Arrays

- So far we have been dealing with single data items
- What if you want to handle multiple related data items of the same type?
- An *array* is a container that holds a related group of values of the same type
  - The grades for this class
  - The average daily temps in Santa Cruz
  - Etc.

Details

- Arrays have a fixed size that specifies how many data values they can hold
- The elements in an array are numbered 0 through *n*-1, where *n* is the size of the array
- Element 0 is the first element in any array
  - This has to do with the way that arrays are stored in memory

Declaring Arrays

- [] indicates that you are declaring an array
- For any type T in java, T[] denotes an array of that type
  - Declaring a variable: int foo;
  - Declaring an array: int[] foo;
- Any type can be made into an array

```
int[] foo;
String[] bar;
char[] list;
double[] temps;
```

Allocating Elements

- After declaring the array, you have to allocate the elements of the array

  ```
  <arrayVariable> = new <type> [ <size> ];
  ```

- You must allocate the elements before using the array
- Once the elements are allocated, the array size is fixed (i.e. it can’t be changed)
  - But you can destroy and allocate a new array with the same name

Examples

```
int[] foo;
foo = new int[10];

double[] bar;
bar = new double[100];

String[] names;
names = new String[116];
```

More Examples

```
int[] foo = new int[10];

double[] temps = new double[365];

String[] names = new String[1000];
```

Indexing an Array Element

- The elements of an array are accessed (indexed) by

  <arrayname>[<index>]

  - Where <index> is less than the size of the array
  - The result is just a variable of the original type

Examples

```
int[] foo = new int [100];
foo[0] = 0;
foo[5] = 73;
int a = foo[99];
foo[17] = foo[12];
System.out.println(“foo[9] = “ + foo[9]);
```

What’ s Really Going On Here

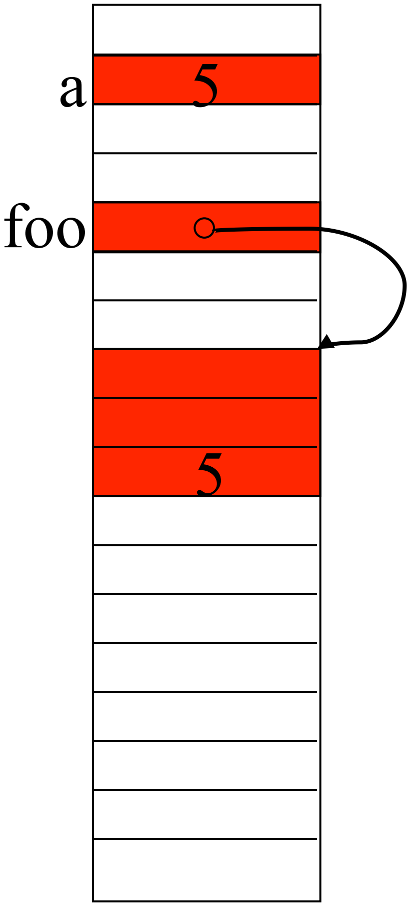
```
int a;

a = 5;

int[] foo;

foo = new int[3]

foo[2] = 5;
```



Array Initialization

- One step at a time

```
int[] a;
a = new int[2];
a[0] = 37;
a[1] = 12;
```

- All at once

```
int[] a = {37, 12};
```

```
//ArraySum.java -sum the elements in an array and
//compute their average
class ArraySum {
    public static void main(String [] args) {
        int[] data = {11,12,13,14,15,16,17};
        int sum = 0;
        double average;
        for (int i = 0; i < 7; i++) {
            sum = sum + data [i];
            System.out.print(data[i] + ", ");
        }
        average = sum / 7.0;
        System.out.println("\n\nsum = “ + sum + “,
    average = “ + average);
    }
}
```

Array Length

- The length of the array is important
- This information is stored with the array
  - Accessed with <arrayname>.length

```
int[] foo = {1,2,3};
for(int i = 0; i < foo.length; i++)
    System.out.println(“foo[i] = “ + foo[i]);
```

Passing Arrays to Methods

- Exactly the same as any other variable

```
int[] foo = {1, 2, 3};

someMethod(foo);

static void someMethod(int[] bar) { ... };
```

Arrays and Methods

- Recall that the array variable and the contents are created separately
- The array name is a reference to the array of values
- When passing an array to a method, the reference is copied into a local variable, but the contents are the same
  - Changing array elements in a method will affect the original values!

```
class SortArray {
    public static void main(String[] args) {
        int[] list = { 17, 3, 24 };

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

        sort(list);

        for(int i = 0; i < list.length; i++)
            System.out.println(list[i]);
    }
static void sort(int[] list) {
    for(int i = 1; i < list.length; i++) {
      if(list[i] < list[i-1]) {
        int temp = list[i-1];
        list[i-1] = list[i];
        list[i] = temp;

        for(int j = i-1; j > 0; j--) {
          if(list[j] < list[j-1]) {
            int temp = list[j-1];
            list[j-1] = list[j];
            list[j] = temp;
} } } } } }
```

Copying Arrays

- What happens if we do this:

```
int[] a, b = {1,2,3};
a = b;
```

- Probably not what we wanted
  - a and b refer to the same physical memory
- Instead:

```
a = (int[])b.clone();
```

What’ s Really Going On Here

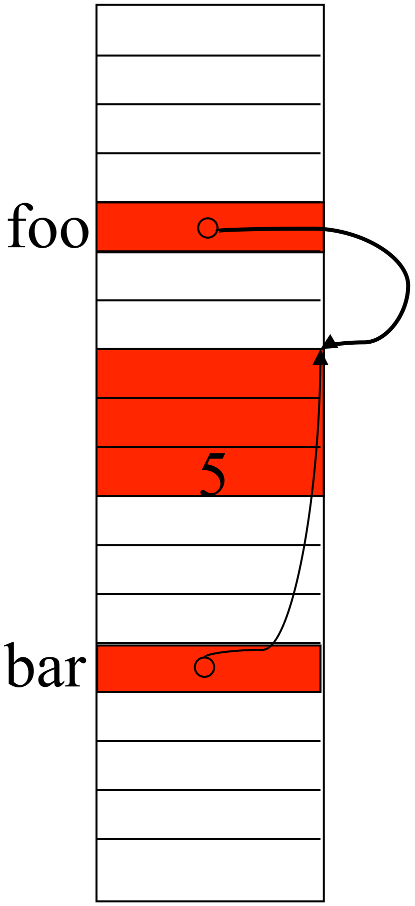
```
int[] foo;

foo = new int[3]

foo[2] = 5;

int[] bar;

bar = foo;
```



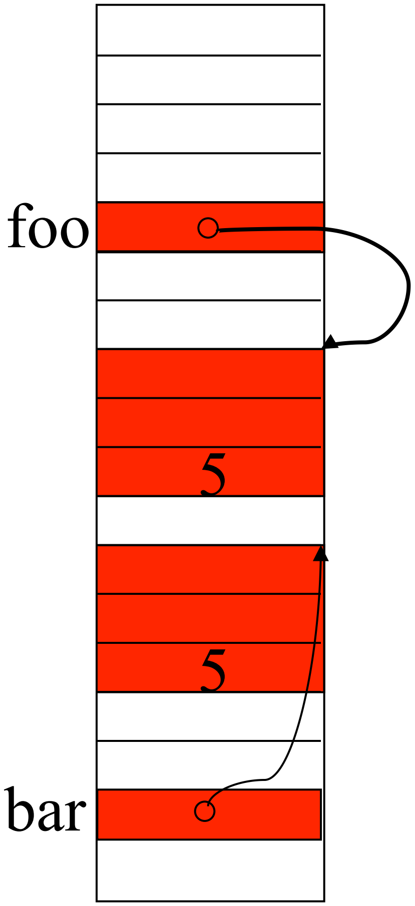
What’ s Really Going On Here

int[] foo;

foo = new int[3]

foo[2] = 5;

int[] bar;



bar = (int[])foo.clone();

Example

- Calculate the min, max, and average of an array of values typed by the user

```
class MMA {
  public static void main(String[] args) {
    double[] foo;
    int size;
    double min, max, sum, avg;
     Scanner in = new Scanner(System.in);

    System.out.println(“Please enter the
 size of the array”);
    size = in.nextInt();
    foo = new double[size];
```

```
      System.out.println(“Enter the elements”);
       for(int i = 0; i < size; i++)
           foo[i] = in.nextDouble();

       min = max = foo[0];
       for(int i = 0; i < foo.length; i++) {
            if(foo[i] < min)
               min = foo[i];
           if(foo[i] > max)
               max = foo[i];
           sum += foo[i];
        }
        avg = sum/foo.length;
    }
}
```

Selection Sort

- Find the smallest element
- Put it at the start of the list
- Find the smallest element in the rest of the list
- Put it in the second spot on the list
- Repeat until the list is sorted

```
//SelectionSort.java -sort an array of integers
import tio.*;
class SelectionSort {
    public static void main(String [ ] args) {
        int [] a = {7,3,66,3,-5,22,-77,2};

        sort(a);

        for (int i =0;i <a.length;i++){
            System.out.println(a [i ]);
        }
    }
//sort using the selection sort algorithm
static void sort(int [] data)) {
    int next,indexOfNext;

    for (next =0;next <data.length -1;next++) {
        indexOfNext = min(data,next,data.length -1);
        swap(data,indexOfNext,next);
    }
}
```

```
static int min(int[] data, int start, int end) {
    int indexOfMin =start;

    for (int i = start+1; i <= end; i++)
        if (data [i] <data [indexOfMin])
            indexOfMin = i;
    return indexOfMin;
}
```

```
  static void swap(int [] data, int first, int second) {
    int temp;
    temp = data [first];
    data [first] = data [second];
    data [second] = temp;
  }
}
```

Searching an Ordered Array

- Data is often stored in large arrays
- Finding a particular element is an important operation
- Faster is better
- If the arrays is unordered, you have to look at every element
- If the array is sorted, you can do better
  - Recall: binary search

Linear Search

```
static int linearSearch(int[] keys, int v) {
    for (int i = 0; i < keys.length; i++) {
        if (keys [i] == v) {
            return i;
        }
    }
    return -1;
}
```

Better Linear Search (sorted list)

```
static int linearSearch(int[] keys, int v){
    for (int i = 0; i < keys.length; i++)
        if (keys[i] == v)
            return i;
        else if (keys[i] > v)
            return -1;
    return -1;
}
```

Binary Search

```
//BinarySearch.java -use bisection search to find
//a selected value in an ordered array
class BinarySearch {
    public static void main(String [ ] args){
        int[] data ={100,110,120,130,140,150};

        int index =binarySearch(data,120);

        System.out.println(index);
    }
    static int binarySearch(int[] keys, int v){
        int position;
        int begin = 0,end = keys.length -1;
        while (begin <= end){
            position = (begin +end)/2;
            if (keys[position] == v)
                return position;
            else if (keys[position ] < v)
                begin = position +1;
            else
                end = position -1;
        }
        return -1;
    }
}
```

Choosing the Best Algorithm

- With *n* data elements:
- Linear search takes *n* steps
- Binary search takes log(*n*) steps
- *n* >> log(*n*)
- Binary search is always faster!
- Aha!

Algorithm Complexity

- In general, it is important to know which algorithms are faster and which are slower
- In particular, we want to know how many operations are required to do a particular algorithm on a given number of data items
- Some algorithms are very efficient, some are doable but slow, and some aren’t doable at all

Examples

n	log(n)	n	2n	2^n
1	0	1	2	2
2	1	2	4	4
3	1.585	3	6	8
4	2	4	8	16
5	2.322	5	10	32
6	2.585	6	12	64
7	2.807	7	14	128
8	3	8	16	256
9	3.17	9	18	512
10	3.322	10	20	1024
100	6.644	100	200	1.26765E+30
1000	9.966	1000	2000	1.0715E+301

Observations

- Notice that
  - The *n* and 2*n* columns grow at the same rate
    - Multiplying by a constant doesn’t make much difference
  - The log(*n*) and n columns grow at very different rates
  - The *n* and 2^n columns also grow at very different rates
    - Different functions of *n* make a big difference

Big O Notation

- Big O notation distills out the important information about how many operations are required for an algorithm
- O(f(*n*)) = c*f(*n*) for any c
    - An O(*n*) takes *on the order of n* operations
- O(log(*n*)) << O(*n*) << O(2^n)
- Putting this into Practice

Putting this into Practice

- Linear Search: O(*n*)
- Binary Search: O(log(*n*))
- Binary search will generally take less time to execute than linear search
- Binary search is a more efficient algorithm

Type and Array

- Recall: You can have an array of any type of object
  - int, double, char, String, boolean
- The details are exactly the same, except that the elements of different types of arrays are of different types

```
//CountWord.java
import tio.*;
public class CountWord {
    public static void main(String[] args) {
        String input;
        char[] buffer;

        System.out.println("type in line");
        input = in.next();

        System.out.println(input);

        buffer = input.toCharArray();

        System.out.println("word count is "+wordCount(buffer));
    }
    // words are separated by nonalphabetic characters
    public static int wordCount(char[] buf)){
        int position =0,wc =0;

        while (position < buf.length) {
            while (position < buf.length && !isAlpha(buf[position ]))
                position++;

            if (position < buf.length)
                wc++;

            while (position < buf.length && isAlpha(buf [position ]))
                position++;
        }
        return wc;
    }
```

```
    public static boolean isAlpha(char c){
        return (c>='a ‘ && c<='z') || (c >='A ‘ && c<='Z');
    }
}
```

Two-Dimensional Arrays

- Recall that data elements of any type can be put in an array
- Arrays of objects can be elements of arrays

```
int[] foo = new int[3];
int[][] bar = new int[3][5];
```

  - bar is an array of 3 arrays of 5 ints
  - bar[0] is an array of 5 ints
  - bar[1] is an array of 5 ints
  - bar[2] is an array of 5 ints

```
// Multiplication table
class Mult {
    public static void main(String[] args) {
        int[][] data = new int[10][10];
        for(int i = 0; i < data.length; i++) {
            for(int j = 0; j < data[i].length; j++) {
                data[i][j] = i * j;
            }
        }
        for(int i = 0; i < data.length; i++) {
            for(int j = 0; j < data[i].length; j++) {
                System.out.print(data[i][j] + “  “);
            }
            System.out.print(“\n”);
}}}
```

Initializing 2D arrays

- Remember that we can provide an initializer list for a 1D array

```
int[] foo = {34, 21, 99, 3};
```

- We can do the same thing for a 2D array

```
int[][] bar = {{3,2,4},{1,2,55},{44,3,9},{4,4,2}};
```

The Game of Life

- This is cool little game, originally developed to simulate certain kinds of growth
- It is "played" on a rectangular (2D) array, like a checker board
- A cell of the board is either alive or dead
- Alive cells are marked with an *

Rules of Life

1. A cell is either empty, indicated by a blank, or alive, indicated by an * .
2. Each cell is thought of as the center of a 3×3 square grid of cells which contains its eight neighbors.
3. A cell that is empty at time *t* becomes alive at time *t* +1 if and only if exactly three neighboring cells were alive at time *t.*
4. A cell that is alive at time *t* remains alive at time *t* +1 if and only if either two or three neighboring cells were alive at time *t*. Otherwise,it dies for lack of company (<2) or overcrowding (>3).
5. The simulation is conducted, in principle, on an infinite two-dimensional grid.

Analysis

- We will simulate a finite grid.
- Because the grid isn’t infinite, we must decide how to deal with the borders.
  - To keep the program simple,we will treat the borders as lifeless zones.
- The initial placement of life forms in the grid will be read from the keyboard as a sequence of * s and dots.
  - The *s will stand for life forms and the dots will represent empty cells.

Analysis

- The user will specify the size of the grid, which will always be square.
- The user will specify the number of generations to simulate.
- The program should echo the initial generation and then print each new generation simulated.

Algorithm

1. Read in the size of the grid
2. Read in the initial generation
3. Read in the number of generations to simulate
4. Print the current generation
5. For the specified number of generations
    1. advance one generation
    2. print the generation

Algorithm for Advancing

1. For each cell in the grid
    1. Compute the number of neighbors
    2. If the cell has 2 neighbors and was alive it stays alive
    3. If the cell has 3 neighbors, it is alive
    4. Otherwise the cell is dead

Methods

- int getSize();
- void getInitialGrid(grid);
- int getGenerations();
- void updateGrid(boolean[][] grid);
- boolean updateCell(boolean[][] grid, int i, int j);
- int countNeighbors(boolean[][] grid, int i, int j);
- void printGrid(boolean[][] grid);

Data

- boolean[][] grid;
- int size;
- int generations;
- boolean[][] oldgrid;

Arrays of Non-Primitive Types

- Recall: Any type can be made into an array
- This is also true for non-primitive types like String

```
String[] foo;
foo = new String[2];
foo[0] = “Hi”;
foo[1] = “Scott”;
```

Arrays of Strings

```
class StringArray {
    public static void main(String[] args) {
        String[] myStringArray = {"zero","one","two",
        "three","four","five","six","seven",
        "eight","nine"};

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

main(String[] args)

- Now we know what String[] args means
- It is an array of *command-line arguments* – the parameters passed to the program on the command line
- java myProgram one two three
  - Passes “one”, “two”, and “three” in args

Command Line Arguments

```
//CommandLine.java -print command line arguments
class CommandLine {
    public static void main(String[] args){
        for (int i = 0; i < args.length; i++)
            System.out.println(args [i]);
    }
}
```