CMSC 202H

ArrayList, Multidimensional Arrays
What’s an Array List

- **ArrayList** is
  - a class in the standard Java libraries that can hold any type of object
  - an object that can grow and shrink while your program is running (unlike arrays, which have a fixed length once they have been created)

- In general, an **ArrayList** serves the same purpose as an array, except that an **ArrayList** can change length while the program is running
The \textbf{ArrayList} Class

- The class \texttt{ArrayList} is implemented using an array as a private instance variable
  - When this hidden array is full, a new larger hidden array is created and the data is transferred to this new array
Using the **ArrayList** Class

- In order to make use of the **ArrayList** class, it must first be imported

  ```java
  import java.util.ArrayList;
  ```

- An **ArrayList** is created and named in the same way as object of any class:

  ```java
  ArrayList aList = new ArrayList();
  ```

(Note that what we are teaching here is an obsolete, simplified form of ArrayList you can use *for now*; for documentation, see: [http://download.oracle.com/javase/1.4.2/docs/api/java/util/ArrayList.html](http://download.oracle.com/javase/1.4.2/docs/api/java/util/ArrayList.html). Later, we will learn the proper form, after covering Generics.)
Adding elements to an **ArrayList**

- The **add** method is used to add an element at the “end” of an **ArrayList**
  ```java
  list.add("something");
  ```
  - The method name **add** is overloaded
  - There is also a two argument version that allows an item to be added at any currently used index position or at the first unused position
How many elements?

- The **size** method is used to find out how many indices already have elements in the **ArrayList**
  
  ```java
  int howMany = list.size();
  ```

- The **set** method is used to replace any existing element, and the **get** method is used to access the value of any existing element
  
  ```java
  list.set(index, "something else");
  String thing = (String) list.get(index);
  ```

  Note that the returned value must be cast to the proper type

- **size** is NOT capacity
  
  - size is the number of elements currently stored in the **ArrayList**
  - Capacity is the maximum number of elements which can be stored. Capacity will automatically increase as needed
ArrayList code Example

public static void main( String[ ] args)
{
    ArrayList myInts = new ArrayList();
    System.out.println( “Size of myInts = “ + myInts.size());
    for (int k = 0; k < 10; k++)
        myInts.add( 3 * k );
    myInts.set( 6, 44 );
    System.out.println( “Size of myInts = “ + myInts.size());
    for (int k = 0; k < myInts.size(); k++)
        System.out.print( myInts.get( k ) + “, “ );
}
// output
Size of myInts = 0
Size of myInts = 10
0, 3, 6, 9, 12, 15, 44, 21, 24, 27
Methods in the Class `ArrayList`

- The tools for manipulating arrays consist only of the square brackets and the instance variable `length`.
- `ArrayLists`, however, come with a selection of powerful methods that can do many of the things for which code would have to be written in order to do them using arrays.
ArrayList Constructors

- **Constructors:**
  - `ArrayList()`  
    - Constructs an empty list with an initial capacity of ten.
  - `ArrayList(int initialCapacity)`  
    - Constructs an empty list with the specified initial capacity.

(Constructor and method descriptions borrowed from Sun javadoc pages)
ArrayList Methods

- **Method Summary** (incomplete)
  - void `add`(int index, `Object` element)
    - Inserts the specified element at the specified position in this list.
  - boolean `add`(Object o)
    - Appends the specified element to the end of this list.
  - int `size()`
    - Returns the number of elements in this list.
ArrayList Methods (cont)

- **Object set**(int index, **Object** element)
  - Replaces the element at the specified position in this list with the specified element.

- **Object get**(int index)
  - Returns the element at the specified position in this list.

- **Object remove**(int index)
  - Removes the element at the specified position in this list. protected

- **void removeRange**(int fromIndex, int toIndex)
  - Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
ArrayList Methods (cont)

- **void clear()**
  - Removes all of the elements from this list.

- **Object clone()**
  - Returns a shallow copy of this ArrayList instance.

- **int indexOf(Object elem)**
  - Searches for the first occurrence of the given argument, testing for equality using the equals method.

- **int lastIndexOf(Object elem)**
  - Returns the index of the last occurrence of the specified object in this list.
The "For Each" Loop

- The `ArrayList` class is an example of a collection class
- Starting with version 5.0, Java has added a new kind of for loop called a *for-each* or *enhanced for* loop
  - This kind of loop has been designed to cycle through all the elements in a collection (like an `ArrayList`
“for-each” example

```java
public class ForEach {
    public static void main(String[] args) {
        ArrayList list = new ArrayList();
        list.add(new Date(1, 1, 1000));
        list.add(new Date(7, 4, 1776));
        list.add(new Date(9, 1, 2011));

        // “for each object, i, in list"
        for( Object i : list )
            System.out.println( i );
    }
}

//--- Output ---
1/1/1000
7/4/1776
9/1/2011
```
Copying an ArrayList

// create an ArrayList of Dates (assume we have some around)
ArrayList a = new ArrayList();
a.add(d1); a.add(d2); a.add(d3);

Assignment doesn’t work

- As we’ve seen with any object, using assignment just makes two variables refer to the same ArrayList.

ArrayList b = a;
Copying an ArrayList

ArrayList’s clone( ) method makes a shallow copy

ArrayList b = a.clone();
Copying an ArrayList

We need to manually make a deep copy

```java
ArrayList b = a.clone();
```
Copying an ArrayList

We need to manually make a deep copy

```java
ArrayList b = a.clone();
for (int k = 0; k < b.size(); k++) {
    Date origDate = (Date) b.get(k);
    b.set(k, new Date(origDate));
}
```
ArrayList vs Array

Why use an array instead of an ArrayList

1. An ArrayList is less efficient than an array

2. ArrayList does not have the convenient square bracket notation

3. The elements of an ArrayList must be a class type (or other reference type). It cannot be a primitive type. (Although wrappers, auto boxing, and auto unboxing make this less of an issue with Java 5)
ArrayList vs Array

Why use an ArrayList instead of an array?

1. Arrays can’t grow. Their size is fixed at compile time.
   - ArrayList grows and shrinks as needed while your program is running
2. You need to keep track of the actual number of elements in your array (recall partially filled arrays).
   - ArrayList will do that for you.
3. Arrays have no methods (just length instance variable)
   - ArrayList has powerful methods for manipulating the objects within it
Some Warnings

- This lecture describes an obsolete form of ArrayList
  - The original form of ArrayList stored Object elements, so you had to constantly do casts
  - The addition of generics to the language completely changed the use of collections like ArrayLists
  - To keep a modicum of backwards compatibility, raw types allow ArrayLists to be used as originally designed
  - Important: just because you can mix types together does not mean you should!
The **Vector** Class

- The Java standard libraries have a class named *Vector* that behaves almost exactly the same as the class *ArrayList*.
- In most situations, either class could be used, however the *ArrayList* class is newer (Java 5), and is becoming the preferred class.
Multidimensional Arrays
Multidimensional Arrays

- Review of 1-dimensional arrays:
  - To declare and initialize:
    ```java
    int[] myArray = new int[4];
    ```
  - To access:
    ```java
    ```
  - To use as an object:
    ```java
    for (i = 0; I < myArray.length; i++) {
    ```
  - To demonstrate that it’s a reference variable:
    ```java
    myArray = null;
    // Now, “myArray[3]” would cause an error
    ```
Multidimensional Arrays

- Extending to 2-dimensional arrays:

- To declare and initialize:
  ```java
  int[][] myArray = new int[3][4];
  // How would you declare 2-dim arrays in C?
  ```

- To access:
  ```java
  myArray[1][3] = myArray[1][3] + 1;
  ```

- To use as an object:
  ```java
  numRows = myArray.length;
  // Following assumes rectangular matrix
  numCols = myArray[0].length;
  ```
Multidimensional Arrays

- But in Java, a 2D array is actually a reference-to-an-array-of-references:

  // Can do:
  myArray[1] = null;
  myArray[1][3] = 47; // This will cause error
  // but myArray[0][3] still okay

  // Can also make it a “ragged” array:
  myArray[1] = new int[20];

  // What do you think following does?
  myArray = new int[10][];

  // ...and what would this do?
  myArray = new int[40];
Multidimensional Arrays

- Luckily, if you don’t want to get fancy, you can pretend that it’s simply a 2-D array
- Even if you do create complex, dynamically allocated, ragged arrays, you don’t have to worry about memory management