# COMPUTER PROGRAMMING – ETSINF – THEORY – Academic year 2016/2017
## Retake second mid term exam – June 26, 2017 – Duration 2 hours

**Notice:** The maximum mark of this exam is 10, but his weight in the final grade is **3 points**.

1. $\boxed{\text{2.5 points}}$ Given a text file whose contents are lines with different tokens per line, the tokens in each line are separated by white spaces, and some of the tokens are valid integers, but not all of them.

   **You have to implement** an static method with one parameter, an object of the class `String` containing the file name. The file name includes the relative or absolute path to the file in the file system, because it could be possible the file is not contained in the current working directory.

   The method should return the sum of all the valid integers contained in the file. More specifically, the method should:

   - Catch exceptions of the class `FileNotFoundException` which can be thrown when the methods tries to open the file. In the catch block, an error message indicating that the file can not be opened should be shown on screen.
   - Read all the lines in the file by using the method `nextLine()` of the class `Scanner`.
   - Split each line by using the method `split(expReg)` of the class `String`. Thanks to this method, an array of objects of the class `String` is obtained, where each object of the class `String` contains one token. The value for `expReg` must be only one white space, i.e. an object of the class `String` with just one white space `" "`. So the call should be `split( " " )`.
   - Parse each token for converting it into an integer by using the method `Integer.parseInt(String)`.
   - If the token is a valid representation of an integer, the value should be added to the sum.
   - Otherwise, an exception of the class `NumberFormatException` will be thrown and your code must catch it. In such case, the token that does not correspond to a valid integer should be shown on screen and your method should continue processing tokens until the end of the file is reached.

**Solution:**

```
public static int readAndAdd( String filename )
{
    int sum = 0;
    Scanner s = null;
    try {
        s = new Scanner( new File( filename ) );
        while( s.hasNextLine() ) {
            String [] tokens = s.nextLine().trim().split( " " );
            for( int i = 0; i < tokens.length; i++ ) {
                try {
                    sum += Integer.parseInt( tokens[i] );
                }
                catch( NumberFormatException e ) {
                    System.out.println( tokens[i] );
                }
            }
        }
    }
    catch( FileNotFoundException e ) {
        System.out.println( "File not found!" );
    }
    finally {
        if ( s != null ) { s.close(); }
    }
    return sum;
}
```

2. $\boxed{\text{2.5 points}}$ The following class is available as an implementation of a linked sequence whose nodes are objects of the class `NodeInt`:

```
public class LinkedSequence {
    private NodeInt first;  // reference to the first element of the sequence.
    . . .
}
```

**You have to implement** a non-static method in this class for finding the first element of an integer `x` given as parameter. If an occurrence of `x` is found, then it should be advanced one position in the linked sequence. However, if an occurrence of `x` is already in the first position of the linked sequence, then it should be moved to the last position and the last value moved to the first one. In the case that `x` does not exist in the linked sequence, then the method does not perform any action.

As an example, given the sequence `[3, 7, 2, 8, 5]`,

- after the call `advance(8)` the sequence will remain as `[3, 7, 8, 2, 5]` and
- after a second call `advance(3)` then the sequence will be changed as `[5, 7, 8, 2, 3]`.

**Solution:**

```
/** Precondition: the sequence contains 2 elements at least */
public void advance( int x )
{
    NodeInt previous = null, current = first;

    while( current != null  &&  current.getValue() != x ) {
        previous = current;
        current = current.getNext();
    }
    if ( current != null ) {
        if ( previous == null ) {
            previous = current;
            while( previous.getNext() != null ) {
                previous = previous.getNext();
            }
        }
        current.setValue( previous.getValue() );
        previous.setValue( x );
    }
}
```

3. 2.5 points **You have to implement** a non-static method in the class `ListIntLinked` with the following profile

```
public void insertAt( int x, boolean start )
```

that, given an integer `x` inserts it at the beginning of the list if the parameter `start` is `true`, otherwise `x` is inserted after the last one. When this method ends the cursor (interest point) should reference the last inserted element.

**Notice:** You only can work with internal attributes of the class `ListIntLinked`. You are not allowed to use the methods.

**Solution:**

```
public void insertAt( int x, boolean start )
{
    NodeInt newNode = new NodeInt(x);

    if ( first == null ) {
        first = last = newNode;
    } else if ( start ) {
        newNode.setNext( first );
        first.setPrevious( newNode );
        first = newNode;
    } else {
        newNode.setPrevious( last );
        last.setNext( newNode );
        last = newNode;
    }
    cursor = newNode;
    ++size;
}
```

4. 2.5 points **You have to implement:** an static method in a class different from class `QueueIntLinked` that given an object of the class `QueueIntLinked` with values in the range [0..9] returns an integer value which corresponds to the concatenation of the digits contained in the queue.

The queue should remain untouched after the execution of the method.

As an example, if the contents of the queue is ← `5 1 4 7` ←, the returned integer value should be 5147.

Notice that this integer value is computed as `((((5 * 10) + 1) * 10) + 4) * 10) + 7`.

**Solution:**

```
/** Version 1: without the use of auxiliary structures */
public static int fromQueueToInt( QueueIntLinked q )
{
    int value = 0;
    for( int i = 0; i < q.size(); i++ ) {
        int x = q.dequeue();
        value = value * 10 + x;
        q.enqueue(x);
    }
    return value;
}

/** Version 2: using an auxiliary queue */
public static int fromQueueToInt( QueueIntLinked q )
{
    int value = 0;
    QueueIntLinked qAux = new QueueIntLinked();
    while( !q.isEmpty() ) {
        int x = q.dequeue();
        value = value * 10 + x;
        qAux.enqueue(x);
    }
    while( !qAux.isEmpty() ) q.enqueue( qAux.dequeue() );

    return value;
}
```