

Arrays

- Data structures of related data items with the same type.
- A group of contiguous memory items with the same name and the same type
- Indexing (indicates a position in the array) begins at 0
 - Also called a subscript
 - Must be an integer or an integer expression
- lvalue – anything that can be used on the left side of an assignment statement
- Arrays are always initialized to 0, `false`, or `null`
- The compiler prevents a program from going out of bounds; not true in C!
- See page 231

Array Methods and Attributes

- Fig 5-3 - 5-5
- `c.length` – the length of array `c`
- To create an array:

```
int c[] = new int[12]
```

declares and allocates an array `c` - it's initialized to all zeros

```
int c[];  
c = new int[12];
```

`int c[12]` is a syntax error

- To initialize an array

```
int n[] = {10, 20, 30, 40, 50};
```

creates a 5 element array
- Any expression can be used to set an array:

```
s[i] = 2 + 2 * i;
```

sets an array element to twice its index plus 2

```
s[2*i+1] = 5;
```

uses an expression for the index
- Note: line 17 of figure 5.5 - `s[]` has class scope
- Final qualifier Fig 5.7
 - Used to create a variable that cannot change, a so called “constant variable”
 - Defined constant, named constant, read-only variables are a better names

```
final int ARRAYSIZE = 10;
```
 - An attempt to modify a final variable will result in a syntax error
 - A convention is all caps for the final name
- Fig 5-8 uses a for loop to sum the elements of an array
- Figure 5.9 uses an array to calculate a frequency (food on a scale of 1 to 10)

- `frequency[]` is initialized with a size of 11 to index the responses from 1 to 10; element 0 is not used, otherwise an index adjustment is needed
 - Lines 18 - 19: the values in the `responses` array are used to index the frequency array
 - The frequency array is automatically initialized to zero values for each element.
 - Programming practices on page 240 are contradictory but they point to a basic trade off - performance Vs clarity!
 - Note the example, `StudentPollWithTextArea` applet
 - Figure 5.10 - histogram pseudocode
- ```
initialize n[] to { } (histogram values)
set yPosition to 25
```

```
Display "Element," "Value" and "Histogram" at
 25,yPosition; 100,yPosition, and 175,yPosition
 respectively (the columns are at 25, 100, and 175)
```

```
vary i from 0 to the length of n[]
 (each row is displayed)
 increment yPosition by 15 (15 pixels per row)
 display i, n[i] at at 25,yPosition and 100,yPosition
 (the index and the histogram value)
```

```
set xPosition to 175 (the start of the histogram)
```

```
vary j from 1 to n[i]
 display a '*' at xPosition,yPosition
 increment xPositon by 7
```

- Figure 5-11 - an array version of the dice rolling program from figure 4-8

### **References and Reference Parameters**

- In general, there are two ways to pass parameters: call by value and call by reference
  - Call by value makes a copy of the variable in the called method; it does not affect the value of the identifier in the calling method
  - Call by reference passes a pointer, giving the method the ability to directly modify the identifier in the calling method
  - References and pointers are identifiers that store addresses
- Java, unlike other programming languages, does not allow the programmer to choose how to pass arguments
  - Primitive types are passed call by value and objects are passed call by reference
  - The same is true for return identifiers
  - Primitive types are also called scalars
  - An array element can be a scalar or an object and it is treated accordingly
- In Java arrays are treated as objects
  - The name of the array without brackets is actually a reference to the array

- Passing arrays to methods
  - Just the name is given in the calling method
  - The called method must indicate an array in its declaration; it's the name with empty brackets – page 245
  - The size of an array is given in its public `length` attribute, `numArray.length`
- Figure 5.12 passing arrays and primitive data types to methods
  - Line 31: array `a` is passed by reference and each element is doubled
    - The effect is felt in the calling program; the change is known there
  - Line 51 an element is passed and is incremented by 2
    - The effect is not felt in the calling program
  - Line 69: `e` is a local identifier in `modifyElement()`

### Sorting Arrays

- Bubble sort - Figure 5.13, page 248
  - Exhaustive pair-wise comparison; swap elements that are out of order
  - Very “expensive” from the overhead point of view, but easy to understand
  - The array is processed `length - 1` times to ensure it's sorted; worst case is when a small element is at the end and only moves one position per pass.
  - Lines 26-28 do a “swap”
    1. Move the first element to a temporary identifier
    2. Replace the first element with the second element
    3. Replace the second element with the temporary element (which stores the first element)
  - Note the reuse of the `print()` method

### Searching Arrays

- Linear search
  - Assumes that the array is not sorted
  - Each element is compared with a search key
  - The search stops when a match is found or the end of the array is reached
  - Good for small unsorted arrays
  - Figure 5.14 uses an array of the first 100 even numbers
  - The `linearSearch()` returns the index of the key if found, or `-1` if it's not found.

**Special Announcement - Skip the rest of the chapter!!! (Notes are included BUT students are not responsible for this material!)**

- Binary search
  - Assumes that the array is in a sorted order
  - The array is divided in half and the middle element is compared with the key
  - If they are equal, the index is found
  - If they are not equal, the search continues with the top half or the bottom half of the array

- If it's less than the middle element, the first half is searched; otherwise the second half is searched
- A good candidate for recursion!
- This search operates very quickly on large arrays, compared to the linear sort. If the size of the array is  $2^n$ , the maximum number of comparisons is  $n!$ 
  - There is some bookkeeping overhead.

**Figure 5.15, Page 238**

- Pseudo code for binary search
  1. initialize `low` to the low subscript, 0
  2. initialize `high` to the high subscript `length-1`
  3. While the "search range" is not negative, `low <= high`
    - 3.1 set `middle = (low + high)/2` to the middle index
    - 3.2 if there is a match, we're done! `middle` is it.
    - 3.3 otherwise reset `middle` to be `low` or `high`, depending on how `middle` compares, effectively reducing the search area by  $\frac{1}{2}$ . In other words:  
If the key is in `[low, middle]` set `middle` to `high`;  
otherwise set `middle` to `low`, since `low` is in `[middle, high]`
  4. Either a match is found at 3.2, or the search area becomes "negative" and the key is not found
- Notes!
  - The `Graphics` object `g` gets passed to `binarySearch` along with the `searchKey` converted to a number.
  - The first time `paint()` is called, it does nothing due to the boolean `timeToSearch`. It gets set to true before `repaint()`
  - The same `Graphics` object gets passed to `printRow()` via `gg`
- Example – search for 8

| Low, Value | High, Value | Mid, Value | Action                    |
|------------|-------------|------------|---------------------------|
| 0,0        | 14,28       | 7,14       |                           |
| 0,0        | 6,12        | 3,6        | line 81, key <= middle    |
| 4,8        | 6,12        | 5,10       | line 84, key >= middle    |
| 4,8        | 4,8         | 4,8        | line 81, key <= middle    |
|            |             |            | Line 79 match with middle |

**Multiple-Subscripted Arrays**

- Table consisting of indexed rows and columns
- Note initialization on page 256, a set of braces for each row, with surrounding braces.
- Indexing begins at 0 for both the rows and columns
- Figure 5.17

- Note how a doubly dimensioned array is declared in line 20; it's passed by name only in line 17
- In line 27, notice that each row of a double dimensioned array can have it's own size!
- You can think of a double dimensioned array as an array of arrays!
  - `a.length` is the number of rows
  - `a[i].length` is the number of elements in row `i`.
- Figure 5.18
  - Each row is a student and each column is a grade, line 7.
  - The number of rows and the elements per row are calculated in line 16 and 17. Why aren't they just set?

`gradec[]` can be changed without changing lines 16 and 17

- `minimum()` and `maximum()` calculate the largest and smallest grade. Why isn't a reference to `grade` passed to the method? In fact, why isn't it passed to any method?

Because it's an instance of the class and has class scope.

- In line 54, what would be the initialization approach if grades could be higher than 100?

Initialize it to the first value or to `Integer.MAX_VALUE`

- In line 28, note that `paint()` is divided into smaller more manageable pieces with a call to `printArray()` and it passes the `Graphics` identifier `g`.
- In line 45, `grade[i]` is a reference to a single row.
  - `average()` is set up to process a singly dimensioned array. It doesn't care that the reference happens to be a row in a doubly dimensioned array.