Mat 2170

The ArrayList Class

Spring 2014

Student Responsibilities

- Reminder: **EXAM** next Thursday, 4/24, at 7:00 pm
  Picnic is Tuesday, 4/22 - get signed up.
- Reading: Textbook, Chapter 11.8, The `ArrayList` class
- Lab 13, utilizing the `ArrayList` class
- Attendance

Wrapper Classes

- Java designers chose to separate primitive types from the standard class hierarchy, mostly for **efficiency** reasons.
- Primitive Java values take less space and allow Java to use more of the capabilities provided by hardware.
- However, there are times when the fact that **primitive types aren't objects** poses problems

  (e.g., there are tools in the Java libraries that work **only** with objects and not primitive types — one such is `ArrayList`).

- To get around this problem, Java includes a wrapper class to correspond to each of the following primitive types:

`boolean`	↔	`Boolean`	`float`	↔	`Float`
`byte`	↔	`Byte`	`int`	↔	`Integer`
`char`	↔	`Character`	`long`	↔	`Long`
`double`	↔	`Double`	`short`	↔	`Short`

- All of the above primitive wrapper classes in Java are **immutable** – their states (contents) cannot be modified after they are created, only replaced.

- The value stored in an instance of any of the wrapper classes **is an object**, and we can use it in any context that requires an object.

Boxing and Unboxing

- Java SE 5.0 (and subsequent versions) **automatically converts values** back and forth between a primitive type and the corresponding wrapper class. These operations are called **boxing** and **unboxing**.

- For example,

  ```
  Integer maxItems = 5;
  ```

  causes Java to call the `Integer` constructor, and is equivalent to:

  ```
  Integer maxItems = new Integer(5);
  ```

- Similarly,

  ```
  int nextMax = maxItems + 1;
  ```

  is equivalent to:

  ```
  int nextMax = maxItems.intValue()+1;
  ```

Storing Large Amounts of Data

- It is often the case that in order to solve a problem by computer, we need to be able to store an unknown and / or a large number of data items.

- It would be difficult to create individual names and storage locations for each.

- Therefore, programming languages such as Java offer ways to store collections of data in various **containers**.

Introduction: Arrays

An array is a **collection of individual data values** with two distinguishing characteristics:

1. An **array** is **ordered**
   We must be able to count off the values — here is the first, here is the second, and so on — just like `Strings`.

2. An array is **homogeneous**
   Every value in the array must be of the **same type**.

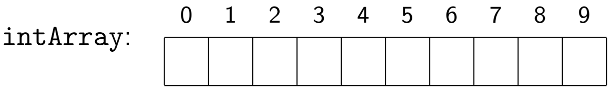
Arrays are a primitive type in Java.
They do not have methods associated with them.

Array Terminology

- An array is a java **container** object that holds a fixed number of values of a **single type**.
- The length of an **array** is established when the array is created. After creation, its length is fixed.
- The individual values in an array are called **elements**.
- The type of object an array can hold is its **element type**.
- Each element is identified by its **position** in the array
  — also called its **index** —
  which always begins at **0** and ends at **length - 1**
  (Just like `String` objects.)

Array Data Storage

The easiest way to visualize arrays is to think of them as a **linear collection** of boxes, much like a row of Post Office boxes, each of which is marked with its index number.
For example:



`intArray`:

0 1 2 3 4 5 6 7 8 9

where `intArray` was declared as an array of `int` of size 10.

The `ArrayList` Class

- The **java.util** package includes a **class** called **ArrayList** that extends the usefulness of arrays by providing additional operations and ease of use.

- The ArrayList class is a **wrapper** for the primitive `array` type — it **encapsulates** an array and provides methods for accessing and interacting with it.

- The ArrayList class **hides the details** of array manipulation.

- All operations on an ArrayList object (and hence, the array within) are accomplished using **method calls**.

Generic Types and Boxing/Unboxing

- **The element type** of any **ArrayList must be a Java class**.

- Automatic conversion of values between a primitive type and the corresponding wrapper class allows an `ArrayList` object to store **primitive values** by using their **wrapper classes**.

- For example, to create a list of integers:

```
ArrayList <Integer> myList = new ArrayList<Integer>();
```

  This statement invokes a constructor to create an ArrayList of `Integer`.

Accessing the Inner int

- Then we may store and access `int` values in `myList` through automatic boxing and unboxing:

- In this statement, Java uses **boxing** to enclose 42 in a wrapper object of type `Integer`:

```
myList.add(42);
```

- Here, the statement **unboxes** the `Integer` to obtain the `int` equivalent:

```
int answer = myList.get(0);
```

Back to the ArrayList Class

- A new **ArrayList** object is created by calling the **ArrayList** **constructor**, for example:

```
ArrayList<String> myNames = new ArrayList<String>();
```

- It is a really good idea to specify the element type, such as <String> in the example above, in **angle brackets** when invoking the constructor.

- Doing this allows Java to check for the correct element type when set() is called, and eliminates the need for a type cast when get() is called.

Generic Types in Java

- In the summary of **ArrayList** methods which follows, the notation $<$ **T** $>$ indicates the **base** or **element** type of the `ArrayList` object.

- In other words, the **type parameter** $< T >$ is a placeholder for the **element type** used in the array.

- Class definitions that include a **type parameter** are called **generic types**.

`ArrayList` Methods

`boolean add(<T> element)`

Adds a new element to the end of the `ArrayList`; the return value is always `true`

`void add(int index, <T> element)`

Inserts a new element into the `ArrayList`; **before** the position specified by `index`

`<T> remove(int index)`

Removes the element at the specified position and returns that value

`boolean remove(<T> element)`

Removes the first instance of `element`, if it appears; returns `true` if a match is found `void clear()`

Removes all elements from the `ArrayList`

`int size()`

Returns the number of elements in the `ArrayList`

`<T> get(int index)`

Returns the object at the specified index

`<T> set(int index, <T> value)`

Sets the element at the specified index to the new value and returns the old value `indexOf(<T> value)`

Returns the index of the first occurrence of the specified value, or -1 if it does not appear

`boolean contains(<T> value)`

Returns `true` if the `ArrayList` contains the specified value

`boolean isEmpty()`

Returns `true` if the `ArrayList` contains no elements

Cycling through ArrayList Elements

- One of the most useful things about element selection in an ArrayList is that the index does not have to be a constant — in many cases we use the control object of a `for` loop.

- The standard `for` loop pattern that cycles through each of the ArrayList elements:

```
for (int i = 0; i < myList.size(); i++) {
     Operations involving the $i^{th}$ ArrayList element}
```

- As an example, we can reset every element in `intList` to twenty-nine using the following:

```
for (int i = 0; i < intList.size(); i++) {
     intList.set(i, 29);}
```

Human–Readable Index Values

The fact that Java starts index numbering at **zero** can be confusing and should be **hidden** from users.

There are two standard approaches for shifting between Java and human–readable index numbers:

1. **Use Java's index numbers** internally, but **add one** whenever those numbers are presented to the user.
2. **Use index values beginning at 1** and **ignore element 0** in each array.

   This requires allocating an additional element for each array, but has the advantage that the internal and external index numbers correspond.

The Auto–increment Operator, ++

- A program uses the auto–increment operator in various statements in the following form:

```
pegs.set(pegIndex++, new GPoint(x, y));
```

- The **pegIndex++** expression adds one to `pegIndex` just as it has all along. The question is: **what value is used as the index**? It depend on the location of the ++.
  - **Object++** : the object is incremented **after** the value of the expression is determined.
  - **++Object** : the object is incremented **first**, then the new value is used in context.

- The auto–decrement operator (−−) behaves similarly.

Creating an indexed ArrayList

```
import acm.program.*;
import java.util.*;

public class FillArrayList extends ConsoleProgram {
   public void run()
   {
     println("This program fills an ArrayList with " +
             "values matching the indices.");

     int listLength = readInt("How large would you like\n"
                  + " your list?  Enter length: ");

     // method returns a filled list
     ArrayList<Integer> myList = indexIntArrayList(listLength);

     printIntArrayList(myList);
   }
```

Filling an Indexed ArrayList

```
private ArrayList<Integer> indexIntArrayList(int n) {

  // create an empty list of Integers
  ArrayList<Integer> theList = new ArrayList<Integer>();

  // Fill the list with values that match the indices
  for ( int cnt = 0; cnt < n; cnt++){
     theList.add(cnt);
  }

  // return the indexed list
  return theList;
}
```

Printing an ArrayList of Integers

```
private void printIntArrayList(ArrayList<Integer> theList) {

  // Display each value in the list, separated by commas and
  // surrounded by brackets, with 15 per line

  print("[");
  for ( int cnt = 0; cnt < theList.size(); cnt++){
     print(theList.get(cnt));
     if (cnt != theList.size()-1) {
         print(", ");
         if ((cnt % 15) == 0)
             println();
      }
  }
  println("]");
}
```

Passing ArrayList Objects as Parameters

- When an ArrayList is passed as a parameter to a method or is returned by a method as a result, only the **reference** to the object is actually passed between the methods.
- Since the reference, or address, of the array is passed in, the elements of an array are effectively **shared** between the caller and callee.
- If a method changes an element of an array passed as a parameter, that change will **persist** after the method returns.

Reversing an ArrayList

```
import acm.program.*;
import java.util.*;

public class ReverseArrayList extends ConsoleProgram {
   public void run()
   {
      println("This program reverses the elements " +
              "in an ArrayList.");
      println("Use " + SENTINEL + " to signal the " +
              "end of the list.");

      ArrayList<Integer> myList = readIntArrayList();
      reverseArrayList(myList);
      printIntArrayList(myList);
   }
```

readIntArrayList()

```
/* Reads the data from the user into the list */

private ArrayList<Integer> readIntArrayList()
{
   ArrayList<Integer> list = new ArrayList<Integer>();

   int value = readInt(" ? ");
   while (value != SENTINEL)
   {
      list.add(value);
      value = readInt(" ? ");
   }

   return list;
}

/* Private constant --- Define the end-of-data value */
private static final int SENTINEL = 0;
```

reverseArrayList() & swapElements()

```
/* Reverses the data in an ArrayList */
private void reverseArrayList(ArrayList<Integer> list)
{
   for (int i = 0; i < list.size() / 2; i++)
   {
      swapElements(list, i, list.size() - i - 1);
   }
}


/* Exchanges two elements in an ArrayList */
private void swapElements(ArrayList<Integer> list,
                          int p1, int p2)
{
   int temp = list.get(p1);
   list.set(p1, list.get(p2));
   list.set(p2, temp);
}
```

Lab 13

- There are two projects assigned in Lab 13:

  1. `ArrayListStats` : finding max, min, average, and standard deviation of a list of integer values entered by the user

  2. `TemperatureStats` : which does the same for a list of random `Temperature` objects.

- The standard deviation of a list of values is given by:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (\mu - x_i)^2}{n}}$$

where:

- the x_i are the list elements
- n is the length of the list
- $\sum_{i=1}^{n} (\mu - x_i)^2$ is the sum of the squares of the average minus each list element

Lab 13 Notes

- Pay close attention to the instructions in the lab write–up.
- When the user is asked if they wish to continue, your program must require that they enter a 'y', 'Y', 'n', or 'N', using one or more methods.
- If the user answers 'y' or 'Y', repeat execution, otherwise display a final message indicating end of the program has been reached.

Using Arrays for Tabulation

- Arrays are very useful when we have a set of data values and need to **count** how many of them fall into each of a given, finite set of ranges.

  This process is called **tabulation**.

- Tabulation uses arrays in a slightly different way from those applications that use them to simply store a list of data.

Tabulation

- When a tabulation program is implemented, each data value is used to **compute an index** into an integer array that counts how many values fall into that category.

- The example of tabulation used in the text is a program that counts how many times each of the 26 letters of the English alphabet appears in a sequence of text lines.

- Such a program would be useful in solving codes and ciphers.

Cryptograms

- A **cryptogram** is a **puzzle** in which a message is **encoded** by replacing each letter in the original text with some other letter — with the substitution pattern remaining the same throughout the message.

- The usual **strategy** for solving a cryptogram:

> Assume that the most common letters in the coded message correspond to the most common letters in English.

The most common letters: E, T, A, O, I, N, S, H, R, D, L, U

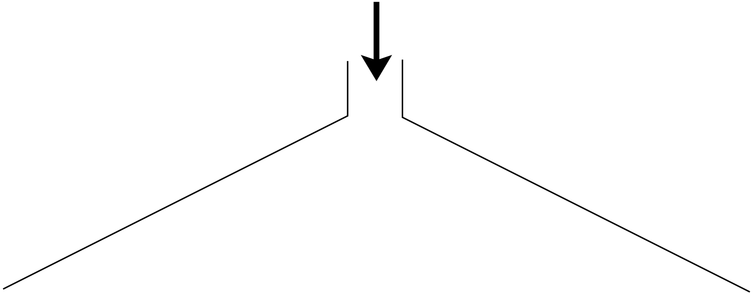
(which won't be a surprise if you've seen Wheel of Fortune)

Implementation Strategy

- Instead of counting each of the characters by hand, it would be much easier to have a program to do the job — type in a coded message, and out pops a table showing how often each letter appears. . .
- Basic Idea: count letter frequencies by using an array with 26 elements to count the number of times each letter appears.
- As the program processes the text, it increments the array element that corresponds to each letter.

Implementation Concept

TWAS BRILLIG



1	1	0	0	0	0	1	0	2	0	0	2	0	0	0	0	0	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Create a table of letter frequencies

```
public class LetterFrequencies extends ConsoleProgram {
  public void run()
  {
    println ("This program counts letter frequencies.");
    println ("Empty line indicates end of input.");
    initFrequencyTable();
    String line = readLine("Enter text to scan: ");
    while(line.length() > 0)
    {
      countLetterFrequencies(line);
      line = readLine("Enter next line, blank to exit: ");
    }
    printFrequencyTable();
  }

  // private global data member:
  private ArrayList<Integer> frequencyTable;
```

```
// Initialize list of 26 counters, setting each to zero

  private void initFrequencyTable()
  {
    frequencyTable = new ArrayList<Integer>();
    for(int i = 0; i < 26; i++)
      frequencyTable.add(0);
  }
```

```
// Count the letter frequencies in a line of text
// basing the table index on each character’s place in
// the alphabet, and incrementing the associated cell.

private void countLetterFrequencies(String line)
{
  for (int i = 0; i < line.length(); i++)
  {
    char ch = line.charAt(i);
    if (Character.isLetter(ch))
    {
      int index = Character.toUpperCase(ch) - ’A’;
      frequencyTable.set(index, frequencyTable.get(index)+1);
    }
  }
}
```

```
// Display frequency table - using characters
//   to step through the indices

private void printFrequencyTable()
{
  for (char ch = ’A’; ch <= ’Z’; ch++)
  {
    int index = ch - ’A’;
    println(ch + ": " + frequencyTable.get(index));
  }
}
```

Constant Lookup Tables

- One of the most common applications of array initialization is to create constant arrays, called **lookup tables**, which are used to **look up a value by its index number**.

- Suppose we use the integers 1 through 12 to represent the names of the months from January to December.

Constant Lookup Tables

- Easily convert these integers to the corresponding month name by declaring and initializing the table:

```
ArrayList<String> MonthNames = new ArrayList<String>();

Collections.addAll(MonthNames, "Null", "January", "February",
     "March", "April", "May", "June", "July", "August",
     "September", "October","November", "December");
```

- The expression **MonthNames.get(monthNumber)** can then be used to convert a numeric month to its name, as long as you ensure that `month` lies in the correct range.

Looking Up the Month

```
println("This program will give you the name of the month");
println(" given its number.  For example, month 1 is January");

int which = readInt("Give me a month number " +
                    " [1..12], -1 to quit: ");
while (which != -1) {
  if (1 <= which && which <= 12) {
      println("Your month for " + which +
              " is: " + MonthNames.get(which));
   }
  else println("That value is not in range.");
  which = readInt("Give me a month number, -1 to quit: ");
}

println("Thanks for using my program!  Bye.");
```

Exercises. Write methods to:

1. Sum an `ArrayList`, `bob`, of integers
2. Find the partial sums of an `ArrayList`, `myList`, of integers
3. Produce a copy of an `ArrayList`, `dList`, of floating point values with all non–positive numbers deleted
4. Find the product of an `ArrayList`, `fracList`, of `Rational` objects