

CSC 221: Computer Programming I

Fall 2004

Lists, data access, and searching

- ArrayList class
- ArrayList methods: add, get, size, remove
- example: Notebook class

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 collection of arbitrary objects, accessible via an index

- create an `ArrayList` by calling the `ArrayList` constructor (no inputs)

```
ArrayList words = new ArrayList(); // creates an empty list
```

- 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
 - technically, `get` returns an `Object` (generic type that all objects belong to) when you access an item, you must* cast it to its actual type

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

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

for (int i = 0; i < words.size(); i++) {
    String entry = (String)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:

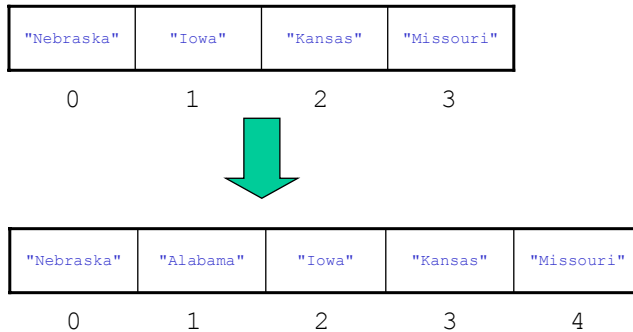
- given an `ArrayList` of state names, output index where "Hawaii" is stored

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
```



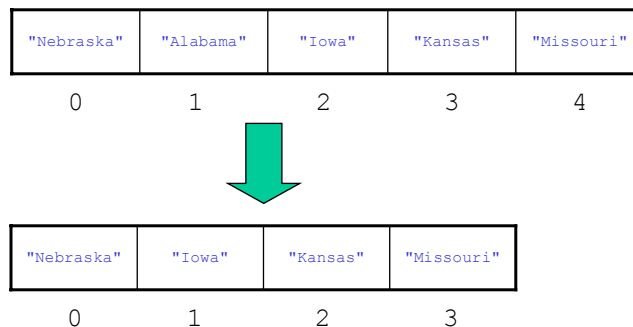
5

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 as `Strings` in an `ArrayList`
- will provide methods for adding notes, viewing the list, and removing notes

```
import java.util.ArrayList;

public class Notebook
{
    private ArrayList 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();
}

/**
 * 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 index where note is to be added
 * @param newNote note to be added to the notebook list
 */
public void storeNote(int priority, String newNote)
{
    notes.add(priority, 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 noteNumber the number of the note to be shown (first note is # 0)
 */
public void showNote(int noteNumber)
{
    if (noteNumber < 0 || noteNumber >= numberOfNotes()) {
        System.out.println("There is no note with that index.");
    }
    else {
        String entry = (String)notes.get(noteNumber);
        System.out.println(entry);
    }
}

/**
 * List all notes in the notebook.
 */
public void listNotes()
{
    System.out.println("NOTEBOOK CONTENTS");
    System.out.println("-----");
    for (int i = 0; i < notes.size(); 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 noteNumber the number of the note to be removed (first note is # 0)
 */
public void removeNote(int noteNumber)
{
    if (noteNumber < 0 || noteNumber >= numberOfNotes()) {
        System.out.println("There is no note with that index.");
    }
    else {
        notes.remove(noteNumber);
    }
}

/**
 * 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 = (String)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) can provide stats on a file

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

suppose we wanted to process a file and maintain simple stats

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

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

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

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

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 words, String desired)
{
    for (int i = 0; i < words.size(); i++) {
        String entry = (String)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

FileScanner class

a class is provided for reading words from a file

```
FileScanner reader = new FileScanner("words.txt");
```

`reader.nextString()` returns next `String` from file, delimited by whitespace

`reader.hasNext()` returns true if a `String` remains to be read from file

```
FileScanner reader = new FileScanner("words.txt");

while (reader.hasNext()) {
    word = reader.nextString();
    word = word.toLowerCase();

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

14

Input exceptions

when processing files, there is a possibility of things going wrong, e.g.,

- try to read from a file that doesn't exist
- try to read a value when the file is empty
- what should the program do when an error occurs?

in Java, you either have to specify code to handle input errors

```
try {
    reader = new FileScanner("words.txt");
}
catch (FileNotFoundException e) {
    /* CODE TO HANDLE THE ERROR */
}
```

or, simply "throw" an exception

- add "throws EXCEPTION_NAME" to the end of any method in danger
- a thrown exception causes an error message to appear and execution to terminate

15

```
import java.io.*;
import java.util.ArrayList;
```

```
public class FileStats
```

```
{
    private int totalWordCount;
    private ArrayList uniqueWords;

    public FileStats(String filename) throws FileNotFoundException
    {
        totalWordCount = 0;
        uniqueWords = new ArrayList();

        FileScanner reader = new FileScanner(filename);
        while (reader.hasNext()) {
            String word = reader.nextString();
            word = word.toLowerCase();

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

    public int getTotalWords()
    {
        return totalWordCount;
    }

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

FileStats class

constructor throws an exception if the file is not found – need to load the java.io library collection to recognize the exception

FileScanner is used to read words from the file

an ArrayList is used to store unique words

16

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)