Arrays – continued

Kees Huizing
September 2009

Multi-dimensional arrays

- to model grids, matrices, screens, tables, etc.
- example: student scores
  - student nr. *i* has scored an *s* for course nr. *c*

course → student ↓	0	1	2	3
0	good	avg	good	NV
1	avg	avg	bad	exc
2	avg	avg	avg	avg

course →	0	1	2	3
0	good	avg	good	NV
1	avg	avg	bad	exc
2	avg	avg	avg	avg

```
String[][] table;

table = new String[4][3];

table[0][0] = "good";
table[1][3] = "exc";
for (int c=0; c < table.length; c++) {
    table[2][c] = "avg";
}
for (int r=0; r < table[1].length; r++) {
    table[r][1] = "avg";
}
//etc.
```

```
//print table
for (int r=0; r < table.length; r++) {
    for (int c=0; c < table[c].length; c++) {
        System.out.print( table[r][c] );
    }
    System.out.println();
}
```

2-dim array is an array of arrays

3

Two-dimensonal arrays

- conventions
  - pixels, geometry, etc.: first index is x-coordinate, second is y-coordinate
  - matrices: first index is row, second is column
  - different :-(
- it doesn’t matter what you choose, as long as it is consistent

4

Ragged arrays

- what does this produce?

```
0
10 11
20 21 22
30 31 32
40 41 42 43
```

```
//declaration
int[][] triangle;
//creation
triangle = new int[5][];
for (int r = 0; r < 5; r++) {
    triangle[r] = new int[r+1];
}
//filling
for (int r = 0; r < triangle.length; r++) {
    for (int c = 0; x < triangle[r].length; c++) {
        triangle[r][c] = 10*r + c;
    }
}

//printing
for (int r = 0; r < triangle.length; r++) {
    for (int c = 0; x < triangle[c].length; c++) {
        System.out.print(triangle[r][c] + " ");
    }
    System.out.println();
}
```


5

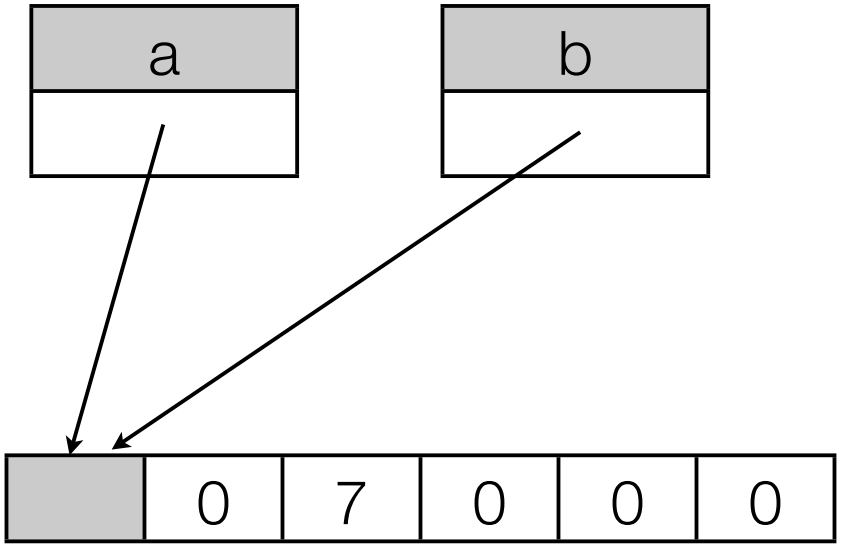
Aliasing

- **int[ ] a;** declares a variable that holds a reference to an array
- **int[ ] b;** ditto
- **a = new int[5];** creates 5 variables and a “points to them”
- **b = a;** makes **b** point to the same 5 variables **a** points to
- **System.out.println(b[1]);**
  - will print **0**
- **a[1] = 7;**
  System.out.println(a[1]);
  - will print **7**
- **System.out.println(b[1]);**
  - will print **7**

6

Aliasing

- **int[ ] a;** declares a variable that holds a reference to an array
- **int[ ] b;** ditto
- **a = new int[5];** creates 5 variables and **a** "points to them"
- **b = a;** makes **b** point to the same 5 variables **a** points to
- **System.out.println(b[1]);**
  - will print **0**
- **a[1] = 7;**
  System.out.println(a[1]);
  - will print **7**
- **System.out.println(b[1]);**
  - will print **7**



7

Aliasing ctd

- assignment **b = a** only copies *reference*
- to copy *content* use loop
- or **b = Arrays.copyOf(a, 5);**

8

Frequency count

- Suppose we want to count how often each grade (1-10) occurs in a list of 100 grades

```
int[] frequencies = new int[11];
//assert all elts of frequencies are 0

for (int i = 0; i<100; i++) {
    int grade = scanner.nextInt();
    assert 1<=grade && grade<=10;


}
```

Frequency count

- Suppose we want to count how often each grade (1-10) occurs in a list of 100 grades

```
int[] frequencies = new int[11];
//assert all elts of frequencies are 0

for (int i = 0; i<100; i++) {
    int grade = scanner.nextInt();
    assert 1<=grade && grade<=10;

    frequencies[ grade ] += 1;
}
```


10

Array as return type

Calculate frequency of digits in number *n*

```
int[] digitFreq(int n) {
    assert n>=0;
    int[] freqs = new int[10];
    //assert all elts of freqs are 0;

    int m=n;
    // special case
    if (m==0) { freqs[0] += 1; }

    while (m>0) {
        int d = m % 10;
        assert 0<=d && d<10;
        freqs[d] += 1;
        m /= 10;
    }

    return freqs;
}
```

11

More on arrays

- quick initialization:

  String[ ] beers = {“Duvel”, “Leffe Dubbel”, “Hoare Tripel”, };

- elsewhere:

  trappists = **new String[ ]** { “Orval”, “Chimay”, “Westmalle”, “Rochefort”,
  “Westvleteren”, “Achel”, “La Trappe”};

12

Specification

Specification

```
int[] digitFreq(int n)
    assert n>=0;
    int[] freqs = new i
    //assert all elts of
    ...
```

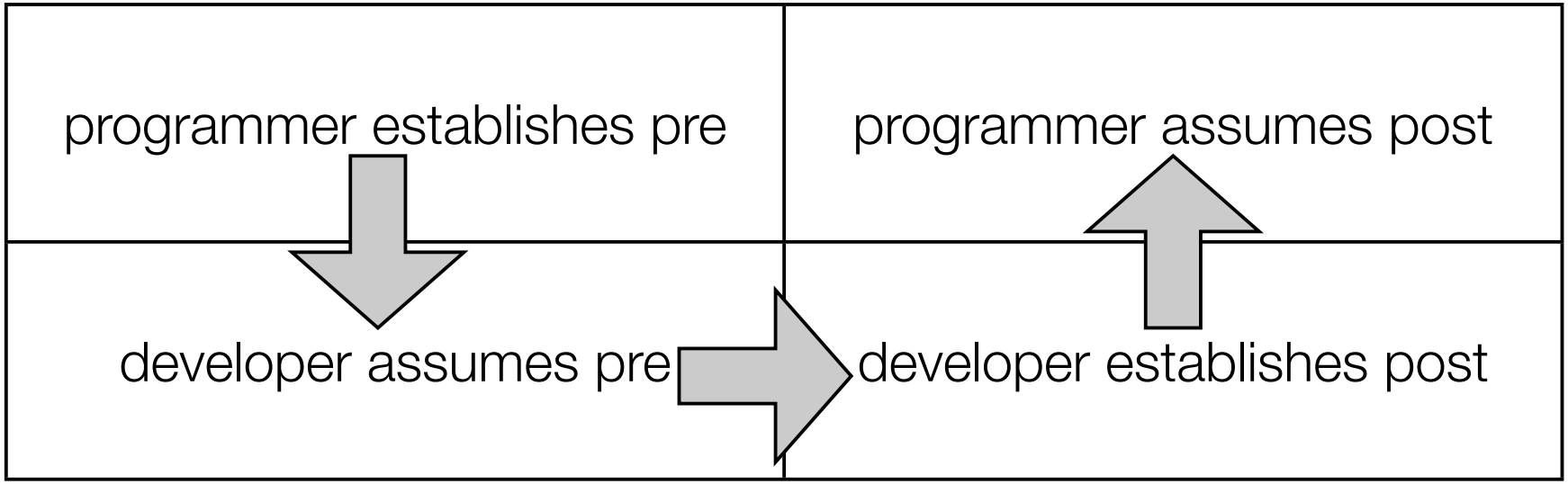
how do you know this?

- method builder has to assume that method is called correctly:
  - **precondition (requires clause)**
- method caller has to assume that method produces correct result
  - **postcondition (ensures clause)**

14

Specification

- See method as a *product*
  - developer offers a product
  - programmer uses product
  - delivery contract with rights and duties of parties



15

Specification example

What are the pre- and postcondition?

```
//pre: n>=0
//post: \result == r && r*r <= n && n < (r+1)*(r+1)
f(int n) {
    int r = 0;
    while (r+1)*(r+1) <= n {
        r +=1;
    }
 }
```

pre: $n \geq 0$

post: $r^2 \leq n < (r+1)^2$
or $r = \lfloor \sqrt{n} \rfloor$ (floor)

Usage

```
assert n >= 0;
w = f( 10 );
assert w*w <= n && n < (w+1)*(w+1);
```

apply pre- and post to *actual parameters*

16

specification format

@pre or @requires: precondition

@post or @ensures: postcondition

@inv: loop invariant

@modifies: instance vars that are possibly changed

@uses: instance variabels that are consulted