CPSC 211

SOLUTIONS to PRACTICE EXERCISES II

Recursion

1. The output of `cheers(4)` is:

```
Hip
Hip
Hip
Hurrah
```

2.

```
public static void cheers(int n)  {
        if ( n<=1)
          System.out.println("Hurrah");
        else {
          cheers(n-1);
          System.out.println("Hip ");
        }
}
```

3. a.

```
public static void cheers(int n)  {
        if ( n<=1)
          System.out.println("Hurrah");
        else {
          System.out.println("Hip ");
          cheers(n-1);
          System.out.println("Hip ");
        }
}
```

b.

```
public static void cheers(int n)  {
        if ( n<=1)
          System.out.println("Hurrah");
        else {
          System.out.println("Hip ");
          cheers(n-2);
          System.out.println("Hip ");
        }
```

4. Let's say that we define the size of the pattern to be $\log_2$ of the number of stars in the middle (and longest) line. In this case , the following is a list of the first three patterns:

size 0
```
*
```

size 1
```
*
* *
  *
```

size 2
```
*
* *
  *
* * * *
    *
    * *
      *
```

and the pattern shown in the question is of size 3. Moreover, sub-patterns of the same size are printed with different indentation. So, a pattern of size n and indentation d consists of:

- a pattern of size n-1 with indent d
- a line with 2^n asterisks with indent d
- a pattern of size n-1 with indent $d + 2^{n-1}$ spaces

Initially, the pattern will start with indent 0. Therefore, we need a recursive helper function with the two parameters (size and indent) and the original function with only one parameter, the pattern size.

```
public static void drawWeirdPattern( int size ) {

        drawWeirdPattern( size, 0 );
}


private static void drawWeirdPattern( int size, int indent ) {

        // base case
        if (size <= 0){
          drawLine(1, indent);
          return;
        }

        // recursive step
        drawWeirdPattern(size-1, indent);
        drawLine(power(2, size), indent);
        drawWeirdPattern(size-1, indent+power(2, size - 1));
}
private static void drawLine( int size, int indent ) {
```

```
            for (int i = 0; i < indent; i++)
               System.out.print(' ');
            for (int i = 0; i < size; i++)
               System.out.print('*');
            System.out.println();
      }

      private static int power(int base, int exponent) {
            int result = 1;
            for (int i = 1; i <= exponent; i++)
               result *= base;
            return result;
}
```

5. Here is a solution that works on lists of strings

```
      public static boolean findElt(List<String> sortedList, String elt){
            int midpoint = sortedList.size()/2;
            // base case
            if (elt.equals(sortedList.get(midpoint)))
                  return true;
            else if (sortedList.size()==1)
                  return false;
            else if (elt.compareTo(sortedList.get(midpoint)) <0)
                  // recursive step
                  return findElt(sortedList.subList(0,midpoint),elt);
            else
                  return
findElt(sortedList.subList(midpoint,sortedList.size()),elt);
      }
```

Or a more generic answer:

```
public static <E extends Comparable<? super E>> boolean findElt
      (List<E> sortedList, E elt) {
            int midpoint = sortedList.size()/2;
            // base case
            if (elt.equals(sortedList.get(midpoint)))
                  return true;
            else if (sortedList.size()==1)
                  return false;
            else if (elt.compareTo(sortedList.get(midpoint)) <0)
```

```
                  // recursive step
                  return findElt(sortedList.subList(0,midpoint),elt);
              else
                  return
findElt(sortedList.subList(midpoint,sortedList.size()),elt);
      }
```

More on Collections

1. The most efficient collection for this problem is a map which associates a breed with a set containing all the dogs of that breed that have been registered.

```
public class DogRegistry {

      // Declaration of the collection component

      private Map<String, Set<Dog>> dogMap;

      …

      public DogRegistry() {
         dogMap = new HashMap<String, Set<Dog>>();
      }

      public void addDog(Dog dog) {

         String breed = dog.getBreed();
         Set<Dog> dset = dogMap.get(breed);

         if ( dset == null ){
            dogMap.put( breed, new HashSet<Dog>() );
            dogMap.get(breed).add(dog);
         }
         else
            dset.add(dog);

      }


      public Set<Dog> getDogsOfBreed( String breed) {
         Set<Dog> copyOfBreeds = new Set<Dog>;
         copyOfBreeds.addAll(dogMap.get(breed));
         return copyOfBreeds;
      }
```

```
                …
        }
```

2.
```
public static <K,V> void removeValuesFromMap( Map<K,V> data, V item )
    {
          Set<K> keys = data.keySet();

          Iterator<K> itr = keys.iterator();

          while( itr.hasNext() )
          {
            K next = itr.next();
            if( data.get( next ).equals( item ) )
              itr.remove();
          }
}
```

3.
```
public static <K,V> Map<K,V> removeValuesInNewMap(Map<K,V> data, V item)
    {
            Map<K, V> newData = new HashMap<K, V>();

            Set<K> keys = data.keySet();
            Iterator<K> itr = keys.iterator();

            while(itr.hasNext())
            {
                    K next = itr.next();
                    V nextValue = data.get(next);
                    if (nextValue.equals(item) == false)
                            newData.put(next, nextValue);

            }
            return newData;
    }
```

Or, you can do it with a foreach loop:

```
public static <K,V> Map<K,V> removeValuesInNewMap(Map<K,V> data, V item)
        {
                Map<K, V> newData = new HashMap<K, V>();

                Set<K> keys = data.keySet();

                for (K next : keys)
                {
                        V nextValue = data.get(next);
                        if (nextValue.equals(item) == false)
                                newData.put(next, nextValue);
```

```
            }
            return newData;
    }
```

Streams

```
   1. public static void copyToNewFileSorted(String fromFilename,
                    String toFilename)
    {
         ArrayList<String> list = new ArrayList<String>();

         try {
               InputStream stream = new  FileInputStream(fromFilename);
               Reader reader = new InputStreamReader(stream);
               BufferedReader bufferedReader = new BufferedReader(reader);
               PrintWriter printWriter = new PrintWriter(toFilename);

               String line = bufferedReader.readLine();
               while (line != null)
               {
                      list.add(line);
                      line = bufferedReader.readLine();
               }

               Collections.sort(list);

               for(String word : list)
               {
                      printWriter.write(word + "\n");
               }

               printWriter.close();
               bufferedReader.close();
               reader.close();

         } catch (FileNotFoundException e) {
               System.out.println("Exception thrown, file " + fromFilename +
                                                          " not found");
               e.printStackTrace();
         } catch (IOException e) {
               System.out.println("IO exception thrown");
               e.printStackTrace();
         }
    }
```

Threads

1. The problem with the code is that the main thread doesn't necessarily wait until the other thread finishes before it begins printing the data. If we put a `join` statement for the sorting thread before printing the data then the main thread will wait until the sorting is done to print the values in the arrays.

Therefore we insert the following statements after the `x.start()` call (and before the print) in the main method:

```
try {
   x.join();
}
catch( InterruptedException e ) {}
```

2. The important fact to notice is that we don't really care what order the threads end in. We want the joinAll method to return as soon as all the threads terminate.

```
public void joinAll( Thread[] threads ) {
        for ( int i = 0; i < threads.length; i++ ) {
          try {
             threads[i].join();
          }
          catch( InterruptedException e ) {}
        }
}
```

Or, using foreach:

```
public void joinAll( Thread[] threads ) {
        for ( Thread t : threads ) {
          try {
             t.join();
          }
          catch( InterruptedException e ) {}
        }
}
```

3.

a) DEADLOCK

b)

- `thread na1 calls connect( other ) on m1`
- `na1 locks m1's lock preventing any other thread from executing critical sections of code in connect() or acknowledge() on m1.`
- `"Connecting Machine 1 to Machine 2" is printed on the console`
- `na1 is interrupted`
- `na2 calls connect( other ) on m2`
- `na2 locks m2's lock preventing any other thread from executing critical sections of code in connect() or acknowledge() on m2`

- "Connecting Machine 2 to Machine 1" is printed on the console
- na2 is interrupted
- na1 calls other.acknowledge() on m2 but cannot lock m2's lock because na2 holds the lock on m2, so na1 waits for the lock on m2
- na2 calls other.acknowledge() on m1 but cannot lock m1's lock because na1 holds the lock on m1, so na2 waits for the lock on m1
- we're now in a deadlock situation as each thread is waiting for the other to release a lock