Grouping Objects

• Many applications require a grouping of objects. We have already seen this:
  – The canvas in our shapes project contained a collection of geometric objects
  – The labclass in our LabClass project contained a collection of students
  – The library in our Books project contained a collection of individual books

• In this chapter we will learn to group objects together in a single composite object and learn how to write methods that examine and change this composite object
Types of Collections

• Collections that vary in size
  – Java offers a wide variety of collections that can vary in size getting larger as new objects are added or getting smaller as objects are removed
  – In this chapter we will study an ArrayList where a sequence of items are indexed and an individual item can be examined or changed based on its index number

• Collections with a fixed size, known as arrays
  – For some applications it is appropriate to fix the number of items to be grouped and allocate all the memory that will be needed at the time of declaration
  – We will study the use of arrays
  – Arrays are common to most programming languages, including Fortran that was first designed in the 1950s
Processing Collections

• In working with collections, it is necessary to have a systemic way to visit each of the objects in the collection

• To process items in an ArrayList, we will use a while loop to visit each item in sequence

• Built-in collections in Java also provide iterators that are objects themselves and provide systematic access to objects in a collection

• To process items in a fixed size array, we will use a for loop to visit each element of the array

• While loops and for loops are general constructs in Java that are used to implement repetition
Using the Java Libraries

• The Java programming environment provides access to thousands of library classes that are organized into packages

• Learning to use these libraries (or APIs as they are sometimes called) is a critically important skill for learning how to program in Java

• These libraries are free and have been thoroughly tested, so using these packages gives you a major head start in developing your application

• Libraries have to be imported; the first library we will import is for the ArrayList

    import java.util.ArrayList;
An Auction System

• There are four classes in this project: Auction, Bid, Lot, and Person

• The Person class is very simple and only stores the bidder’s name

• The Bid class is equally simple only storing the bidder and the amount bid

• The Lot class contains a description of the lot, the lot number, and details about the highest bid

• A bid becomes the highest bid if either it is the first bid or it is higher than the current highest bid

• Bid values are type long, an extended precision integer value so that it can hold big numbers
/**
 * A class to model an item (or set of items) in an
 * auction: a lot.
 * @author David J. Barnes and Michael Kolling.
 * @version 2001.06.08
 */

public class Lot {

    // A unique identifying number.
    private final int number;
    // A description of the lot.
    private String description;
    // The current highest bid for this lot.
    private Bid highestBid;

    /**
     * Construct a Lot, setting its number and description.
     * @param number The lot number.
     * @param description A description of this lot.
     */
    public Lot(int number, String description) {
        this.number = number;
        this.description = description;
    }

    // methods go here
}
/**
 * Attempt to bid for this lot. A successful bid
 * must have a value higher than any existing bid.
 * @param bidder Who is bidding.
 * @param value The value of the bid.
 */

double bid(person bidder, long value)
{
    if((highestBid == null) ||
        (highestBid.getValue() < value)) {
        // This bid is the best so far.
        setHighestBid(new Bid(bidder, value));
    } else {
        System.out.println("Lot number: " + getNumber() +
            " (" + getDescription() + ")" +
            " already has a bid of: " +
            highestBid.getValue());
    }
}

/**
 * @return The lot's number.
 */

double getNumber()
{
    return number;
}
/**
 * @return The lot's description.
 */
public String getDescription()
{
    return description;
}

/**
 * @return The highest bid for this lot. This could be null if there are no current bids.
 */
public Bid getHighestBid()
{
    return highestBid;
}

/**
 * @param highestBid The new highest bid.
 */
private void setHighestBid(Bid highestBid)
{
    this.highestBid = highestBid;
}
import java.util.*;

/**
 * A simple model of an auction.
 * The auction maintains a list of lots of arbitrary length.
 * @author David J. Barnes and Michael Kolling.
 * @version 2001.06.08
 */

public class Auction {
    // The list of Lots in this auction.
    private ArrayList lots;
    // The number that will be given to the next lot entered
    // into this auction.
    private int nextLotNumber;

    /**
     * Create a new auction.
     */
    public Auction()
    {
        lots = new ArrayList();
        nextLotNumber = 1;
    }

    // methods go here
}
/**
 * Enter a new lot into the auction.
 * Lots can only be entered into the auction by an
 * Auction object.
 * @param description A description of the lot.
 */
public void enterLot(String description)
{
    lots.add(new Lot(nextLotNumber, description));
    nextLotNumber++;
}

/**
 * Show the full list of lot numbers and lot descriptions in
 * this auction. Include any details of the highest bids.
 */
public void showLots()
{
    Iterator it = lots.iterator();
    while(it.hasNext()) {
        Lot lot = (Lot) it.next();
        System.out.println(lot.getNumber() + " : " +
                            lot.getDescription());
        // Include any details of a highest bid.
        Bid highestBid = lot.getHighestBid();
        if(highestBid != null) {
            System.out.println("    Bid: " +
                                highestBid.getValue());
        }
        else {
            System.out.println("    (No bid)");}
    }
}
/**
* Return the lot with the given number. Return null if a lot with this number does not exist.
* @param number The number of the lot to return.
*/
public Lot getLot(int number)
{
    if((number >= 1) && (number < nextLotNumber)) {
        // The number seems to be reasonable.
        Lot selectedLot = (Lot) lots.get(number-1);
        // Include a confidence check to be sure we have the right lot.
        if(selectedLot.getNumber() != number) {
            System.out.println("Internal error: " +
            "Wrong lot returned. " +
            "Number: " + number);
        }
        return selectedLot;
    }
    else {
        System.out.println("Lot number: " + number +
                           " does not exist.");
        return null;
    }
}
The Personal Notebook Project

• The personal notebook allows the user to store a sequence of notes; the size of the collection can vary as the notebook is used by the user

• The notes will be a sequence of objects; suppose the notebook contains “size” notes
  – Each note has a position in the collection expressed as an integer value
  – The first note is at position 0
  – The last note is at position size-1

• Visit the project entitled notebook1 and investigate the behavior of this collection of notes
import java.util.ArrayList;
/**
 * A class to maintain an arbitrarily long list of notes. 
 * Notes are numbered for external reference by a human user.
 * In this version, note numbers start at 0.
 * @author David J. Barnes and Michael Kolling.
 * @version 2001.06.08
 */
public class Notebook
{
    // Storage for an arbitrary number of notes.
    private ArrayList notes;

    /**
     * Perform any initialization that is required for the
     * notebook.
     */
    public Notebook()
    {
        notes = new ArrayList();
    }

    // methods to manipulate the notes go here
}
/**
 * Store a new note into the notebook.
 * @param note The note to be stored.
 */
public void storeNote(String note) {
    notes.add(note);
}

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

/**
 * Show a note.
 * @param noteNumber The number of the note to be shown.
 */
public void showNote(int noteNumber) {
    if ((noteNumber < 0) || (noteNumber >= numberOfNotes())) {
        // This is not a valid note number, so do nothing.
    }
    else if (noteNumber < numberOfNotes()) {
        // This is a valid note number, so we can print it.
        System.out.println(notes.get(noteNumber));
    }
}
Adding More Functionality

• Notes may become outdated, so it would be nice to have a method that removes an existing note
  – What will this do to the size of the collection?
  – What will this do to the index numbers of the remaining notes? (be very specific in your answer)

• You may want to see a list of all existing notes at one time
  – This will require a way to sequence through the notes from beginning to end
  – You will learn two techniques for doing this

• Visit the BlueJ lab and run the project Notebook2
Removing a Note

• To remove a note, the user must know its index number; the signature of the method will be `public void removeNote(int noteNumber)`

• What is a valid range for the `noteNumber`?
• How will this be detected in the method?

• What should happen if the given `noteNumber` is not valid?
• When would “do nothing” be a good option?
• When would display an explicit error message be a good option?
/**
 * Remove a note from the notebook if it exists.
 * @param noteNumber The number of the note to be removed.
 */

public void removeNote(int noteNumber) {
    if(noteNumber < 0) {
        // This is not a valid note number, so do nothing.
    }
    else if(noteNumber < numberOfNotes()) {
        // This is a valid note number.
        notes.remove(noteNumber);
    }
    else {
        // This is not a valid note number, so do nothing.
    }
}
Improving the removeNote method

• The first criticism of the code is the structure of the if statement; it should be possible to have only two alternative, the index is valid or the index is invalid

• Redo the code to reflect these two alternatives

• The second criticism is to do nothing if the index is invalid; the user may need to be told what the error is and how it can be corrected

• Redo the code so that a brief but informative error message is printed
Printing the List of Notes

- An ArrayList is a particularly easy collection since the items in the collect have a linear index going from 0 to the size of the collection minus one

- Here is the method

```java
/**
 * List all notes in the notebook.
 */
public void listNotes()
{
    int index = 0;
    while(index < notes.size()) {
        System.out.println(notes.get(index));
        index++;
    }
}
```

- A user of this method requested that the index as well as the note should be printed. Make this change.
Iterating Over Collections

• Visiting each item in a collection one at a time is a task performed for many applications, so Java has a built-in Iterator class to help do this work.

• When you declare an Iterator object, you need to specify a particular instance of a collection, as in:

```java
Iterator it = notes.iterator();
```

• Two methods are used to perform the iteration:
  – `<iterator>.hasNext()` returns true if there is another item in the collection, otherwise it returns false.
  – `<iterator>.next()` returns the next object in the collection; the object is not removed from the collection, only the value is returned.
Redo listNotes to use an Iterator

- Rewrite the listNotes method using an iterator
- Then add the feature of printing the index value as well as the note
- Which approach did you like: sequencing through the index values explicitly or using an Iterator
- The ArrayList is perhaps the easiest collection built into the java utilities, so it is easy to write your own iterator directly
- Many of the other collections are much more complex, but every collection has a built in iterator that you can use “for free”
Lab9 – Modification to the Notebook

• In the notebook2 project when you add a new note to your notebook, it is always added as the last note.
• You will add a new method that will allow the user to insert a note at a specified position
• Notes already in the notebook beyond the specified position will have their index values increased by one.

• When the list of notes is printed, every note is printed.
• This may be inconvenient for a large notebook.
• You will add a new method that will allow a subrange of notes to be printed as specified by a starting index and an ending index, inclusive.
Fixed-size Collections

• The primary data structure for a fixed size collection is the array
  – Arrays are present in almost every programming language since the 1950s
  – There are a standard way to organize homogeneous data, that is data of a single type
  – Arrays can be single or multi-dimensional
  – Many well-known algorithms process arrays
  – Arrays are more efficient than other collections
  – Arrays can store both primitive types and objects

• Some disadvantages
  – All of the space must be allocated even if unused
  – Arrays cannot expand beyond their declared sizes
A log-file Analyzer

- This program assumes that the times for thousands of webpage hits have been logged, as shown here.
- Try to interpret this data, it represents a time; is it ordered?
- An analyzer program should be able to answer a variety of questions:
  - Regardless of day, what hour of the day is the most busy or the least busy?
  - Considering the days of the week, what day has the most traffic or least traffic?
  - If a system requires two hours downtime a week for maintenance, what two hour block should be used?
Using fixed-sized arrays

• Since there are always the same number of hours in a day or the same number of days in a week, etc. fixed sized arrays are appropriate

• If we wanted to count the number of hits for each hour in the day, the array would have to store 24 integer values. Here is the declaration and initialization:

```java
private int [] hourCounts;  hourCounts = new int [24];
```

• What would you need to count the hits for the days of the week?

• What would you need to count the hits for each two hour period over a week?
More About Arrays - 1

- Array elements can be accessed individually; suppose we have `int[] dayCounts = new int[7];`
  - Conceptually suppose the first element of the array, `dayCounts[0]` corresponds to Sunday and the last element, `dayCounts[6]`, corresponds to Saturday
  - What is being tested in the following boolean expr.?
    - `dayCounts[2] > 1000`

- We can also change individual array elements
  - `dayCounts[0] = 0;`
  - `dayCounts[3] += 1; this can also be accomplished by dayCounts[3]++;`
More About Arrays - 2

• We can declare all the initial values in an array and set the size implicitly, as in
  `int[] dayCounts = {10, 20, 30, 30, 20, 40, 40};`

• We can declare multi-dimensional arrays, such as a two dimensional array where the first index represents the day of the week, 0..6, and the second index represents one of 4 locations, 0..3
  – `int[][] attendance = new int [7][4];`
  – What does `attendance[2][3]` access?
  – How would we increase the attendance on Friday at location 2 by one person?
The For Loop

- Arrays are commonly processed using a for loop
- The general syntax is:
  
  ```java
  for(<initialization>; <condition>; <post-body action>)
  ```
- For example, to print out the dayCounts array
  ```java
  for(int day = 0; day < 7; day++)
      { System.out.println(dayCounts[day]); } 
  ```
- Suppose we wanted to find the highest count in the dayCounts array:
  ```java
  int highest = dayCounts[0];
  for(int day = 1; day < 7; day++)
      { if(highest < dayCounts[day])
          { highest = dayCounts[day]; } }
  System.out.println("The most hits in one day is " + highest);
  ```
The Weblog-Analyzer Project

• There are four classes: LogAnalyzer, LogFileReader, LogEntry, and LoglineTokenizer

• We will concentrate on the LogAnalyzer itself and assume the method name in the other classes make it clear what the method does, for example,
  – hasMoreEntries() in the log reader returns true if there are unread entries and false if all entries have been read
  – nextEntry() fetches the next unread entry which is returned as type LogEntry
  – getHour() for a single log entry returns the hour value for that entry
A Sample Task

• Read all the data and count the number of hits for each hour of the day

• There will be three methods
  – analyzeHourlyData: read each entry and increments one of the hour counts based on the time
  – printData: prints all of the raw data, it is very lengthy
  – printHourlyCounts: for each hour, from 0 to 23, print the number of hits detected

<table>
<thead>
<tr>
<th>Hr</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>149</td>
</tr>
<tr>
<td>1</td>
<td>149</td>
</tr>
<tr>
<td>2</td>
<td>148</td>
</tr>
<tr>
<td>3</td>
<td>109</td>
</tr>
<tr>
<td>4</td>
<td>92</td>
</tr>
<tr>
<td>5</td>
<td>177</td>
</tr>
</tbody>
</table>

and so forth
public class LogAnalyzer
{

    private int[] hourCounts;

    private LogfileReader reader;

    public LogAnalyzer()
    {
        hourCounts = new int[24];
        reader = new LogfileReader();
    }
}

The Methods

/**
 * Analyze the hourly access data from the log file.
 */
public void analyzeHourlyData()
{
    while(reader.hasMoreEntries()) {
        LogEntry entry = reader.nextEntry();
        int hour = entry.getHour();
        hourCounts[hour]++;
    }
}
/**
 * Print the hourly counts.
 * These should have been set with a prior
 * call to analyzeHourlyData.
 */
public void printHourlyCounts()
{
    System.out.println("Hr: Count");
    for(int hour = 0; hour < hourCounts.length; hour++) {
        System.out.println(hour + ": " + hourCounts[hour]);
    }
}
/**
 * Print the lines of data read by the LogfileReader
 */
public void printData()
{  reader.printData();}
Labs 10 & 11: Addition Analyses

- Lab10: Extend the hours analysis to add two new methods: totalHits that returns the sum of all the hits for all the data & busiestHour that return the complete log entry for the hour with most hits.

- Lab11: This is a programming assignment to be completed outside of the lab and turned in by class time on Tuesday June 8.
  - Rewrite the LogAnalyzer class so that it looks at the data from the perspective of the weekday.
  - Write methods to print the total number of hits for each of the seven week days, find the busiest and quietest days of the week (Hint: the first data item on May 1, 2002 was a Wednesday).