Problem 1

Write a program which defines a very simple class **Person** (as shown below) and then, in class **StreamNames**, defines a static function **getPersons** which takes a list of **String**s and returns a list of **Person**s

- with names taken from the input list of **String**s;
- with only names of length greater than 3;
- with names modified so they all start with a capital letter followed by lowercase letters;
- is sorted alphabetically by names;
- contains only the first three objects obtained.

Implementation of the function *must* consist of *exactly one* statement: **return** statement.

For example, the following program

download *StreamNames.java*

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

  // should be in a separate file
class Person {
    private String name;
    Person(String n) { name = n; }
    @Override
    public String toString() { return name; }
}

public class StreamNames {
    public static void main (String[] args) {
       List<String> list = Arrays.asList(
               "john", "al", "KENNY", "jenny","noemi");
       System.out.println(getPersons(list));
    }

    static List<Person> getPersons(List<String> list) {
        // ...
    }
}
```

should print

```
[Jenny, John, Kenny]
```

No other **import**s should be needed.

Problem 2

Write a program according to the following scheme

download *ByteThreads.java*

```
import java.io.InputStream;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

public class ByteThreads extends Thread {
    final static String iFileName = "Majas.jpg";

    int countAll = 0;   // non-static
    int counter = 256;  // non-static

    public static void main(String[] args) {
        new ByteThreads().start();
    }

    ByteThreads() {
        for (int i = 0; i < 256; ++i)
            new Thread(new ByteReader(i, this)).start();
    }

    @Override
    public void run() {
        // ...
    }
}
```

In the **main** function, we create *one* object of type **ByteThreads** and, as it itself represents a thread, we start it.

In its constructor, we create and start 256 threads (what is not very wise...) passing to their constructors objects of our class **ByteRedaer** implementing the **Runnable** interface: to its constructor we pass consecutive values from the range $[0, 255]$ and the reference to the object of type **ByteThreads** being created.

Each of these threads opens the same file (its name can be given as a static field of the **ByteThreads** class) and searches for bytes equal to the value passed to the constructor. Each time it finds one, it increments the counter countAll in the **ByteThreads** object passed to its constructor, the same for all threads (what is highly ineffective!) It also counts the bytes found and after having read the whole file prints the result, decrements the counter of the active threads from the **ByteThreads** class and terminates.

The thread represented by the **ByteThreads** object checks from time to time (for example, every 200 ms) if the counter of active threads reached zero — if so, it checks

the length of the file and prints it together with the vaue of the countAll counter — these two values should be the same.

The printout of the program could look like this

```
Thread for 144 completed with count = 369
Thread for 103 completed with count = 434
Thread for 158 completed with count = 394
Thread for 156 completed with count = 522
Thread for 162 completed with count = 321
      ...  many lines omitted  ...
Thread for 243 completed with count = 342
Thread for 253 completed with count = 380
Thread for 244 completed with count = 419
Thread for 245 completed with count = 459
Thread for 255 completed with count = 360
Thread for 246 completed with count = 401
Thread for 247 completed with count = 417
Thread for 252 completed with count = 337
Thread for 251 completed with count = 386
countAll  : 99280
Files.size: 99280
```

You can get the length of a file, in bytes, as a **long**, using

```
Files.size(Paths.get(iFileName));
```
