CONCEPT EXERCISES

4.1 What is a collection? What is a Collection class? Give an example of a collection that is not an instance of a collection class. Programming Project 4.1 has an example of a collection class that is not a Collection class.

A ***collection*** is an object that is composed of elements.

A ***collection class*** is a class in which each instance of the class is a collection.

A ***Collection class*** is a class that implements the Collection interface.

An array is a collection, but is not an instance of a collection class.

PROGRAMMING EXERCISES

4.1 For each of the following, create and initialize a parameterized instance, add two elements to the instance, and then print out the instance:

a. an ArrayList object, scoreList, of Integer objects;

```
ArrayList<Integer> scoreList = new ArrayList<Integer>();

scoreList.add (25);
scoreList.add (18);
System.out.println (scoreList);
```

b. a LinkedList object, salaryList, of Double objects;

```
LinkedList<Double> salaryList = new LinkedList<Double>();

salaryList.add (3800.00);
salaryList.add (3400.25);
System.out.println (salaryList);
```

c. a TreeSet object, wordSet, of String objects;

```
TreeSet<String> wordSet = new TreeSet<String>();
```

```
wordSet.add (“relax”);
wordSet.add (“exhale”);
System.out.println (wordSet);
```

d. a HashSet object, employeeSet of FullTimeEmployee objects;

```
HashSet<FullTimeEmployee> employeeSet =
                        new HashSet<FullTimeEmployee>();

employeeSet.add (new FullTimeEmployee (“Smith”, 200.00));
employeeSet.add (new FullTimeEmployee (“Jones”, 410.00));
System.out.println (employeeSet);
```

e. a TreeMap object, studentMap, with name keys and grade-point average values;

```
TreeMap<String, Double> studentMap =
                        new TreeMap<String, Double>();

studentMap.put (“Smith”, 3.8);
studentMap.put (“Jones”, 3.5);
System.out.println (studentMap);
```

f. a HashMap object, tripMap, with flight-number keys and mileage values.

```
HashMap<Integer, Integer> tripMap =
                        new HashMap<Integer, Integer>();

tripMap.put (1374, 571);
tripMap.put (2771, 1033);
System.out.println (tripMap);
```

4.2 Develop a main method in which two ArrayList objects are created, one with String elements and one with Integer elements. For each list, add three elements to the list, remove the element at index 1, add an element at index 0, and print out the list.

```
public static void main (String[ ] args)
{
        ArrayList<String> wordList = new ArrayList<String>();
```

```
            wordList.add ("one");
            wordList.add ("two");
            wordList.add ("three");
            wordList.remove (1);
            wordList.add (0, "zero");
            System.out.println (wordList);

            ArrayList<Integer> scoreList = new ArrayList<Integer>();

            scoreList.add (1);
            scoreList.add (2);
            scoreList.add (3);
            scoreList.remove (1);
            scoreList.add (0, 0);
            System.out.println (scoreList);
        } // method main
```

4.4 Suppose we have the following:

```
            LinkedList<String> team = new LinkedList<String> ();
            team.add ("Garcia");
            Iterator<String> itr = team.iterator();
            Integer player = itr.next();
```

What error message will be generated? When (at compile-time or at run-time)? Test your hypotheses.

A compile-time error message is generated:

Incompatible types

Found: java.lang.String
Required: java.lang.Integer
Integer player = itr.next();

4.5 Use the ArrayList class three times. First, create an ArrayList object, team1, with elements of type String. Add three elements to team1. Second, create team2, another ArrayList object with elements of type String. Add four elements to team2. Finally, create an ArrayList object, league, whose elements are ArrayList objects in which each element is of type team<String>. Add team1 and team2 to league.

```
public static void main (String[ ] args)
{
          ArrayList<String> team1 = new ArrayList<String>();

          team1.add ("player1");
          team1.add ("player2");
          team1.add ("player3");

          ArrayList<String> team2 = new ArrayList<String>();

          team2.add ("player4");
          team2.add ("player5");
          team2.add ("player6");
          team2.add ("player7");

          ArrayList<ArrayList<String>> league = new
                                                  ArrayList<ArrayList<String>>();
          league.add (team1);
          league.add (team2);
          System.out.println (league);
} // method main
```

A run-time exception is generated (NoSuchElementException) because there are only three elements in wordSet – duplicates are not allowed – but the **for** statement calls itr.next() four times.

Chapter 5

CONCEPT EXERCISES

5.1 What is wrong with the following method for calculating factorials?

```
/**
 *  Calculates the factorial of a non-negative integer, that is, the product of
 *  all integers between 1 and the given integer, inclusive.
 *  The worstTime(n) is O(n), where n is the given integer.
 *
 *  @param n the non-negative integer whose factorial is calculated.
 *
 *  @return the factorial of n.
 *
 *  @throws IllegalArgumentException if n is less than 0.
 *
*/
public static long factorial (int n)
{
        if (n < 0)
                throw new IllegalArgumentException( );
        if (n <= 1)
                return 1;
        return fact (n+1) / (n+1);
} // fact
```

For n > 1, a simplest case will never be reached because the argument for each recursive call is n + 1.

5.8 If a call to the binarySearch method is successful, will the index returned always be the smallest index of an item equal to the key sought? Explain.

Not necessarily. Suppose the array contains

No, No, Yes

A search for “No” will return 1 (the index of the middle element), not 0.

PROGRAMMING EXERCISES

5.3 Given two positive integers *i* and *j*, the greatest common divisor of *i* and *j*, written

gcd (i, j)

is the largest integer *k* such that

(i % k = 0) and (j % k = 0).

For example, gcd (35, 21) = 7 and gcd (8, 15) = 1. Develop a recursive method that returns the greatest common divisor of *i* and *j*. Here is the method specification:

```
/**
 *  Finds the greatest common divisor of two given positive integers
 *
 *  @param i – one of the given positive integers.
 *  @param j – the other given positive integer.
 *
 *  @return the greatest common divisor of iand j.
 *
 *  @throws IllegalArgumentException – if either i or j is not a positive
integer.
 *
 */
public static int gcd (int i, int j)
```

Develop and run a test suite to test your method.

Big hint: According to Euclid's algorithm, the greatest common divisor of *i* and *j* is *j* if *i* % *j* = 0. Otherwise, the greatest common divisor of *i* and *j* is the greatest common divisor of *j* and *(i % j)*.

```
import org.junit.*;
import static org.junit.Assert.*;
import org.junit.runner.Result;
import static org.junit.runner.JUnitCore.runClasses;
import java.util.*;
import java.io.*;

public class GCDTest
```

```
{
    public static void main(String[ ] args)
    {
        Result result = runClasses (GCDTest.class);
        System.out.println ("Tests run = " + result.getRunCount() +
                            "\nTests failed = " + result.getFailures());
    } // method main

    int expected,
          actual;

    @Test
    public void test1()
    {
        expected = 7;
        actual = gcd (35, 21);
        assertEquals (expected, actual);
    } // test1

    @Test (expected = IllegalArgumentException.class)
    public void test2()
    {
        gcd (35, 0);
    } // test2

    @Test
    public void test3()
    {
        expected =1;
        actual = gcd (8, 15);
        assertEquals (expected, actual);
    } // test3

  /**
  *  Finds the greatest common divisor of two given positive integers
  *
  *  @param i – one of the given positive integers.
  *  @param j – the other given positive integer.
  *
  *  @return the greatest common divisor of iand j.
  *
  *  @throws IllegalArgumentException – if either i or j is not a positive
integer.
  *
  */
  public static int gcd (int i, int j)
  {
        if (i <= 0 || j <= 0)
           throw new IllegalArgumentException();
        return findGCD (i, j);
  } // method gcd

  public static int findGCD (int i, int j)
  {
        if (i % j == 0)
           return j;
        return findGCD (j, i % j);
  } // method findGCD

} // class GCDTest
```

5.4 A ***palindrome*** is a string that is the same from right-to-left as from left-to-right. For example, the following are palindromes:

ABADABA
RADAR
OTTO
MADAMIMADAM
EVE

For this exercise, we restrict each string to upper-case letters only. (You are asked to remove this restriction in the next exercise.)

Develop a method that uses recursion to test for palindromes. The only input is a string that is to be tested for palindromehood. The method specification is

```
/**
 *  Determines whether a given string of upper-case letters is a
 *  palindrome.  A palindrome is a string that is the same from right-to-
 *  left as from left-to-right.
 *
 *  @param s – the given string
 *
 *  @return true – if the string s is a palindrome. Otherwise, return
false.
 *
 *  @throws NullPointerException – if s is null.
 *  @throws IllegalArgumentException – if s is the empty string.
 *
 */
public static boolean isPalindrome (String s)
```

Develop and run a test suite to test your method.

```
import org.junit.*;
import static org.junit.Assert.*;
import org.junit.runner.Result;
import static org.junit.runner.JUnitCore.runClasses;
import java.util.*;
import java.io.*;

public class PalindromeTest
{
   public static void main(String[ ] args)
   {
      Result result = runClasses (PalindromeTest.class);
      System.out.println ("Tests run = " + result.getRunCount() +
                      "\nTests failed = " + result.getFailures());
   } // method main

   @Test
   public void test1()
   {
      assertEquals (true, isPalindrome ("RADAR"));
   } // test1

   @Test
   public void test2()
   {
      assertEquals (false, isPalindrome ("BANANA"));
   } // test2

   @Test
   public void test3()
   {
      assertEquals (true, isPalindrome ("T"));
   } // test3

   @Test
   public void test4()
```

```
{
   assertEquals (true, isPalindrome ("OTTO"));
} // test4

@Test
public void test5()
{
   assertEquals (false, isPalindrome ("OTOTTO"));
} // test5

@Test (expected = NullPointerException.class)
public void test6()
{
   isPalindrome (null);
} // test6

@Test (expected = IllegalArgumentException.class)
public void test7()
{
   isPalindrome ("");
} // test7

/**
 *  Determines whether a given string of upper-case letters is a
 *  palindrome.  A palindrome is a string that is the same from
 *  right-to-left as from left-to-right.
 *
 *  @param s – the given string
 *
 *  @return true – if the string s is a palindrome. Otherwise, return
 *                false.
 *
 *  @throws NullPointerException - if s is null.
 *  @throws IllegalArgumentException - if s is the empty string.
 *
 */
public boolean isPalindrome (String s)
{
     if (s.length() == 0)
          throw new IllegalArgumentException();
     return checkPalindrome (s);
} // method isPalindrome


/**
 *  Determines whether the substring of given string of upper-case
```

```
     *  letters is a palindrome. A palindrome is a string that is the same
     *  from right-to-left as from left-to-right.
     *
     *  @param s – the given string
     *
     *  @return true – if s is a palindrome.  Otherwise, return false.
     *
     */
    public boolean checkPalindrome (String s)
    {
        if (s.length() <= 1)
           return true;
        if (s.charAt (0) != s.charAt (s.length() - 1))
           return false;
        return checkPalindrome (s.substring (1, s.length() - 1));
    } // method checkPalindrome

} // class PalindromeTest
```

5.5 Extend the recursive method (and test class) developed in Programming Exercise 5.4 so that, in testing to see whether s is a palindrome, non-letters are ignored and no distinction is made between upper-case and lower-case letters. Throw IllegalArgumentException if s has no letters. For example, the following are palindromes:

Madam, I'm Adam.

Able was I 'ere I saw Elba.

A man. A plan. A canal. Panama!

Hint: The toUpperCase() method in the String class returns the upper-case String corresponding to the calling object.

```
import org.junit.*;
import static org.junit.Assert.*;
import org.junit.runner.Result;
import static org.junit.runner.JUnitCore.runClasses;
import java.util.*;
import java.io.*;

public class PalindromeTest2
{
```

```
public static void main(String[ ] args)
{
   Result result = runClasses (PalindromeTest2.class);
   System.out.println ("Tests run = " + result.getRunCount() +
                  "\nTests failed = " + result.getFailures());
} // method main

@Test
public void test1()
{
   assertEquals (true, isPalindrome ("Madam, I'm Adam."));
} // test1

@Test
public void test2()
{
   assertEquals (false, isPalindrome ("Madam, I am Adam."));
} // test2

@Test
public void test3()
{
   assertEquals (true, isPalindrome ("!@0o#$%"));
} // test3

@Test
public void test4()
{
   assertEquals (true, isPalindrome ("'Tis Ivan on a visit!"));
} // test4

@Test
public void test5()
{
   assertEquals (false, isPalindrome ("Toto"));
} // test5

@Test (expected = NullPointerException.class)
public void test6()
{
   isPalindrome (null);
} // test6

@Test (expected = IllegalArgumentException.class)
public void test7()
{
    isPalindrome ("!@#$%");
} // test7

/**
 *  Determines whether a given string of characters is a
 *  palindrome.  A palindrome is a string that is the same from
 *  right-to-left as from left-to-right.  Non-letters are ignored,
 *  and the method is case insensitive.
 *
 *  @param s – the given string
 *
 *  @return true – if the string s is a palindrome. Otherwise, return
 *                 false.
 *
 *  @throws NullPointerException - if s is null.
 *  @throws IllegalArgumentException - if s is has no letters.
 *
 */
public boolean isPalindrome (String s)
{
   String letters = new String();

   for (int i = 0; i < s.length(); i++)
      if (Character.isLetter (s.charAt (i)))
          letters += s.charAt (i);
   if (letters.length() == 0)
     throw new IllegalArgumentException();
   s = letters.toUpperCase();
   return checkPalindrome (s);
} // method isPalindrome

    /**
 *  Determines whether the substring of given string of upper-case
 *  letters is a palindrome. A palindrome is a string that is the same
 *  from right-to-left as from left-to-right.
 *
 *  @param s – the given string
 *
 *  @return true – if s is a palindrome.  Otherwise, return false.
 *
 */
public boolean checkPalindrome (String s)
{
     if (s.length() <= 1)
         return true;
     if (s.charAt (0) != s.charAt (s.length() - 1))
```

```
            return false;
        return checkPalindrome (s.substring (1, s.length() - 1));
    } // method checkPalindrome

} // class PalindromeTest2
```

5.7 Develop a recursive method to determine the number of distinct ways in which a given amount of money in cents can be changed into quarters, dimes, nickels, and pennies. For example, if the amount is 17 cents, then there are six ways to make change:

1 dime, 1 nickel and 2 pennies;
1 dime and 7 pennies;
3 nickels and 2 pennies;
2 nickels and 7 pennies;
1 nickel and 12 pennies;
17 pennies.

Here are some pairs. The first number in each pair is the amount, and the second number is the number of ways in which that amount can be changed into quarters, dimes, nickels and pennies:

17	6
5	2
10	4
25	13
42	31
61	73
99	213

Here is the method specification:

```
  /**
   *  Calculates the number of ways that a given amount can be changed
```

```
 *  into pennies, nickels, dimes and quarters.
 *
 *  @param amount – the given amount.
 *
 *  @return 0 – if amount is less than 0; otherwise, the number of ways
 *                 that amount can be changed into pennies, nickels, dimes
 *                 and quarters.
 *
 */
public static int ways (int amount)
```

Hint: Make the above method a wrapper for the following:

```
/**
 *  Calculates the number of ways that a given amount can be changed

 *  into coins whose values are no larger than a given denomination.
 *
 *  @param amount – the given amount.
 *  @param denomination – the given denomination (1 = penny,
 *                  2 = nickel, 3 = dime, 4 = quarter).
 *
 *  @return 0 – if amount is less than 0; otherwise, the number of ways
 *                 that amount can be changed into coins whose values are no
 *                 larger than denomination.
 *
 */
public static int ways (int amount, int denomination)
```

For the sake of simplifying the ways method, develop a coins method that returns the value of each denomination. Thus, coins (1) returns 1, coins (2) returns 5, coins (3) returns 10, and coins (4) returns 25.

Develop and run a test suite to test your ways method.

Hint: The number of ways that one can make change for an *amount* using coins no larger than a quarter is equal to the number of ways that one can make change for *amount* – 25 using coins no larger than a quarter plus the number of ways one can make change for *amount* using coins no larger than a dime.

```
import org.junit.*;
import static org.junit.Assert.*;
import org.junit.runner.Result;
import static org.junit.runner.JUnitCore.runClasses;
import java.util.*;
import java.io.*;

public class WaysTest
{
    public static void main(String[ ] args)
    {
        Result result = runClasses ( WaysTest.class);
        System.out.println ("Tests run = " + result.getRunCount() +
                            "\nTests failed = " + result.getFailures());
    } // method main

    protected static final int PENNY = 1;
    protected static final int NICKEL = 2;
    protected static final int DIME = 3;
    protected static final int QUARTER = 4;

    @Test
    public void test1()
    {
        assertEquals (6, ways (17));
    } // test1

    @Test
    public void test2()
    {
        assertEquals (2, ways (5));
    } // test2

    @Test
    public void test3()
    {
        assertEquals (4, ways (10));
    } // test3

    @Test
    public void test4()
    {
       assertEquals (13, ways (25));
    } // test4

    @Test
    public void test5()
    {
```

```
    assertEquals (31, ways (42));
 } // test5

 @Test
 public void test6()
 {
    assertEquals (73, ways (61));
 } // test6

 @Test
 public void test7()
 {
    assertEquals (213, ways (99));
 } // test7

 protected int ways (int amount)
 {
   return ways (amount, QUARTER);
 }

protected int ways (int amount, int denomination)
{
  if (amount < 0)
        return 0;
  if (denomination == PENNY)
        return 1;
  return ways (amount - coins (denomination), denomination) +
            ways (amount, denomination - 1);
} // method ways


/**
 *  For each denomination (1, 2, 3, 4), returns the value
 *  (1, 5, 10, 25) of that denomination.
 *
 */
protected int coins (int denomination)
{
  if (denomination == PENNY)
        return 1;
  if (denomination == NICKEL)
        return 5;
  if (denomination == DIME)
        return 10;
  return 25;
} // method coins
} // class  WaysTest
```