Exercise 4.4

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

Exercise 4.5

```
ArrayList<Student> cs101;
```

Exercise 4.6

```
private ArrayList<MusicTrack> tracks;
```

Exercise 4.7

```
library = new ArrayList<Book>();
cs101 = new ArrayList<Student>();
tracks = new ArrayList<MusicTrack>();
```

with Java 7, these can be simplified to:

```
library = new ArrayList<>();
cs101 = new ArrayList<>();
tracks = new ArrayList<>();
```

Exercise 4.8

```
10
```

Exercise 4.9

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

Exercise 4.10

```
14
```

Exercise 4.11

```
files.add(favoriteTrack);
```

Exercise 4.12

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

Exercise 4.13

```
5
```

Exercise 4.14

```
public void checkIndex(int index)
{
    if(index < 0 || index >= files.size()) {
        System.out.println("The valid range is 0 to " +
```

```
                            (files.size() - 1));
    }
}
```

If the collection is empty, there is no valid index and the error message will still appear. This special case could be catered for my a separate if statement that checks for an empty collection and prints an error message specific to that case.

Exercise 4.15

The following are all valid. The first is in the spirit of the previous exercise. The second rewrites to the opposite form and the third gives the most concise representation.

```
public boolean validIndex(int index)
{
    if(index < 0 || index >= files.size()) {
        return false;
    }
    else {
        return true;
    }
}

public boolean validIndex(int index)
{
    if(index >= 0 && index < files.size()) {
        return true;
    }
    else {
        return false;
    }
}

public boolean validIndex(int index)
{
    return index >= 0 && index < files.size();
}
```

Exercise 4.16

```
    /**
     * List a file from the collection.
     * @param index The index of the file to be listed.
     */
    public void listFile(int index)
    {
        if(validIndex(index)) {
            String filename = files.get(index);
            System.out.println(filename);
        }
    }

    /**
     * Remove a file from the collection.
     * @param index The index of the file to be removed.
```

```
     */
    public void removeFile(int index)
    {
        if(validIndex(index)) {
            files.remove(index);
        }
    }
```

Exercise 4.18

```
public void listAllFiles()
```

Exercise 4.19

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

Exercise 4.24

```
    /**
     * Show a list of all the files in the collection,
     * along with their index values.
     */
    public void listAllFiles()
    {
        // A variable to keep track of the index position.
        int position = 0;
        for(String filename : files) {
            System.out.println(position + ": " + filename);
            position++;
        }
    }
```

Exercise 4.26

```
    /**
     * List the names of files matching the given search string.
     * If none are found, print a message.
     * @param searchString The string to match.
     */
    public void listMatching(String searchString)
    {
        // Assume that there will be no matches.
        boolean noMatch = true;

        for(String filename : files) {
            if(filename.contains(searchString)) {
                System.out.println(filename);
                // We found at least one match.
                noMatch = false;
            }
        }

        if(noMatch) {
            System.out.println("No files matched: " +
                               searchString);
        }
    }
```

Exercise 4.27

```
    /**
     * Play a sample from all files matching the given search string.
     * @param artist The string to match.
     */
    public void playSamplesBy(String artist)
    {
        int position = 0;

        for(String filename : files) {
            if(filename.contains(artist)) {
                playAndWait(position);
            }
            position++;
        }
    }
```

Exercise 4.28

```
for(Track track : tracks)
```

Exercise 4.29

```
boolean found = false;
while(!found) {
    if(the keys are in the next place) {
        found = true;
    }
}
```

Exercise 4.30

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

Exercise 4.31

```
int sum = 0;
int num = 1;

while(num <= 10) {
    sum += num;
    num++;
}
System.out.println("The sum of the values 1 to 10 is " + sum);
```

Exercise 4.32

```
/**
 * 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;
}
```

```
 * 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 4.34

The value does not vary.

```
public int findFirst(String searchString)
{
    int index = 0;
    // Record that we will be searching until a match is found.
    boolean searching = true;
    int size = files.size();

    while(searching && index < size) {
        String filename = files.get(index);
        if(filename.contains(searchString)) {
            // A match. We can stop searching.
            searching = false;
        }
        else {
            // Move on.
            index++;
        }
    }
    if(searching) {
        // We didn't find it.
        return -1;
    }
    else {
        // Return where it was found.
        return index;
```

```
    }
}
```

Exercise 4.38

Add:

```
      stopPlaying();
```

to `playTrack()` once the index has been validated.

Exercise 4.39

```
    /**
     * Remove tracks whose titles match the search string.
     * A partial match is used.
     * @param title The string to be matched.
     */
    public void removeTitles(String title)
    {
        Iterator<Track> it = tracks.iterator();
        while(it.hasNext()) {
            Track t = it.next();
            if(t.getTitle().contains(title)) {
                it.remove();
            }
        }
    }
```

Exercise 4.40-4.42

```
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 4.43

```
    /**
     * Select and play a single random track.
     */
    public void randomPlay()
    {
        if(tracks.size() > 0) {
            Random rand = new Random();
            int index = rand.nextInt(tracks.size());
            playTrack(index);
        }
    }
```

Exercise 4.45

The `Collections.shuffle()` method offers a way to randomly order a collection. This is used in the first version, which does not require a separate random number generator:

```
    /**
     * Play all tracks once in a random order.
     */
    public void randomPlayAll()
    {
        ArrayList<Track> leftToPlay = new ArrayList<Track>(tracks);
        Collections.shuffle(leftToPlay);
        for(Track t : leftToPlay) {
            player.playSample(t.getFilename());
        }

    }
```

The second version makes a copy of the list and repeatedly removes a random track to be played until the list is empty.

```
    /**
     * Play all tracks once in a random order.
     */
    public void randomPlayAll()
    {
        Random rand = new Random();
        ArrayList<Track> leftToPlay = new ArrayList<Track>(tracks);
```

```
        while(leftToPlay.size() > 0) {
            int index = rand.nextInt(leftToPlay.size());
            Track t = leftToPlay.remove(index);
            player.playSample(t.getFilename());
        }
    }
```

Exercise 4.47

```
boolean successful = selectedLot.bidFor(new Bid(bidder, value));
```

Exercise 4.48

```
    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 4.49

```
    /**
     * 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 4.50

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 4.51

```
    /**
     * 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)
    {
        Lot lot = null;
        if(lots.size() > 0) {
            lot = lots(0);
            int nextIndex = 1;
            while(lot.getNumber() != number && nextIndex <
lots.size()) {
                lot = lots.get(nextIndex);
                nextIndex++;
            }
        }
        if (lot == null || lot.getNumber() != number) {
            System.out.println("Lot number: " + number +
                               " does not exist.");
            return null;
        }
        else {
            return lot;
        }
    }
```

Exercise 4.52

```
    /**
     * 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 4.53

`LinkedList` has these methods that `ArrayLis`t does not have:

```
void addFirst(E o)
void addLast(E o)
E getFirst()
E getLast()
E removeFirst()
```

```
E removeLast()
```

Where '`E`' refers to the type of item stored in the list.

`ArrayList` has these methods that `LinkedList` does not have:

```
void ensureCapacity(int minCapacity)
void trimToSize()
```

Exercise 4.54

```
    /**
     * 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. " +
                               "It must be in the range 1 ... 12");
        }
        else {
            for(Membership member : members) {
                if(member.getMonth() == month) {
                    count++;
                }
            }
        }
        return count;
    }
```

Exercise 4.55

```
    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. " +
                               "It 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 4.56

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

Exercise 4.57

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

Exercise 4.58

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

Exercise 4.59

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

Exercise 4.60

```
    /**
     * 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 4.61

The busiest time of day: 18

Exercise 4.62

```
Person[] people;
```

Exercise 4.63

```
boolean[] vacant;
```

Exercise 4.65

```
int[] counts;

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

Exercise 4.66

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

Exercise 4.67

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

Exercise 4.68

The brackets must be square rather than round.

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

Exercise 4.69

It throws an ArrayIndexOutOfBoundsException: 24

Exercise 4.70

```
    /**
     * 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 4.71

```
    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 4.72

```
    /**
     * Create an object to analyze hourly web accesses.
     * @param filename The file to be analyzed.
     */
    public LogAnalyzer(String filename)
    {
        // Create the array object to hold the hourly
        // access counts.
        hourCounts = new int[24];
        // Create the reader to obtain the data.
        reader = new LogfileReader(filename);
    }
```

Exercise 4.73

```
/**
 * 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 4.75

```
/**
 * 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 4.76

```
/**
 * 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 4.77

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

Exercise 4.78

```
/**
 * 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 4.82

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 4.83

```
    /**
     * Show a list of all the files in the collection.
     */
    public void listAllFiles()
    {
        for(int i = 0; i < files.size(); i++) {
            System.out.println(files.get(i));
        }
    }
```

Exercise 4.84

A do-while loop only tests it condition after executing the statements in the loop's body at least once. The loop continues to execute while the condition is true, as with the while loop.

```
int i = 1;
do {
    System.out.print(i + " ");
    i++;
} while(i <= 10);
System.out.println();
```

Exercise 4.85

Solutions that use a do-while often end up including a duplicate of something close to the loop's condition inside the body of the loop, in order to avoid doing part of the

body one-too-many times. The do-while loop should be used sparingly, as it is rarely superior to iteration using a while loop.

```
/**
 * Show a list of all the files in the collection.
 */
public void listAllFiles()
{
    if(files.size() > 0) {
        // There is at least one file.
        int i = 0;
        do {
            System.out.println(files.get(i));
            i++;
        } while(i < files.size());
    }
}
```

Exercise 4.86

```
/**
 * Find the index of the first file matching the given
 * search string.
 * @param searchString The string to match.
 * @return The index of the first occurrence, or -1 if
 *         no match is found.
 */
public int findFirst(String searchString)
{
    int index = 0;
    // Record that we will be searching until a match is found.
    boolean searching = true;

    if(files.size() > 0) {
        // There is at least one to check.
        do {
            String filename = files.get(index);
            if(filename.contains(searchString)) {
                // A match. We can stop searching.
                searching = false;
            }
            else {
                // Move on.
                index++;
            }
        } while(searching && index < files.size());
    }
    if(searching) {
        // We didn't find it.
        return -1;
    }
    else {
        // Return where it was found.
        return index;
    }
}
```