Exercise 4.2

```
private ArrayList<Book> library;
```

Exercise 4.3

```
10
```

Exercise 4.4

```
items.get(4);
```

Exercise 4.5

```
14
```

Exercise 4.6

```
notes.add(meeting);
```

Exercise 4.7

```
dates.remove(2);
```

Exercise 4.8

```
5
```

Exercise 4.9

```
    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, so we can remove it.
            notes.remove(noteNumber);
        }
        else {
            // This is not a valid note number, so do nothing.
        }
    }
```

Exercise 4.10

```
public void listAllNotes()
```

Exercise 4.11

No. We have no idea how many lines we would need.

Exercise 4.16

```
    public void showNote(int noteNumber)
    {
        if(noteNumber < 0) {
            System.out.println("This is not a valid note number");
        }
        else if(noteNumber <= numberOfNotes()) {
            // This is a valid note number, so we can print it.
            System.out.println(notes.get(noteNumber));
        }
        else {
            System.out.println("This is not a valid note number");
        }
    }
```

Exercise 4.17

```
public void multiplesOfFive()
{
    int multiple = 10;
    while(multiple <= 95) {
        System.out.println(multiple);
        multiple = multiple + 5;
    }
}
```

Exercise 4.18

```
/**
 * Sum the numbers from a to b, inclusive.
 */
public int sum(int a, int b)
{
    int sum = 0;
    int number = a;
    while(number <= b) {
        sum = sum + number;
        number = number + 1;
    }
    return sum;
}
```

Exercise 4.19

```
/**
 * A crude determination of whether n is prime or not.
 * More efficient methods are possible.
 */
public boolean isPrime(int n)
{
    int divisor = 2;
    while(divisor < n ) {
        if(n % divisor != 0) {
            return false;
        }
        divisor = divisor + 1;
    }
    return true;
}
```

Exercise 4.20

```
public boolean search(String searchString)
{
    int index = 0;
    boolean found = false;
    while(index < notes.size() && !found) {
        String note = notes.get(index);
        if(note.contains(searchString)) {
            found = true;
        }
        else {
            index++;
        }
    }
    if(found) {
        System.out.println("Found search term in note: " +
notes.get(index));
    }
    else {
        System.out.println("Search term not found.");
    }
}
```

Exercise 4.21

```
    public void listNotes()
    {
        int index = 0;
        for(String note : notes) {
            System.out.println(index + ": " + note);
            index++;
        }
    }
```

Exercise 4.22

The value does not vary.

```
public boolean search(String searchString)
{
    int index = 0;
    boolean found = false;
    int size = notes.size();

    while(index < size && !found) {
        String note = notes.get(index);
        if(note.contains(searchString)) {
            found = true;
        }
        else {
            index++;
        }
    }
    if(found) {
        System.out.println("Found search term in note: " +
notes.get(index));
    }
    else {
        System.out.println("Search term not found.");
    }
}
```

Exercise 4.23

```
    /**
     * Show a note.
     * @param noteNumber The number of the note to be shown.
     */
    public void showNote(int noteNumber)
    {
        if(noteNumber < 1) {
            // 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 - 1));
        }
        else {
            // This is not a valid note number, so do nothing.
        }
    }

    /**
     * List all notes in the notebook.
     */
    public void listNotes()
    {
        int noteNumber = 1;
        for(String note : notes) {
            System.out.println(noteNumber + ": " + note);
            noteNumber++;
        }
    }

    /**
     * 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 < 1) {
            // This is not a valid note number, so do nothing.
        }
        else if(noteNumber <= numberOfNotes()) {
            // This is a valid note number.
            notes.remove(noteNumber - 1);
        }
        else {
            // This is not a valid note number, so do nothing.
        }
    }
```

Exercise 4.25-4.27

```
import java.util.ArrayList;

/**
 * Store details of club memberships.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Club
{
    private ArrayList<Membership> members;

    /**
     * Constructor for objects of class Club
     */
    public Club()
    {
        members = new ArrayList<Membership>();
    }

    /**
     * Add a new member to the club's list of members.
     * @param member The member object to be added.
     */
    public void join(Membership member)
    {
        members.add(member);
    }

    /**
     * @return The number of members (Membership objects) in
     *         the club.
     */
    public int numberOfMembers()
    {
        return members.size();
    }
}
```

Exercise 4.28

```
    public void close()
    {
        for(Lot lot : lots) {
            System.out.println(lot.getNumber() + ": " +
                               lot.getDescription());
            // Include any details of a highest bid.
            Bid highestBid = lot.getHighestBid();
            if(highestBid != null) {
                System.out.println("    Highest bidder: " +
                                   highestBid.getBidder().getName());
                System.out.println("    Bid: " +
                                   highestBid.getValue());
            }
            else {
                System.out.println("    Not sold");
            }
        }
    }
```

Exercise 4.29

```
    /**
     * Returns a list of unsold lots
     */
    public ArrayList<Lot> getUnsold()
    {
        ArrayList<Lot> unsoldLots = new ArrayList<Lot>();
        for(Lot lot : lots) {
            Bid highestBid = lot.getHighestBid();
            if(highestBid == null) {
                unsoldLots.add(lot);
            }
        }
        return unsoldLots;
    }
```

Exercise 4.30

The getLot method assumes that a Lot is stored at location getLotNumber()-1 in its ArrayList. If lots can be removed then index numbers may be changed. The getLot method always checks for consistency so if there is an inconsistency the an error message is printed in the terminal window.

Exercise 4.31

```
    /**
     * 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)
    {
        for(Lot lot : lots) {
            if(lot.getNumber() == number) {
                return lot;
            }
            else if (lot.getNumber() > number) {
                System.out.println("Lot number: " + number +
                                   " does not exist.");
                return null;
            }
        }
        return null;
    }
```

Exercise 4.32

```
    /**
     * Remove the lot with the given lot number.
     * @param number The number of the lot to be removed
     * @return The Lot with the given number, or null if
     * there is no such lot.
     */
    public Lot removeLot(int number) {
        //First we find the lot with the given number
        Lot lot = getLot(number);
        if(lot != null) {
            //Then we can use the method remove with lot as argument
            lots.remove(lot);
        }
        return lot;
    }
```

Exercise 4.33

The LinkedList has these methods that ArrayList does not have:

void addFirst(Object o)
void addLast(Object o)
Object getFirst()
Object getLast()
Object removeFirst()
Object removeLast()

The ArrayList has these methods that LinkedList does not have:

void ensureCapacity(int minCapacity)
void trimToSize()

Exercise 4.34

```
    /**
     * Determine the number of members who joined in the
     * given month
     * @param month The month we are interested in.
     * @return The number of members.
     */
    public int joinedInMonth(int month)
    {
        int count = 0;

        if(month < 1 || month > 12) {
            System.out.println("Month " + month + " out of range.
Must be in the range 1 ... 12");
        }
        else {
            for(Membership member : members) {
                if(member.getMonth() == month) {
                    count++;
                }
            }
        }
        return count;
    }
```

Exercise 4.35

```
    public ArrayList purge(int month, int year)
    {
        ArrayList<Membership> purged = new ArrayList<Membership>();
        if(month < 1 || month > 12) {
            System.out.println("Month " + month + " out of range.
Must be in the range 1 ... 12");
        }
        else {
            Iterator<Membership> it = members.iterator();
            while(it.hasNext()) {
                Membership member = it.next();
                if(member.getMonth() == month && member.getYear() ==
year) {
                    // Must use the remove method from the iterator.
                    // Check the documentation for the Iterator for
more info.
                    it.remove();
                    purged.add(member);
                }
            }
        }
        return purged;
    }
```

Exercise 4.36

```
    public void printProductDetails()
    {
        for(Product product : stock) {
            System.out.println(product.toString());
        }
    }
```

Exercise 4.37

```
    public Product findProduct(int id)
    {
        for(Product product : stock) {
            if(product.getID() == id) {
                return product;
            }
        }
        return null;
    }
```

Exercise 4.38

```
    public int numberInStock(int id)
    {
        Product product = findProduct(id);
        if(product != null) {
            return product.getQuantity();
        }
        else {
            return 0;
        }
    }
```

Exercise 4.39

```
    public void delivery(int id, int amount)
    {
        Product product = findProduct(id);
        if(product != null) {
            product.increaseQuantity(amount);
        }
    }
```

Exercise 4.40

```
    /**
     * Print details of all the products which has stock
     * levels below the given amount
     */
    public void printLowStockProducts(int upperLimit)
    {
        for(Product product : stock) {
            if(product.getQuantity() < upperLimit) {
                System.out.println(product.toString());
            }
        }
    }

    /**
     * Add a product to the list.
     * @param item The item to be added.
     */
    public void addProduct(Product item)
    {
        if( ! stock.contains(item)) {
            stock.add(item);
        }
    }

    /**
     * Try to find a product in the stock with the given name.
     * @return The identified product, or null if there is none
     *         with a matching name.
     */
    public Product findProduct(String name)
    {
        for(Product product : stock) {
            if(product.getName().equals(name)) {
                return product;
            }
        }
        return null;
    }
```

Exercise 4.41

The busiest time of day: 18

Exercise 4.42

```
Person[] people;
```

Exercise 4.43

```
boolean[] vacant;
```

Exercise 4.45

```
int[] counts;

boolean[] occupied = new boolean[5000];
```

Exercise 4.46

```
readings = new double[60];
urls = new String[90];
machines = new TicketMachine[5];
```

Exercise 4.47

None. It only creates an array to hold String objects.

Exercise 4.48

The brackets must be square rather than round.

```
double[] prices = new double[50]
```

Exercise 4.49

It throws an ArrayIndexOutOfBoundsException: 24

Exercise 4.50

```
    /**
     * Print the hourly counts.
     * These should have been set with a prior
     * call to analyzeHourlyData.
     */
    public void printHourlyCounts()
    {
        System.out.println("Hr: Count");
        int hour = 0;
        while(hour < hourCounts.length) {
            System.out.println(hour + ": " + hourCounts[hour]);
            hour++;
        }
    }
```

Exercise 4.51

```
    public void printGreater(double[] marks, double mean)
    {
        for(int index = 0; index < marks.length; index++) {
            if(marks[index] > mean) {
                System.out.println(marks[index]);
            }
        }
    }
```

Exercise 4.52

```
    public void listNotes()
    {
        for(int index = 0; index < notes.size(); index++) {
            System.out.println(notes.get(index));
        }
    }
```

Exercise 4.53

```
    public void listNotes()
    {
        for(String note : notes) {
            System.out.println(note);
        }
    }
```

Exercise 4.54

```
    /**
     * Return the number of accesses recorded in the log file
     */
    public int numberOfAccesses()
    {
        int total = 0;
        // Add the value in each element of hourCounts to total.
        for(int hourCount : hourCounts) {
            total = total + hourCount;
        }
        return total;
    }
```

Exercise 4.56

```
    /**
     * Return the busiest hour of day
     */
    public int busiestHour()
    {
        int busiestHour = 0;
        for(int hour = 1; hour < hourCounts.length; hour++) {
            if(hourCounts[hour] > hourCounts[busiestHour]) {
                busiestHour = hour;
            }
        }
        return busiestHour;
    }
```

Exercise 4.57

```
    /**
     * Return the quietest hour of day
     */
    public int quietestHour()
    {
        int quietestHour = 0;
        for(int hour = 1; hour < hourCounts.length; hour++) {
            if(hourCounts[hour] < hourCounts[quietestHour]) {
                quietestHour = hour;
            }
        }
        return quietestHour;
    }
```

Exercise 4.58

In the above implementation, it is the first one that is found.

Exercise 4.59

```
    /**
     * Return the two-hour period which is busiest.
     */
    public int busiestTwoHourPeriod()
    {
        int busiestPeriod = 0;
        int busiestPeriodCount = 0;
        for(int hour = 0; hour < hourCounts.length-1; hour++) {
            int periodCount = hourCounts[hour] + hourCounts[hour+1];
            if(periodCount > busiestPeriodCount) {
                busiestPeriod = hour;
                busiestPeriodCount = periodCount;
            }
        }
        return busiestPeriod;
    }
```

Exercise 4.63

Reasons for choosing a fixed size array could be:

- Performance is slightly better.
- Not so good if students are added and removed from time to time.

Reasons for keeping the dynamically sized list:

- No need to keep track of the current number of students.
- Good for future enhancements (for instance if we want to have a method to remove a student from the list).

Exercise 4.65

```
    public void listNotes()
    {
        if(! notes.isEmpty()) {
            int index = 0;
            do {
                System.out.println(notes.get(index));
                index++;
            } while(index < notes.size());
        }
    }
```