

CPSC 211

SOLUTIONS to PRACTICE EXERCISES II

Last Updated: 11/29/2010 7:57 PM

Recursion

1. The output of `cheers(3)` is:

```
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.

```
public static boolean isPalindrome(String string) {  
    int length = string.length();  
    if (length < 2) return true;  
    if (string.charAt(0) != string.charAt(length-1))  
        return false;  
    return isPalindrome(string.substring(1,length-1));  
}
```

5. 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:

| <u>size 0</u> | <u>size 1</u> | <u>size 2</u> |
|---------------|---------------|---------------|
| * | * | * |
| | * * | * * |
| | * | * |
| | | * * * * |
| | | * |
| | | * * |
| | | * |

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 $d+n$ 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+size);
}

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;
}
}

```

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

    Map<String, Set<Dog>> 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) {
        return dogMap.get(breed);
    }

    ...
}
```

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();
    }
}
```

Stacks & Queues

1.

a. Stack s is:

E ← top
A

b. Stack s is empty. The sequence will throw an `EmptyStackException`

2. We can try:

```
public static <E> void reverse( Stack<E> stack ) {  
  
    Queue<E> queue = new LinkedList<E>();  
  
    while (!stack.empty())  
        queue.offer(stack.pop());  
  
    E obj = queue.poll();  
    while ( obj != null ){  
        stack.push(obj);  
        obj = queue.poll();  
    }  
}
```

This won't work if the stack contains null items. The following is a better solution:

```
public static <E> void reverse( Stack<E> stack ) {  
  
    Queue<E> queue = new LinkedList<E>();  
  
    while (!stack.empty())  
        queue.offer(stack.pop());  
  
    while (!queue.empty() ){  
        stack.push(queue.poll());  
    }  
}
```

Streams

```
1. public static void getLines(String file, String key) {
    try {
        BufferedReader in = new BufferedReader(new FileReader(file));
        String line = in.readLine();

        while ( line != null ) {
            if ( line.contains(key)){
                System.out.println(line);
            }
            line = in.readLine();
        }
    }
    catch ( Exception e ){
        System.out.println("Error in reading " + file);
    }
}
```

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 after the `x.start()` then the main thread will wait until the sorting is done to print the values in the array. Therefore the last two statements of the main method should be like the following:

```
x.start();

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

for (int i = 0; i < data.length ; i++)
    System.out.println( data[i] );
}
}
```

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 ) {}
    }
}
```

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