COMPUTER PROGRAMMING – ETSINF – ACADEMIC YEAR 2016/2017
Second mid term exam of theory – June 5th, 2017 – duration 2 hours

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

1. 2.5 points It is available a text file for which it is assumed that each line contains a single numerical value of type `double`. The decimal separator is the point, not the comma.

   Nevertheless, no all the lines in the text file contain a string which corresponds to a valid real number. Those lines should be rejected. The goal of this problem is to create a new text file with just the correct lines. Each line in the new file should contain a unique real value, i.e. of type `double`.

   The new file should be created in the same directory where the original one is located, and the name of the new file will be composed by adding the suffix "`_new`" to the name of the original text file.

   For instance, if the name of the original file is "`../../numbers.txt`" then the name of the new one should be "`../../numbers.txt_new`".

   If the original file does not exist or can not be accessible by the current user who runs the program, no warning or error messages should appear on screen. If it exists and one or more lines should be rejected because they do not contain a valid real number, no warning or error messages should be shown on screen, and all the valid lines should be processed.

   What to do: Write a method with one parameter containing the name of the original text file as an object of the class `String` for performing the task explained above.

   NOTICE: You should take into account some exceptions can be thrown due to input/output operations, for instance `FileNotFoundException`. Additionally, you should also take into account that they can be thrown the exceptions `InputMismatchException` and `NumberFormatException` when reading numeric values. The last one in the case you use the method `Double.parseDouble(String)`.

Solution:

```
// Alternative solution 1
public static void filterNonRealNumbers( String filename )
{
    File f = new File( filename );
    Scanner s = null;
    PrintWriter p = null;
    try {
        s = new Scanner(f).useLocale(Locale.US);
        p = new PrintWriter( filename + "_new" );
        while( s.hasNextLine() ) {
            try {
                String lA = s.nextLine().trim();
                p.println( Double.parseDouble(lA) );
            }
            catch( NumberFormatException n )
            {
            }
        }
    }
    catch( FileNotFoundException e ) { }
    finally {
        if ( s != null ) { s.close(); }
        if ( p != null ) { p.close(); }
    }
}
```

```
// Alternative solution 2
public static void filterNonRealNumbers( String filename )
{
    File f = new File( filename );
    Scanner s = null;
    PrintWriter p = null;
    try {
        s = new Scanner(f).useLocale(Locale.US);
        p = new PrintWriter( filename + "_new" );
        while( s.hasNextLine() ) {
            try {
                p.println( s.nextDouble() );
            }
            catch( InputMismatchException e ) { }
            finally { s.nextLine(); }
        }
    }
    catch (FileNotFoundException e) { }
    finally {
        if ( s != null ) { s.close(); }
        if ( p != null ) { p.close(); }
    }
}
```

2. 2.5 points **What to do:** Implement a new constructor in the class `StackIntLinked` for creating a new stack that is a copy of the stack received as parameter. Then, after executing the following line:

```
StackIntLinked newStack = new StackIntLinked( StackIntLinked s );
```

it will be possible to make the operations *push* and *pop* in one of both stacks without making changes in the other.

NOTICE: The implementation **should** use the attributes of the class `StackIntLinked`. It is **not** allowed to use the methods of this class.

Solution:

```
public StackIntLinked( StackIntLinked s )
{
    this();
    if ( ! s.isEmpty() ) {
        NodeInt p = this.top = new NodeInt( s.top.getValue() );
        NodeInt n = s.top.getNext();
        while( null != n ) {
            p.setNext( new NodeInt( n.getValue() ) );
            p = p.getNext();
            n = n.getNext();
        }
    }
    this.size = s.size;
}
```

3. 2.5 points **What to do:** Add the method `equals(Object)` into the class `ListIntLinked` to override the one implemented in the class `Object`.

   Given two objects of the class `ListIntLinked`, they are considered to be equal if and only if:

   1. they have the same elements located in the same position, and
   2. their cursor is referencing to the same element, i.e. to the same position.

   What to do: Complete the following code in order to return `true` when comparing two lists according to the above mentioned criteria.

```
/** Returns true if the list referenced by 'o' contains the same values
    contained in the current one. */
public boolean equals( Object o )
{
    boolean rc = false;

    if ( o instanceof ListIntLinked ) {

        ListIntLinked other = (ListIntLinked) o;

        // COMPLETE THE CODE to return if this and other are equal
}
```

   NOTICE: The position of the cursor in both lists must be untouched after the comparison.

Solution:

```
/** Returns true if the list referenced by 'o' contains the same values
    contained in the current one. */
public boolean equals( Object o )
{
    boolean rc = false;

    if ( o instanceof ListIntLinked ) {

        ListIntLinked other = (ListIntLinked) o;

        NodeInt n1 = this.first, n2 = other.first;

        rc = ( this.size == other.size );

        while( true == rc  &&  n1 != null  &&  n2 != null ) {

            rc = ( n1.getValue() == n2.getValue()
               && !( n1 == this.cursor ^ n2 == other.cursor) );

            n1 = n1.getNext();
            n2 = n2.getNext();
        }
    }
    return rc;
}
```

4. 2.5 points In order to avoid codification problems during the transfer of a text file, all 'ñ' have been substituted by '~n', a subsequence of two characters, and analogously all 'Ñ' have been substituted by '~N'.

It is available the class `ListCharLinked` which implements a list with interest point whose elements are characters, i.e. variables of type `char`. So, any text can be stored by using an object of this class, where all the characters (letters and no letters) of the text will be in a linked sequence.

This class has available the following methods which are similar to the ones you studied in class `ListIntLinked` but adapted for working with characters instead of integers.

```
public void begin()              public void next()
public void insert( char ch )    public char remove()  or  public void remove()
public char get()                public int size()
public boolean isEmpty()         public boolean isValid()
```

What to do: Implement an static method outside the class `ListCharLinked` that given an object of this class makes the proper operations in order to reconstruct all the occurrences of '`ñ`' and '`Ñ`' that in the text (linked sequence) appear as subsequences of two characters as explained above.

As an example, if a line of a text contains "`Juan Nú~nez de la Pe~na, ESPA~NA`", the contents in the object of class `ListCharLinked` should be modified to leave the same line as "`Juan Núñez de la Peña, ESPAÑA`".

NOTICE: You can assume that character '`~`' will always be followed by an '`n`' or an '`N`'.

Solution:

```
public static void reconstructSpanishText( ListCharLinked l )
{
    l.begin();
    while( l.isValid() ) {
        char ch = l.get();
        if ( ch == '~' ) {
            l.remove();
            ch = l.remove();
            l.insert( (ch == 'n') ? 'ñ' : 'Ñ' );
        } else {
            l.next();
        }
    }
}
```

If you use the version of `remove()` that does not return the removed character, then the solution is:

```
public static void reconstructSpanishText( ListCharLinked l )
{
    l.begin();
    while( l.isValid() ) {
        char ch = l.get();
        if ( ch == '~' ) {
            l.remove();
            ch = l.get();
            l.remove();
            l.insert( (ch == 'n') ? 'ñ' : 'Ñ' );
        } else {
            l.next();
        }
    }
}
```