Arrays (JFS 13)

- Motivation – bulk data handling and processing
- Declaring and using
- Basic algorithms
- More advanced techniques

Arrays: Motivation (JFS 13)

- Suppose a meteorologist needs to total rainfall over a period for each day of the week
- By convention, say:
  0 = Monday, 1 = Tuesday, ..., 5 = Saturday, 6 = Sunday
- The rainfall for each day is to be recorded as a floating-point number in mm
- The total rainfall per day could be stored in individual variables:

```
float rain0, rain1, rain2, rain3, rain4,
      rain5, rain6;
```

- This is clumsy, and would become unworkable if the rainfall were to be recorded for *every day of a year*!
- It needs a collection of identical kinds of items
  - Each identified by a particular index or subscript
  - For example the day number, size code
- Collections of identical items occur frequently, for example:
  - sales of products, indexed by code 0, 1, 2, ...
  - diameters of bolts, indexed by size 0, 1, 2, ...
  - marks for a test, indexed by mark 0, 1, 2, ...

- For such a situation, Java provides the ***array***
  - A collection of identical kinds of elements
  - Indexed by a subscript
- An array with a single subscript is called a list, vector or one-dimensional array
- The number of items in an array is fixed by its declaration
  - The array cannot grow or shrink
- [The libraries contain other data structures that *can* grow and shrink.]

Declaring Arrays

- An array is created in two steps:
  - 1) A variable is declared to hold the array:

```
float rain[];     // name an array of floats
```

  - 2) Space is allocated for the array:

```
rain = new float[7];
```

- Both steps are needed
  - Until space has been allocated an array is only just a name
- It is quite common to do both steps at the same time:

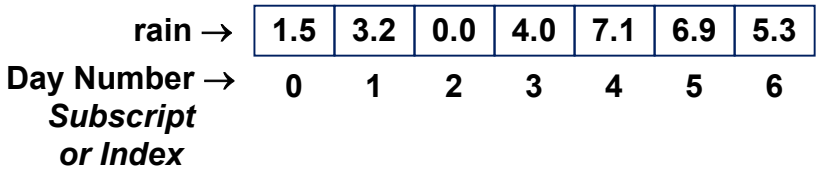
```
float rain[] = new float[7];
```

- [If a global variable then also (usually) `private`.]
- Note: Since an array is *non-primitive data*, the array variable rain actually holds a reference/pointer to the allocated memory

- Notice the use of brackets
  - They indicate that a subscript will be used
- Actually the brackets can be written after the type or name when an array is declared:

```
float rain[];
float[] rain;
```

- The first form is closer to how subscripts are used, though the second form is used in [Bell and Parr]
- An array like `rain` can hold 7 values
- The size of an array is fixed once space has been allocated
  - Can refer to it in Java code like this: `rain.length`



- As shown in the figure, array subscripts start at 0
  - The first element is numbered 0
  - The last is numbered 6
- It is easy to mistakenly expect such an array to be numbered from 1 to 7 since it has 7 values
- A run-time error will occur if the program tries to refer to a subscript is out of range
  - Here, less than 0 or greater than 6
- Once an array has been declared and space has been allocated, the values of its elements can be set and used

Using Arrays

- Array elements can be used directly:

```
rain[5]          // Rainfall for Saturday (5)
rain[0] + rain[1] + rain [2]
          // Total rainfall for Mon - Weds
```

- "Indexed variables" can be used in expressions and on the left of assignment statements:

```
rain[4] = 6.2; // Set rainfall for Friday
float mtwTotal = rain[0] + rain[1] + rain[2];
```

  - Just like any variable
- The subscript or index can be an *expression:*

```
int d = 3;      // d is a "day of interest"
rain[d] = 4.7;
total = rain[d-1] + rain[d] + rain[d+1];
```

  - *A subscript expression is re-evaluated at every access*

- Since arrays contain identical kinds of elements, they are particularly suitable for *repetitive* operations
- Often a `for` loop is appropriate for this
  - e.g. to add up all the elements in an array:

```
float total = 0.0f; // For week total
for (int day = 0; day < rain.length; day++) {
                                  // go through days
  total = total + rain[day]; // add one day
}
```

Note: Refers to a different element of rain at each repetition of loop body

- To calculate the average rainfall, calculate the total and then do:

```
float average = total / rain.length;
```

- Note the use of `rain.length`
  - The code does *not need adpating* if we change the array size

Declaring Arrays (again)

- Since the programmer must decide the size of an array in advance, it is good practice to use a constant for this
- A constant is a 'variable' declared as `final`; the value given to it cannot be changed
- As an example of a constant, the size of an array can be declared and used as:

```
final int days = 7;  // The number in a week
float rain[] = new float[days];
                     // Use this as the size
```

- This has several advantages:
  - The meaning of the array size might not be obvious: e.g. there might be different arrays of size 7 - giving the size a *name* makes the *intention* clearer
  - There might be several arrays that *must have the same size*: redefining one constant will affect *all sizes consistently*

Initialising Arrays

- The elements of a new array will probably contain nothing useful
  - So the program must usually give each item a specific value initially
- If the array contains numbers, a good initial value is probably zero:

```
rain[0] = 0.0f; rain[1] = 0.0f;
rain[2] = 0.0f; rain[3] = 0.0f;
rain[4] = 0.0f; rain[5] = 0.0f;
rain[6] = 0.0f;
```


- However this needs one assignment per array element
  - Not a good idea if the array has many elements!
- A much better idea is to use a *loop* to carry out such a repetitive task:

```
for (int day = 0; day < days; day++) // Week
    rain[day] = 0.0f;  // Initialise one day
```

- Notice that the number of `days` has been used here as a constant, instead of writing 7 into the program
- This makes it clearer that the loop deals with all array elements
- Using 7 literally would be problematic if the size of array were changed later (perhaps not in this application!)

- A loop is also useful if the initial values can be decided *according to some rule*
- Consider an array that stores the 12-times table for multiplier 0 to 10:
- The initialisation might be:

```
int twelveX[] = new int[11];
for (multiplier = 0; multiplier < 11;
                              multiplier++) {
  twelveX[multiplier] = multiplier * 12;
}
```

twelveX →	**0**	**12**	**24**	**36**	**…**	**108**	**120**
Subscript →	**0**	**1**	**2**	**3**	**…**	**9**	**10**

- Sometimes the initial values are irregular and have to be given directly:

```
// ages of children 0, 1, 2
age[0] = 2; age[1] = 3; age[2] = last + 4
```

- If these values are to fill the whole array, can conveniently use an *array initializer*:

```
int age[] = {2, 3, last + 4};
```

- Note that the size of such an array is implied by the number of initial values
  - And it replaces the declaration, space allocation and initial assignments
- Other ways of getting initial values into an array:
  - Ask the user to type them into a text field in a window, then read and store these values
  - Read the values from a file on disc

Using Arrays – more algorithms

- It is also often necessary to search through an array according to some criterion, e.g. find the day with the highest rainfall:

```
int wettest = 0;      // Assume day 0 the wettest
for (int day = 0; day < rain.length; day++) {
                      // Go through the days
  if (rain[day] > rain[wettest]) // Current wetter?
    wettest = day;               // Yes, note it
}
```

- At the end of this loop, a value for `wettest` like 4 means that day 4 had the highest rainfall (or equal wettest)
- This is a *common pattern*:
  - Wettest, driest, number of very wet/quite dry days, find a completely dry day, and many other situations

- How many quite dry days?

```
int dryDays = 0;      // To count the days
for (int day = 0; day < rain.length; day++) {
                      // Go through the days
  if (rain[day] < 1.0f)  // Quite dry day?
    dryDays++;           // Yes, count it
}
```

- Sometimes it is necessary to search through an array for a *particular value*, e.g. is there a day that had no rain?
  - Special case: There might be no such day

```
int dryDay = -1;            // Day initially unknown
for (int day = 0; day < rain.length; day++) {
                            // Go through the days
  if (rain[day] == 0.0f)    // No rain this day?
    dryDay = day;           // Note this as a dry day
}
```

- So if there were no rain on day 5, `dryDay` would be set to 5
  - And if there is no dry day, then `dryDay` remains -1

- There are two potential complications here:
  - If were no rain on days 3 and 5, `dryDay` would be set to 5 (i.e. the *last* such case)
  - If *every* day had some rain, `dryDay` would stay at -1; a later part of the program would have to check for this condition
- When searching an array, often the *first* suitable value is wanted
  - Continuing to search the array wastes time (especially if it is large)
- A `for` loop can be undesirable because it searches *all* the array elements

Short-cutting searches: two solutions

- First solution:
- Instead a `while` loop may be better because it can finish as soon as the required value is found:

```
int dryDay = -1;            // Day initially unknown
int day = 0;                // Start with first day
while (dryDay == -1 &&      // Still looking and ...
     day < rain.length) {   // not reached array end?
  if (rain[day] == 0.0f)    // No rain this day?
      dryDay = day;         // Note this as a dry day
  day++;                    // Move to next day
}
```


- Notice that there are *two* reasons that such a loop terminates:
  - Finding the desired condition
  - or reaching the end of the array
- Also note that a loop like this has to explicitly advance the loop counter (`day`)
  - Forgetting to do this will mean the program loops indefinitely!

- Second solution:
- We can use a `for` loop with a `break;` statement:

```
int dryDay = -1;            // Day initially unknown
for (int day = 0; day < rain.length; day++) {
                            // Go through the days
  if (rain[day] == 0.0f) { // No rain this day?
    dryDay = day;           // Note this as a dry day
    break;
  }
}
```

- Executing `break;` causes an immediate exit from the *closest enclosing loop*:
  - Best of both worlds?
  - Some programmers disagree with using `break;`

Arrays as Parameters

- Just as a method can take simple types like integers or floats as parameters, it can take any kind of object including arrays
- This is where `array.length` is particularly useful, because the method cannot know in advance how long the array will be
- For example, a general method can be written to set an array of floats to zero:

```
// Take any float array, set elements to 0
private void zeroise(float arr[]) {
  for (int s = 0; s < arr.length; s++)
                    // Go through all elements
    arr[s] = 0.0f; // Set element s to zero
}
```

- Another method might copy the contents of one array to a second array if they are the same length:

```
// Copy array1 to array2 if compatible
private void copy(float array1[], float array2[]) {
  int len1 = array1.length; // First array length
  int len2 = array2.length; // Second array length
  if (len1 == len2) {       // Arrays same length?
    for (int s = 0; s < len1; s++)
                          // Go through all elements
      array2[s] = array1[s]; // Copy element s
  }
}
```


- Notice that this is *different* from assigning the first array to the second:

```
anotherArray = anArray;
```

  - which makes `anotherArray` the *same* array as `anArray`:
  - `anArray` and `anotherArray` become pointers to the *same* memory elements
- In contrast:

```
copy(anArray, anotherArray);
```

  - gives two entirely independent copies of the same values
  - Copying keeps the arrays separate

Other Kinds of Arrays (JFS 13, 14)

- Arrays are not restricted to holding numbers:

```
char vowels[] = {'a', 'e', 'i', 'o', 'u'};
String name[] = {"Joe", "Sue", "Pat"};
boolean married[] = {false, true, false};
```

  - Perhaps `name` and `married` are "parallel arrays", holding corresponding information
- In fact arrays can hold *any kind of object*, such as:
  - Records – see example on next slide
  - Graphical items like buttons:

```
JButton[] controls = new JButton[10];
```


- Given the definition of a telephone directory entry earlier, a whole telephone directory might be declared and set up as:

```
Entry directory[] = new Entry[1000];
directory[0] =      // Set first subscriber
     new Entry("Jo Smith", "467420", true);
directory[1] =      // set second subscriber
     new Entry("Mary Brown", "467423", false);
...
```

- Obviously, to fill in the complete directory would be a lot of work
  - Normally such information would be read from a file
- Working with the entry data can look complex:

```
directory[i].setUnlisted(true);
g.drawString(directory[i].getName(), x, y);
```

- Listing the whole (listable) directory in text area `output`:

```
output.setText("");          // Clear output
for (int n = 0; n < directory.length; n++) {
                             // Go through the entries
  if (!directory[n].isUnlisted()) { // Check status
    output.append(directory[n].getName() +
           ": " + directory[n].getNumber() + "\n");
  }
}
```


- Searching for the number for a subscriber:

```
String name = nameText.getText();  // To be found
int number = -1;             // Number initially unknown
for (int n = 0; n < directory.length; n++) {
                             // Go through the directory
  if (name.equals(directory[n].getName())) {
                             // Name match?
    number = directory[n].getNumber();
                             // Note the number
    break;                   // And exit the search
  }
}
```

- Note:
  - `.equals` requires a precise match
  - `.equalsIgnoreCase` is more flexible
  - Partial match would be even better

Two-dimensional Arrays

- The elements of an array can also be arrays!
- This is best thought of as a table, also called a two-dimensional array
- Examples of two-dimensional arrays are:
  - A contour map, where each pair of x and y coordinates gives a height
  - A chess-board, where each square gives the name of a piece occupying it
  - Sales in a chain of retail stores, where one subscript identifies the shop and a second subscript identifies the product code; array elements give the sales of each product per store

- The contour map might be declared as:

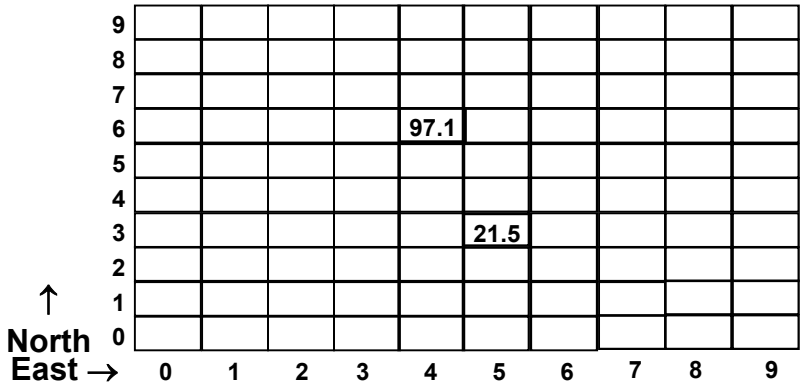
```
final int EASTINGS = 1000;    // Grid dimensions
final int NORTHINGS = 1000;
float height[][] =
            new float[EASTINGS][NORTHINGS];
```

  - `EASTINGS` is the west-east coordinate
  - `NORTHINGS` is the south-north coordinate
  - `height` is in metres – one per x, y coordinate
- Heights in the contour map would be set and used as follows:

```
height[5][3] = 21.5f;    // Set height at grid (5,3)
hilltop = height[4][6]; // Get height at grid (4,6)
```

- As can be seen, two-dimensional arrays are like one-dimensional arrays except that they have a pair of subscripts
- See next slide

- We can picture the two-dimensional array like this:



- Note: There would be a value in every element of the grid

- Nested `for` loops are often used to process two-dimensional arrays
- For example: If the retail chain sales table is declared like this:

```
final int STORES = 100;     // Size of chain
final int PRODUCTS = 1000; // How many stocked
int sales[][] = new int[STORES][PRODUCTS];
```

- (And the sales have all been recorded in the array)
- Then the total sales can be calculated by:

```
int totalSales = 0;
for (int s = 0; s < STORES; s++)
  for (int p = 0; p < PRODUCTS; p++)
    totalSales = totalSales + sales[s][p];
```

Array Example

Rainfall Array

End of section