

# CSC 221: Computer Programming I

Fall 2005

## Lists, data access, and searching

- ArrayList class
- ArrayList methods: add, get, size, remove, contains, set, indexOf, toString
- example: Notebook class
- files and lists
- parallel lists
- autoboxing/unboxing
- examples: word counts, word frequencies, ComparePlans revisited

1

## Composite data types

### String is a composite data type

- each String object represents a collection of characters in sequence
- can access the individual components & also act upon the collection as a whole

many applications require a more general composite data type, e.g.,

- ✓ a to-do list will keep track of a sequence/collection of notes
- ✓ a dictionary will keep track of a sequence/collection of words
- ✓ a payroll system will keep track of a sequence/collection of employee records

Java provides several library classes for storing/accessing collections of arbitrary items

2

## ArrayList class

an `ArrayList` is a generic collection of objects, accessible via an index

- must specify the type of object to be stored in the list
- create an `ArrayList<?>` by calling the `ArrayList<?>` constructor (no inputs)

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

- add items to the end of the `ArrayList` using `add`

```
words.add("Billy");           // adds "Billy" to end of list
words.add("Bluejay");        // adds "Bluejay" to end of list
```

- can access items in the `ArrayList` using `get`
  - similar to `Strings`, indices start at 0

```
String first = words.get(0);  // assigns "Billy"
String second = words.get(1); // assigns "Bluejay"
```

- can determine the number of items in the `ArrayList` using `size`

```
int count = words.size();     // assigns 2
```

3

## Simple example

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

words.add("Nebraska");
words.add("Iowa");
words.add("Kansas");
words.add("Missouri");

for (int i = 0; i < words.size(); i++) {
    String entry = words.get(i);
    System.out.println(entry);
}
```

since an `ArrayList` is a composite object, we can envision its representation as a sequence of indexed memory cells

"Nebraska"	"Iowa"	"Kansas"	"Missouri"
0	1	2	3

exercise:

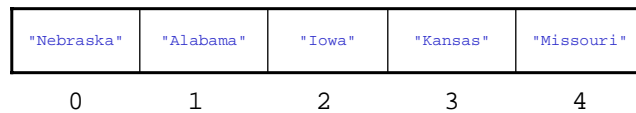
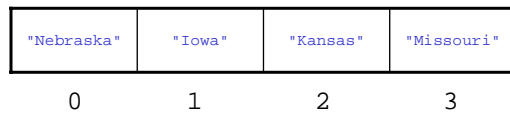
- given an `ArrayList` of state names, output index where "Hawaii" is stored

4

## Other ArrayList methods

the generic `add` method adds a new item at the end of the `ArrayList`  
a 2-parameter version exists for adding at a specific index

```
words.add(1, "Alabama") // adds "Alabama" at index 1, shifting  
                        // all existing items to make room
```



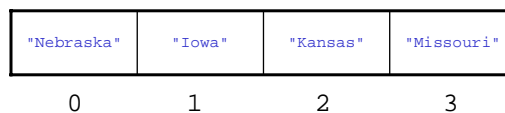
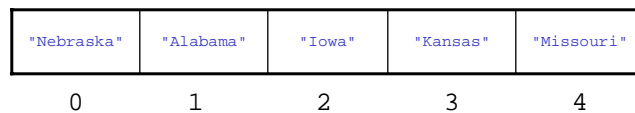
5

## Other ArrayList methods

in addition, you can remove an item using the `remove` method

- specify the item to be removed by index
- all items to the right of the removed item are shifted to the left

```
words.remove(1);
```



6

## Notebook class

consider designing a class to model a notebook (i.e., a to-do list)

- will store notes in an `ArrayList<String>`
- will provide methods for adding notes, viewing the list, and removing notes

```
import java.util.ArrayList;

public class Notebook
{
    private ArrayList<String> notes;

    public Notebook() { ... }

    public void storeNote(String newNote) { ... }
    public void storeNote(int priority, String newNote) { ... }

    public int numberOfNotes() { ... }

    public void listNotes() { ... }

    public void removeNote(int noteNumber) { ... }
    public void removeNote(String note) { ... }
}
```

any class that uses an `ArrayList` must load the library file that defines it

7

```
...
/**
 * Constructs an empty notebook.
 */
public Notebook()
{
    notes = new ArrayList<String>();
}

/**
 * Store a new note into the notebook.
 * @param newNote note to be added to the notebook list
 */
public void storeNote(String newNote)
{
    notes.add(newNote);
}

/**
 * Store a new note into the notebook with the specified priority.
 * @param priority 1 <= priority <= numberOfNotes()
 * @param newNote note to be added to the notebook list
 */
public void storeNote(int priority, String newNote)
{
    notes.add(priority-1, newNote);
}

/**
 * @return the number of notes currently in the notebook
 */
public int numberOfNotes()
{
    return notes.size();
}
...
```

constructor creates the (empty) `ArrayList`

one version of `storeNote` adds a new note at the end

another version adds the note at a specified index

`numberOfNotes` calls the `size` method

8

## Notebook class (cont.)

```
...
/**
 * Show a note.
 * @param notePriority the number of the note to be shown (first note is #1)
 */
public void showNote(int notePriority)
{
    if (notePriority <= 0 || notePriority > numberOfNotes()) {
        System.out.println("There is no note with that priority.");
    }
    else {
        String entry = notes.get(notePriority-1);
        System.out.println(entry);
    }
}

/**
 * List all notes in the notebook.
 */
public void listNotes()
{
    System.out.println("NOTEBOOK CONTENTS");
    System.out.println("-----");
    for (int i = 1; i <= numberOfNotes(); i++) {
        System.out.print(i + " ");
        showNote(i);
    }
}
...
```

showNote checks to make sure the note number is valid, then calls the get method to access the entry

listNotes traverses the ArrayList and shows each note (along with its #)

9

## Notebook class (cont.)

```
...
/**
 * Removes a note.
 * @param notePriority the number of the note to be removed (first is #1)
 */
public void removeNote(int notePriority)
{
    if (notePriority <= 0 || notePriority > numberOfNotes()) {
        System.out.println("There is no note with that priority.");
    }
    else {
        notes.remove(notePriority-1);
    }
}

/**
 * Removes a note.
 * @param note the note to be removed
 */
public void removeNote(String note)
{
    boolean found = false;
    for (int i = 0; i < notes.size(); i++) {
        String entry = notes.get(i);
        if (entry.equals(note)) {
            notes.remove(i);
            found = true;
        }
    }

    if (!found) {
        System.out.println("There is no such note.");
    }
}
}
```

one version of removeNote takes a note #, calls the remove method to remove the note with that number

another version takes the text of the note and traverses the ArrayList – when a match is found, it is removed

uses boolean variable to flag whether found or not

10

## In-class exercises

download [Notebook.java](#) and try it out

- add notes at end
- add notes at beginning and/or middle
- remove notes by index
- remove notes by text

add a method that allows for assigning a random job from the notebook

- i.e., pick a random note, remove it from notebook, and return the note

```
/**
 * @return a random note from the Notebook (which is subsequently removed)
 */
public String handleRandom()
{
}
}
```

11

## Another example: File stats

most word processors (e.g., Word, WordPerfect) can provide stats on a file

- e.g., number of words, number of characters, ...

suppose we wanted to alter/extend the Document class to keep track of

- the total number of words in the file
- the number of unique words in the file

basic algorithm: keep count of total words, an `ArrayList` of unique words

```
while (STRINGS REMAIN TO BE READ) {
    nextWord = NEXT WORD IN FILE;

    totalWordCount++;

    if (NOT ALREADY STORED IN LIST) {
        uniqueWords.add(word);
    }
}
```

note: we could utilize a `Word` class to represent individual words (as in HW5) or just store & manipulate as `Strings`

12

## Storing unique values

to keep track of unique words, need to ignore words that are already stored

- we could write our own method to search through the `ArrayList`

```
private boolean contains(ArrayList<String> words, String desired)
{
    for (int i = 0; i < words.size(); i++) {
        String entry = words.get(i);
        if (entry.equals(desired)) {
            return true;
        }
    }
    return false;
}

-----

if (!contains(uniqueWords, word)) {
    uniqueWords.add(word);
}
```

fortunately, the `ArrayList` class already has such a method

```
if (!uniqueWords.contains(word)) {
    uniqueWords.add(word);
}
```

13

## DocumentCount class

```
import java.util.Scanner;
import java.io.File;
import java.util.ArrayList;

public class DocumentCount
{
    private int numWords;
    private ArrayList<String> uniqueWords;

    public DocumentCount(String filename) throws java.io.FileNotFoundException
    {
        numWords = 0;
        uniqueWords = new ArrayList<String>();

        Scanner infile = new Scanner(new File(filename));
        while (infile.hasNext()) {
            String nextWord = infile.next();
            nextWord = nextWord.toLowerCase();
            nextWord = StringUtils.stripNonLetters(nextWord);

            numWords++;

            if (!uniqueWords.contains(nextWord)) {
                uniqueWords.add(nextWord);
            }
        }
    }

    public int getNumWords()
    {
        return numWords;
    }

    public int getNumUniqueWords()
    {
        return uniqueWords.size();
    }
}
```

as before, a counter field is used to keep track of the total number of words

an `ArrayList` is used to store unique words  
the number of unique words can be extracted using the `size` method

what if we wanted:

```
/**
 * Displays all unique words
 */
public void showWords() { ??? }
```

14

## ArrayList methods

for `ArrayList<TYPE>`, we have already seen:

<code>TYPE get(int index)</code>	returns object at specified index
<code>TYPE add(TYPE obj)</code>	adds obj to the end of the list
<code>TYPE add(int index, TYPE obj)</code>	adds obj at index (shifts to right)
<code>TYPE remove(int index)</code>	removes object at index (shifts to left)
<code>int size()</code>	removes number of entries in list
<code>boolean contains(TYPE obj)</code>	returns true if obj is in the list (assumes TYPE has an <code>equals</code> method)

other useful methods:

<code>TYPE set(int index, TYPE obj)</code>	sets entry at index to be obj
<code>int indexOf(TYPE obj)</code>	returns index of obj in the list (assumes obj has an <code>equals</code> method)
<code>String toString()</code>	returns a String representation of the list e.g., "[foo, bar, biz, baz]"

15

## toString

the `toString` method is especially handy

- if a class has a `toString` method, it will automatically be called when printing or concatenating

```
ArrayList<String> words = new ArrayList<String>();
words.add("one");
words.add("two");
words.add("three");

System.out.println(words);           // toString method is automatically
                                     // called to convert the ArrayList
                                     // → displays "[one, two, three]"
```

- when defining a new class, you should always define a `toString` method to make viewing it easier
- `toString` is good for debugging or printing short ArrayLists, but its lack of structure does not work well for larger lists

16

## Another example: word frequencies

now consider an extension: we want a list of words and their frequencies

- i.e., keep track of how many times each word appears, report that number

basic algorithm: keep a frequency count for each unique word

```
while (STRINGS REMAIN TO BE READ) {
    word = NEXT WORD IN FILE;

    totalWordCount++;

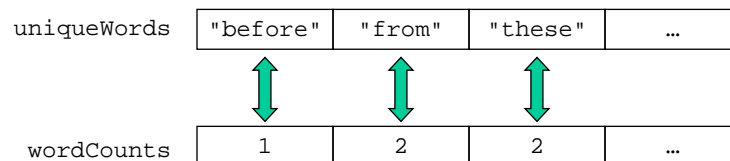
    if (ALREADY STORED IN LIST) {
        INCREMENT THE COUNT FOR THAT WORD;
    }
    else {
        ADD TO LIST WITH COUNT = 1;
    }
}
```

17

## Parallel lists

one approach to storing the words and their associated counts is with *parallel lists*

- related values are stored in separate lists, with corresponding values at corresponding indices  
e.g., `uniqueWords` stores the words  
`wordCounts` stores the frequency counts for those words



note: `wordCounts.get(i)` is the number of times that `uniqueWords.get(i)` appears in the file

18

## Autoboxing & unboxing

natural assumption: will store the frequency counts in an ArrayList of ints

```
private ArrayList<String> uniqueWords;  
private ArrayList<int> wordCounts;
```

- unfortunately, ArrayLists can only store object types (i.e., no primitives)
- fortunately, there exists an object type named `Integer` that encapsulates an `int` value
- the Java compiler will automatically
  - convert an `int` value into an `Integer` object when you want to store it in an `ArrayList` (called *autoboxing*)
  - convert an `Integer` value back into an `int` when need to apply an arithmetic operation on it (called *unboxing*)

19

```
import java.util.Scanner;  
import java.io.File;  
import java.util.ArrayList;
```

```
public class DocumentFreq  
{  
    private int numWords;  
    private ArrayList<String> uniqueWords;  
    private ArrayList<Integer> wordCounts;  
  
    public DocumentFreq(String filename) throws java.io.FileNotFoundException  
    {  
        numWords = 0;  
        uniqueWords = new ArrayList<String>();  
        wordCounts = new ArrayList<Integer>();  
  
        Scanner infile = new Scanner(new File(filename));  
        while (infile.hasNext()) {  
            String nextWord = infile.next();  
            nextWord = nextWord.toLowerCase();  
            nextWord = StringUtils.stripNonLetters(nextWord);  
  
            numWords++;  
  
            int index = uniqueWords.indexOf(nextWord);  
            if (index >= 0) {  
                int count = wordCounts.get(index);  
                wordCounts.set(index, count+1);  
            }  
            else {  
                uniqueWords.add(nextWord);  
                wordCounts.add(1);  
            }  
        }  
    }  
  
    public void showWords()  
    {  
        for (int i = 0; i < uniqueWords.size(); i++) {  
            System.out.println(uniqueWords.get(i) + ": " + wordCounts.get(i));  
        }  
        System.out.println();  
    }  
}
```

## DocumentFreq class

as each word is read in, the `ArrayList` is searched to see if it is already stored – if so, its corresponding count is incremented; if not, the word and initial count of 1 are added to the parallel lists

int values are autoboxed as `Integer`s when storing  
`Integer` values are unboxed when accessed

20

```

import java.util.Scanner;
import java.io.File;

public class ComparePlans
{
    private ServicePlan plan1;
    private ServicePlan plan2;
    private ServicePlan plan3;
    private ServicePlan plan4;
    private ServicePlan plan5;

    public ComparePlans(String filename) throws java.io.FileNotFoundException
    {
        Scanner infile = new Scanner(new File(filename));
        plan1 = new ServicePlan(infile.nextLine(), infile.nextInt(),
                                infile.nextDouble(), infile.nextDouble(),
                                infile.nextInt(), infile.nextDouble());

        infile.nextLine();
        plan2 = new ServicePlan(infile.nextLine(), infile.nextInt(),
                                infile.nextDouble(), infile.nextDouble(),
                                infile.nextInt(), infile.nextDouble());

        infile.nextLine();
        plan3 = new ServicePlan(infile.nextLine(), infile.nextInt(),
                                infile.nextDouble(), infile.nextDouble(),
                                infile.nextInt(), infile.nextDouble());

        infile.nextLine();
        plan4 = new ServicePlan(infile.nextLine(), infile.nextInt(),
                                infile.nextDouble(), infile.nextDouble(),
                                infile.nextInt(), infile.nextDouble());

        infile.nextLine();
        plan5 = new ServicePlan(infile.nextLine(), infile.nextInt(),
                                infile.nextDouble(), infile.nextDouble(),
                                infile.nextInt(), infile.nextDouble());

    }
    . . .
}

```

## Back to HW3

the last version had the advantage of being able to handle plans from different input files

but, each file had to store exactly 5 plans

plus, the code is highly repetitive!

21

```

import java.util.Scanner;
import java.io.File;
import java.util.ArrayList;

public class ComparePlans
{
    private ArrayList<ServicePlan> plans;

    public ComparePlans(String filename) throws java.io.FileNotFoundException
    {
        plans = new ArrayList<ServicePlan>();

        Scanner infile = new Scanner(new File(filename));

        ServicePlan p;
        while (infile.hasNextLine()) {
            p = new ServicePlan(infile.nextLine(), infile.nextInt(),
                                infile.nextDouble(), infile.nextDouble(),
                                infile.nextInt(), infile.nextDouble());

            infile.nextLine();

            plans.add(p);
        }

        public void comparePlans(int minutesUsed)
        {
            System.out.println("Your monthly cost for each plan would be:");
            System.out.println("-----");
            for (int i = 0; i < plans.size(); i++) {
                ServicePlan p = plans.get(i);
                System.out.printf("%s: $%5.2f%n", p.getName(),
                                    p.monthlyCost(minutesUsed));
            }
        }
    }
}

```

## Best version

instead, we could read in each ServicePlan and store it in an ArrayList

the number of plans in the file is irrelevant

the reduces duplication since the code for processing a single plan is placed in a loop

22

## Tuesday: TEST 2

cumulative, but will focus on material since last test

as before, will contain a mixture of question types

- quick-and-dirty, factual knowledge e.g., TRUE/FALSE, multiple choice
- conceptual understanding e.g., short answer, explain code
- practical knowledge & programming skills trace/analyze/modify/augment code

study advice:

- review lecture notes (if not *mentioned* in notes, will not be on test)
- read text to augment conceptual understanding, see more examples & exercises
- review quizzes and homeworks
- review TEST 1 for question formats
- feel free to review other sources (lots of Java tutorials online)