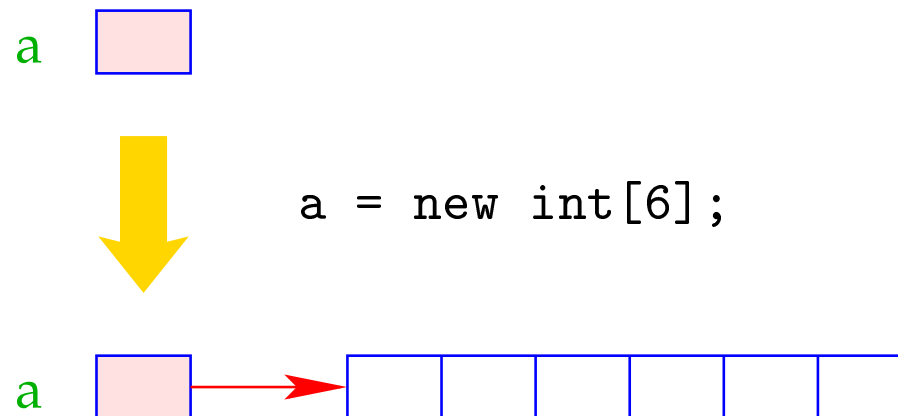


Beispiel: Einlesen eines Felds

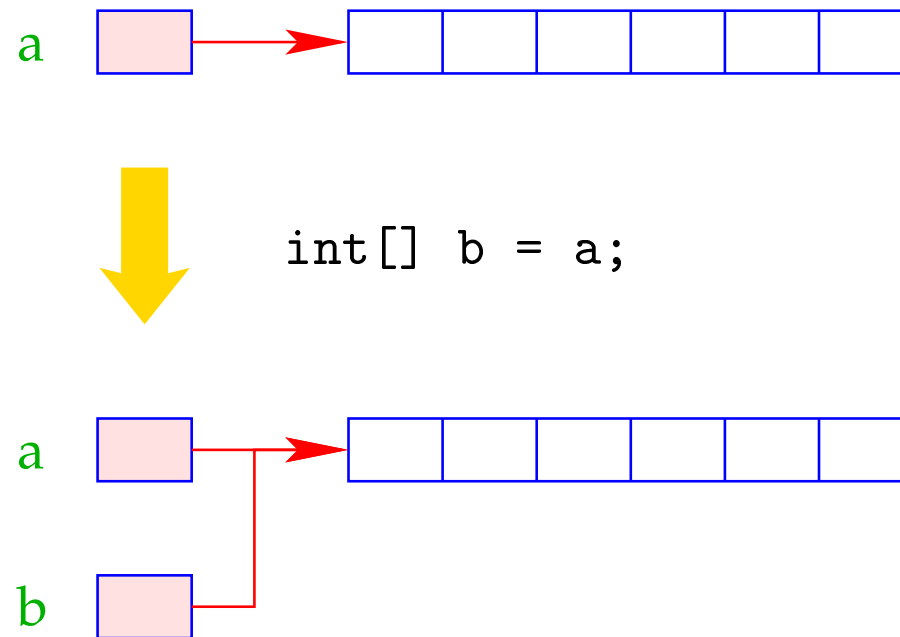
```
int[] a; // Deklaration
int n = read();

a = new int[n];
           // Anlegen des Felds
int i = 0;
while (i < n) {
    a[i] = read();
    i = i+1;
}
```

- `type [] name` ; deklariert eine Variable für ein Feld (`array`), dessen Elemente vom Typ `type` sind.
- Alternative Schreibweise:
`type name []` ;
- Das Kommando `new` legt ein Feld einer gegebenen Größe an und liefert einen `Verweis` darauf zurück:




- Der Wert einer Feld-Variable ist also ein Verweis.
- `int[] b = a;` kopiert den Verweis der Variablen a in die Variable b:



- Die Elemente eines Felds sind von 0 an durchnummeriert.
- Die Anzahl der Elemente des Felds `name` ist `name.length`.
- Auf das i -te Element des Felds `name` greift man mittels `name[i]` zu.
- Bei jedem Zugriff wird überprüft, ob der Index erlaubt ist, d.h. im Intervall $\{0, \dots, \text{name.length}-1\}$ liegt.
- Liegt der Index außerhalb des Intervalls, wird die `ArrayIndexOutOfBoundsException` ausgelöst (↑`Exceptions`).

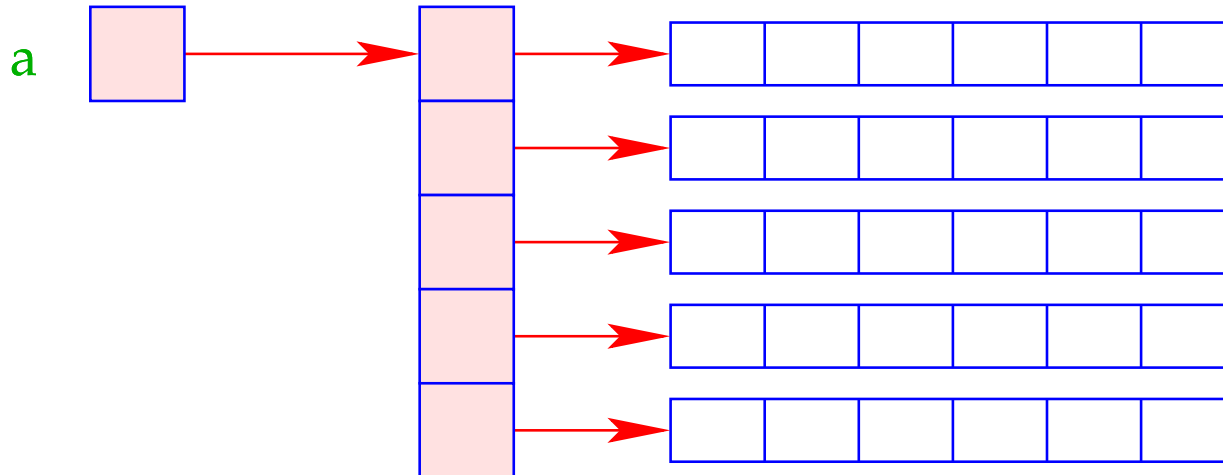
Mehrdimensionale Felder

- **Java** unterstützt direkt nur ein-dimensionale Felder.
- Ein zwei-dimensionales Feld ist ein Feld von Feldern ...

a 



`a = new int [5] [6];`



5.5 Mehr Kontrollstrukturen

Typische Form der Iteration über Felder:

- Initialisierung des Laufindex;
- `while`-Schleife mit Eintrittsbedingung für den Rumpf;
- Modifizierung des Laufindex am Ende des Rumpfs.

Beispiel (Forts.): Bestimmung des Minimums

```
int result = a[0];  
int i = 1;      // Initialisierung  
while (i < a.length) {  
    if (a[i] < result)  
        result = a[i];  
    i = i+1;    // Modifizierung  
}  
write(result);
```

Mithilfe des for-Statements:

```
int result = a[0];  
for (int i = 1; i < a.length; ++i)  
    if (a[i] < result)  
        result = a[i];  
write(result);
```


Allgemein:

for (**init**; **cond**; **modify**) stmt

... entspricht:

{ **init** ; while (**cond**) { **stmt modify** ; } }

... wobei ++i äquivalent ist zu i = i+1 **:-)**

Warnung:

- Die Zuweisung $x = x-1$ ist in Wahrheit ein **Ausdruck**.
- Der Wert ist der Wert der rechten Seite.
- Die Modifizierung der Variable x erfolgt als **Seiteneffekt**.
- Der Semikolon “;” hinter einem Ausdruck wirft nur den Wert weg ... :-)

⇒ ... fatal für Fehler in Bedingungen ...

```
boolean x = false;  
if (x = true)  
    write("Sorry! This must be an error ...");
```

- Die Operatoranwendungen `++x` und `x++` inkrementieren beide den Wert der Variablen `x`.
- `++x` tut das, **bevor** der Wert des Ausdrucks ermittelt wird (**Pre-Increment**).
- `x++` tut das, **nachdem** der Wert ermittelt wurde (**Post-Increment**).
- `a[x++] = 7;` entspricht:
$$\begin{aligned} a[x] &= 7; \\ x &= x+1; \end{aligned}$$
- `a[++x] = 7;` entspricht:
$$\begin{aligned} x &= x+1; \\ a[x] &= 7; \end{aligned}$$

Oft möchte man

- Teilprobleme **separat** lösen; und dann
- die Lösung **mehrfach** verwenden;

⇒ Funktionen, Prozeduren

Beispiel: Einlesen eines Felds

```
public static int[] readArray(int n) {  
    // n = Anzahl der zu lesenden Elemente  
    int[] a = new int[n]; // Anlegen des Felds  
    for (int i = 0; i < n; ++i) {  
        a[i] = read();  
    }  
    return a;  
}
```

- Die erste Zeile ist der **Header** der Funktion.
- `public` sagt, wo die Funktion verwendet werden darf (↑kommt später **`:-)`**)
- `static` kommt ebenfalls später **`:-)`**)
- `int []` gibt den Typ des Rückgabe-Werts an.
- `readArray` ist der Name, mit dem die Funktion aufgerufen wird.
- Dann folgt (in runden Klammern und komma-separiert) die Liste der **formalen Parameter**, hier: `(int n)`.
- Der Rumpf der Funktion steht in geschwungenen Klammern.
- `return expr` beendet die Ausführung der Funktion und liefert den Wert von **`expr`** zurück.

- Die Variablen, die innerhalb eines Blocks angelegt werden, d.h. innerhalb von “{” und “}”, sind nur innerhalb dieses Blocks **sichtbar**, d.h. benutzbar (**lokale Variablen**).
- Der Rumpf einer Funktion ist ein Block.
- Die formalen Parameter können auch als lokale Variablen aufgefasst werden.
- Bei dem Aufruf `readArray(7)` erhält der formale Parameter `n` den Wert `7`.

Weiteres Beispiel: Bestimmung des Minimums

```
public static int min (int[] a) {  
    int result = a[0];  
    for (int i = 1; i < a.length; ++i) {  
        if (a[i] < result)  
            result = a[i];  
    }  
    return result;  
}
```


... daraus basteln wir das **Java**-Programm Min:

```
public class Min extends MiniJava {
    public static int[] readArray (int n) { ... }
    public static int min (int[] a) { ... }
    // Jetzt kommt das Hauptprogramm
    public static void main (String[] args) {
        int n = read();
        int[] a = readArray(n);
        int result = min(a);
        write(result);
    }    // end of main()
}      // end of class Min
```

- Manche Funktionen, deren Ergebnistyp `void` ist, geben gar keine Werte zurück – im Beispiel: `write()` und `main()`. Diese Funktionen heißen **Prozeduren**.
- Das Hauptprogramm hat immer als Parameter ein Feld `args` von `String`-Elementen.
- In diesem Argument-Feld werden dem Programm Kommandozeilen-Argumente verfügbar gemacht.

```
public class Test extends MiniJava {  
    public static void main (String [] args) {  
        write(args[0]+args[1]);  
    }  
} // end of class Test
```

Dann liefert der Aufruf:

```
java Test "Hel" "lo World!"
```

... die Ausgabe: **Hello World!**