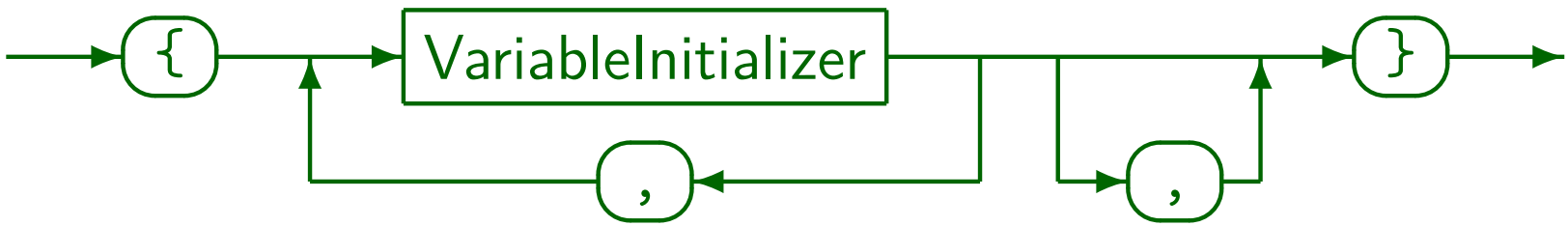
- Man kann Arrays auch anonym erzeugen, d.h. man kann Werte von einem Array-Typ auch in Wertausdrücken aufschreiben, nicht nur speziell in Zuweisungen.

```
y = polynom(new double[] { 1, -2, 4.5 }, x);
```

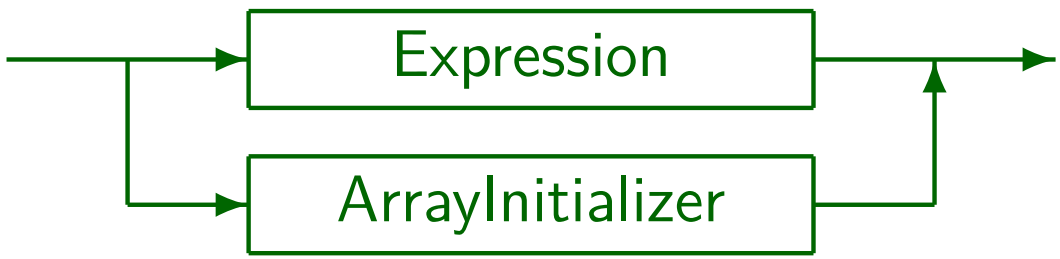
Initialisierung von Arrays (5)

Syntax:

- ArrayInitializer:



- VariableInitializer:



Geschachtelte Array-Initializer werden für mehrdimensionale Arrays (z.B. Matrizen) verwendet (s.u.).

Inhalt

Variable Argument-Anzahl (1)

- Man kann Methoden mit variabler Argument-Anzahl deklarieren, hier wird automatisch ein Array erzeugt.

  Diese Möglichkeit ist neu in Java 5. Variable Argumentanzahlen gab es bereits in der C-Funktion `printf`, man wollte etwas Ähnliches: [java.util.Formatter].
  Man nennt diese Methoden bzw. das Sprach-Feature auch varargs-Methoden.

- Beispiel: Minimum von beliebig vielen ganzen Zahlen:

```
int minimum(int ... args) {
    if(args.length == 0)
        return 0;
    int min = args[0];
    for(int i = 1; i < args.length; i++)
        if(args[i] < min)
            min = args[i];
    return min;
}
```

Variable Argument-Anzahl (2)

- Die variable Argumentanzahl wird durch das Symbol "`...`" nach dem Parameter-Typ spezifiziert.
- Wie man an der Verwendung im Rumpf sieht, ist der Parameter tatsächlich ein Array.
- Der Unterschied liegt darin, dass beim Aufruf dieses Array aus den Argumentwerten automatisch erzeugt wird:

```
int m = minimum(5, 21, 3, 47);
```

- Dies ist äquivalent zu folgendem Aufruf:

```
int m = minimum(new int[] {5, 21, 3, 47});
```

Dieser Aufruf ist auch möglich, wenn die Methode mit variabler Argument-Anzahl deklariert wurde. Die variable Argument-Anzahl ist also nur eine syntaktische Vereinfachung der Array-Initialisierung.

Variable Argument-Anzahl (3)

- Eine Methode kann außer dem Parameter für beliebig viele Argumente noch weitere (normale) Parameter haben, aber der spezielle Parameter muss der letzte sein.

  So ist die Zuweisung zwischen den beim Aufruf angegebenen Werten und den Parametern eindeutig. Nur in dem Fall, dass für den Parameter von Referenztyp nur ein Wert `null` angegeben ist, ist nicht klar, ob dies vielleicht das Array selbst sein soll. Man muss dann einen Cast schreiben.

- Im Beispiel der Minimum-Funktion kann man den Aufruf mit 0 Argumenten ausschließen, indem man ein Argument explizit angibt:

  ```
  int minimum(int erstes, int... rest)
  ```

  Falls man diese Funktion mit einem Argument aufruft, bekommt man als zweites Argument ein leeres Array übergeben (also `rest.length == 0`).

Variable Argument-Anzahl (4)

Für Experten:

- Da `Object` in der Typ-Hierarchie ganz oben steht, und auch primitive Typen mittels "Autoboxing" automatisch in Objekte umgewandelt werden, kann man mit

  ```
  void method(Object... args)
  ```

  eine Methode schreiben, die beliebig viele Argumente von beliebigem Typ akzeptiert.

- So würde aber der folgende Aufruf mehrdeutig sein:

  ```
  m(null);
  ```

  Ist es das erste (und einzige) Argument, oder das Array?

  Man schreibt besser noch eine zweite Variante mit einem Parameter: `void method(Object arg)`. Dies löst auch das Problem, das sonst ein Array-Argument mit entsprechend vielen Argumenten identifiziert wird.

Überladen bei varargs-Methoden (1)

- Da Methoden mit variabler Argument-Anzahl erst in Java 5 nachgerüstet wurden, und man wollte, dass existierende Aufrufe für existierende Methoden (möglichst) nicht gestört werden, wenn man eine varargs-Methode definiert, werden diese erst in einer zweiten Runde berücksichtigt.

  Tatsächlich ist es die dritte Runde. Zwischen den normalen Typanpassungen (widening primitive conversions und impliziten up-casts zu Oberklassen) und der Berücksichtigung variabler Argumentanzahl gibt es noch eine Phase, in der "Boxing/Unboxing" erlaubt sind, z.B. der Übergang von `int` zu `Integer`. Hier war die Trennung insofern noch wichtiger als bei varargs-Methoden, weil auch dieses Feature erst in Java 5 nachgerüstet wurde, und nun z.B. `m(1)` auch `m(Object)` aufrufen kann. Damit hätten aber auch für existierenden Code plötzlich neue Mehrdeutigkeiten entstehen können, die es vorher nicht gab. Im varargs-Fall ändert man den Code dagegen (durch die neue Methode). varargs-Methoden werden auch in der ersten Runde schon berücksichtigt, da aber mit dem entsprechenden Array-Typ als Methode fester Stelligkeit.

Überladen bei varargs-Methoden (2)

```
(1)  class TestOverload5 {
(2)      static void output(double x) {
(3)          System.out.println("double");
(4)      }
(5)
(6)      static void output(int... a) {
(7)          System.out.println("int...");
(8)      }
(9)
(10)     public static void main(String[] args) {
(11)         output(23); // double
(12)     }
(13) }
```

In der ersten Runde wird eine anwendbare Methode gefunden, die Methode mit variabler Argument-Anzahl kommt nicht mehr zum Zug. Sie führt nicht einmal zu einer Mehrdeutigkeit.

Überladen bei varargs-Methoden (3)

```
(1)  class TestOverload6 {
(2)      static void output(int.. a) {
(3)          System.out.println("int...");
(4)      }
(5)
(6)      static void output(int i, int... a) {
(7)          System.out.println("int, int...");
(8)      }
(9)
(10)     public static void main(String[] args) {
(11)         output(1, 2); // Mehrdeutig!
(12)     }
(13) }
```

Eigentlich könnte man die zweite Version von output als spezifischer ansehen, aber das tut Java nicht. Man erhält die Fehlermeldung "reference to output is ambiguous".

Inhalt

Mehrdimensionale Arrays (1)

- Zweidimensionale Arrays sind in der Mathematik als Matrizen bekannt, z.B.:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

- Eine solche Matrix könnte man folgendermaßen anlegen:

```
int[][] matrix = new int[3][3];
matrix[0][0] = 1;
matrix[0][1] = 2;
matrix[0][2] = 3;
matrix[1][0] = 4;
matrix[1][1] = 5;
...
```

Mehrdimensionale Arrays (2)

- Bei einem zweidimensionalen Array wird ein Eintrag also durch zwei Zahlen identifiziert.

  Die Darstellung mit Zeilen und Spalten ist nur eine Visualisierung.
  Die meisten Programmierer würden wohl den ersten Index für die Zeile nehmen, und den zweiten für die Spalte — aber wenn man bei der Ausgabe konsistent ist, könnte man es auch umgekehrt machen.
  Üblicherweise ändert sich bei einem "natürlichen Durchlauf" der am weitesten hinten stehende Index am schnellsten (so wie sich beim Hochzählen von Dezimalzahlen die letzte Ziffer am schnellsten ändert).

- Formal ist die Matrix eine Abbildung

  $$\{0,1,2\} \times \{0,1,2\} \to \texttt{int} : \{-2^{31}, \ldots, 2^{31}-1\}$$

- Jeder Index beginnt bei 0 und endet bei der Länge minus 1.

  Die Länge erhält man als "`matrix.length`" für die erste Dimension, und "`matrix[i].length`" für die zweite Dimension (mit `i` zwischen 0 und 2, s.u.).

Mehrdimensionale Arrays (3)

- Mehrdimensionale Arrays werden meist mit geschachtelten Schleifen verarbeitet.
- Ausgabe der 3×3-Matrix:

```
for(int i = 0; i < 3; i++)
    for(int j = 0; j < 3; j++) {
        System.out.print(matrix[i][j]);
        if(j != 2) // Nicht letzte Spalte
            System.out.print(", ");
        else
            System.out.println();
    }
```

Alternative Lösung auf nächster Folie. Stilistisch ist "`j != 2`" nicht schön, weil darin nicht die Array-Länge 3 steht. Vielleicht sollte man "`j+1 < 3`" schreiben (es gibt noch einen nächsten Durchlauf): sieht auch komisch aus.

Mehrdimensionale Arrays (4)

- Alternative Lösung zur Ausgabe der 3×3-Matrix:

```
for(int i = 0; i < 3; i++) {
    for(int j = 0; j < 3; j++) {
        if(j > 0)
            System.out.print(", ");
        System.out.print(matrix[i][j]);
    }
    System.out.println();
}
```

Hier wird also vor Ausgabe des Elementes getestet, ob es nicht das erste Element der Zeile ist, und dann ggf. das Trennzeichen ausgegeben.
Der Zeilenvorschub wird jeweils nach der inneren Schleife ausgeführt.
Damit die Matrix schön aussieht, muss man für jeden Eintrag gleich viele Zeichen schreiben, das geht mit `System.out.printf("%3d", matrix[i][j]);` falls man mindestens 3 Zeichen pro Zahl ausgeben will. [Java API: Formatter]

Mehrdimensionale Arrays (5)

- In Java ist ein zweidimensionales Array eigentlich ein normales (eindimensionales) Array, das selbst wieder Referenzen auf eindimensionale Arrays als Elemente enthält.

  Streng genommen hat Java also gar keine mehrdimensionalen Arrays.

- So müssen bei der Matrix z.B. nicht alle Zeilen gleich viele Spalten haben.

- Dies erklärt auch, warum man mit

  - `matrix.length`
    die Größe der ersten Dimension erhält, und mit
  - `matrix[i].length`
    die Länge der zweiten Dimension bekommt, also einen konkreten Index für die erste Dimension angeben muss.

Mehrdimensionale Arrays (6)

- Wenn man ein zweidimensionales Array z.B. mit

```
int[][] matrix = new int[3][3];
```

erzeugt, so bewirkt dies eigentlich Folgendes:

```
int[][] matrix = new int[3][];
matrix[0] = new int[3];
matrix[1] = new int[3];
matrix[2] = new int[3];
```

Der `new`-Aufruf nur mit Größen für ein Anfangsstück der Dimensionen ist syntaktisch möglich: Die übrigen `[]` dienen dann nur der Typ-Angabe.
Da Java immer mit Referenzen arbeitet, hat es nicht wirklich zweidimensionale Arrays. In Sprachen wie C++ gibt es dagegen einen Unterschied zwischen einem echten zweidimensionalen Array und einem Array, das Zeiger auf Arrays enthält. Beim echten zweidimensionalen Array sind alle Zeilen gleich lang. Auch dieses Array ist tatsächlich ein Array von Arrays, aber die Arrays sind direkt in dem äußeren Array gespeichert.

Mehrdimensionale Arrays (7)

- Natürlich kann man mehrdimensionale Arrays auch initialisieren:

```
int[][] matrix = {
        { 1, 2, 3 },
        { 4, 5, 6 },
        { 7, 8, 9 }
};
```

- Beim Zugriff muss man für jeden Index ein eigenes Klammerpaar angeben. Das von der mathematischen Notation $M_{i,j}$ her naheliegende

```
matrix[i, j] // falsch!
```

  ist ein Syntaxfehler.

  In C++ wäre es auch falsch (allerdings nur in der Deklaration ein Syntaxfehler, sonst würde aber nicht das Erwartete tun). In Pascal würde es so funktionieren.

Inhalt

Subklassen und Arrays (1)

- Wenn `Unter` eine Unterklasse von `Ober` ist, wird in Java auch `Unter[]` als Untertyp von `Ober[]` behandelt.
- D.h. folgende Zuweisung ist möglich:

```
Ober[] a = new Unter[5];
```

- Solange man das Array nur Referenzen auf Objekte der Klasse `Unter` (oder ihrer Subklassen) einträgt, geht alles gut.
- Versucht man aber ein ein `Ober`-Objekt in das Array zu speichern, so erhält man eine `ArrayStoreException`.

  Dies ist ein Fehler, der nicht zur Compilezeit erkannt werden kann, denn der statische Typ von `a` ist ja `Ober[]`. Der dynamische Typ ist aber `Unter[]`. Der Compiler muss also einen entsprechenden Laufzeit-Test generieren (für jede Zuweisung an ein Array-Element). Wäre die Klasse `Ober` als `final` deklariert (s.u.), könnte der Compiler diesen Test einsparen.

Subklassen und Arrays (2)

- Das folgende Beispiel zeigt, dass man diesen Test wirklich braucht:

```
Unter[] uArr = new Unter[5]; // Ok
Ober[] oArr = uArr;          // Ok
oArr[0] = new Ober();        // ArrayStoreExeption
```

- Würde man dies zulassen, so würde man anschließend beim Zugriff auf `uArr[0]` ein Objekt der Oberklasse `Ober` bekommen.

- Damit wäre auch die statische Typisierung nicht mehr sicher.

  Es ist in Ordnung, wenn der dynamische Typ ein Untertyp des statischen Typs ist (Substitutionsprinzip). Umgekehrt ist es aber nicht in Ordnung. Es wären dann z.B. Zugriffe auf Attribute möglich, die das Objekt gar nicht hat (man würde vermutlich irgendwelchen Müll aus dem Speicher bekommen).