Programmazione 15/01/2019

Soluzione Esercizio 1

```
 0 // MaxFileProcessor.java
 1 import java.io.*;
 2
 3 public class MaxFileProcessor extends FileProcessor {
 4
 5   /* Richiama costruttore della superclasse */
 6   public MaxFileProcessor(String f) throws FileNotFoundException, IOException {
 7     super(f);
 8   }
 9
10   /* Trova il massimo nel blocco */
11   protected byte processBlock(byte[] b) {
12     byte x = b[0];
13     for(int i=1; i<b.length; i++)
14       if(b[i] > x)
15         x = b[i];
16     return x;
17   }
18 }
19
```

```
  0 // FileProcessor.java
  1 import java.io.*;
  2
  3 public abstract class FileProcessor {
  4
  5     /* Nome del file contenente i dati di ingresso */
  6     private String filenameInput;
  7     /* Numero di flussi che devono ancora terminare */
  8     private int toComplete;
  9     /* Flussi ausiliari */
 10     private Worker w1;
 11     private Worker w2;
 12     /* Numero di blocchi */
 13     private long numberOfBlocks;
 14     /* Lunghezza del file in byte */
 15     private long fileLength;
 16     /* Array che contiene i risultati */
 17     private byte[] res;
 18
 19     /* Crea un FileProcessor che lavora sul file indicato come argomento */
 20     public FileProcessor(String filenameInput) throws FileNotFoundException, IOException {
 21         this.filenameInput = filenameInput;
 22         // Calcola la lunghezza del file
 23         RandomAccessFile raf = new RandomAccessFile(filenameInput, "r");
 24         fileLength = raf.length();
 25         raf.close();
 26         // Numero di flussi che devono ancora terminare
 27         this.toComplete = 2;
 28     }
 29
 30     public synchronized byte[] process(int blocksize) throws InterruptedException,
IOException {
 31       // Controlla che la dimensione sia un multiplo della dimensione del blocco
 32       boolean sizeOK = fileLength % blocksize == 0;
 33       if(!sizeOK) throw new IllegalArgumentException();
 34       // Calcola il numero di blocchi
 35       numberOfBlocks = fileLength / blocksize;
 36       // Contenitore dei risultati. Suppongo che un array sia sufficiente (in teoria il
 37       // file potrebbe essere cosi' grande da produrre un numero di risultati maggiore
 38       // della dimensione massima degli array Java).
```

```
39        res = new byte[(int)numberOfBlocks];
40        // Attiva due flussi: uno lavora a partire dalla testa, l'altro dalla coda.
41        // Il valore booleano passato come secondo argomento determina la direzione.
42        w1 = new Worker(filenameInput, true, blocksize);
43        w1.start();
44        w2 = new Worker(filenameInput, false, blocksize);
45        w2.start();
46        // Attende che i due flussi abbiano finito
47        waitForComplete();
48        // Restituisce il risultato
49        return res;
50      }
51
52      /* Attende che tutti i flussi abbiano finito */
53      private void waitForComplete() throws InterruptedException {
54        while(toComplete > 0)
55          wait();
56      }
57
58      /* Chiamato alla fine di ogni flusso ausiliario per indicare che ha
59         completato i suoi lavori */
60      private synchronized void completed() {
61        toComplete--;
62        if(toComplete == 0)
63            notify();
64      }
65
66      /* Chiamato dai flussi ogni volta che terminano di processare un blocco.
67         Restituisce true se ci sono altri blocchi da processare, false altrimenti */
68      private synchronized boolean blocksAvailable() {
69        numberOfBlocks--;
70        return numberOfBlocks >= 0;
71      }
72
73      /* Implementato dalle sottoclassi, defiisce il tipo di elaborazione da compiere */
74      protected abstract byte processBlock(byte[] block);
75
76      /* Flusso di esecuzione */
77      class Worker extends Thread {
78        // true: parte dalla testa false: parte dalla coda
79        private boolean direction;
80        // Dimensione dei blocchi
81        private int blocksize;
82        // Usato per spostarsi nel file
83        private RandomAccessFile r;
84
85        // Crea un flusso ausiliario. Gli argomenti indicano: il file da cui
86        // leggere, la direzione, la dimensione dei blocchi
87        Worker(String filenameInput, boolean direction, int blocksize) throws
FileNotFoundException, IOException{
88          this.direction = direction;
89          this.blocksize = blocksize;
90          r = new RandomAccessFile(filenameInput, "r");
91          if(!direction) {
92            // Se parte dalla coda sposta il puntatore del file all'inizio
93            // dell'ultimo blocco
94            r.seek(r.length()-blocksize);
95          }
96        }
97
98        // Legge un nuovo blocco
99        void readBlock(byte[] buf) throws IOException {
100         r.readFully(buf);
101         if(!direction){
102           // Per il flusso che va al contrario e' necessario riposizionare il
103           // puntatore del file all'inzio del blocco precedente. Bisogna sottrarre
104           // due volte la dimensione del blocco perche' ogni lettura di lo riporta
105           // in avanti di blocksize.
106           long pointer = r.getFilePointer();
```

```
107           pointer = pointer - 2*blocksize;
108           r.seek(pointer);
109         }
110       }
111
112       public void run() {
113         try {
114           // Buffer per i dati del prossimo blocco
115           byte[] b = new byte[blocksize];
116           // Risultato
117           byte c;
118           // Se ci sono blocchi da processare...
119           while(blocksAvailable()) {
120             // Calcola l'indice del blocco e quindi anche del risultato
121             int index = (int)(r.getFilePointer() / blocksize);
122             // Legge il prossimo blocco
123             readBlock(b);
124             // Elabora il blocco e produce il risultato
125             c = processBlock(b);
126             // Scrive il risultato del blocco nel contenitore dei risultati globali
127             res[index] = c;
128           }
129           // Il flusso ha terminato
130           completed();
131           // Chiude il file
132           r.close();
133         } catch (IOException ioe) {
134           System.out.println(ioe.getMessage());
135         }
136       }
137     }
138 }
139
```

Soluzione Esercizio 2

```
1  // Parametro.java
2  import com.thoughtworks.xstream.*;
3
4  public class Parametro {
5    private boolean stabilitaCampioni;
6    private double sommaCampioni;
7    private int numeroCampioni;
8
9    public Parametro(double c) {
10     sommaCampioni = c; numeroCampioni = 1; stabilitaCampioni = false;
11   }
12
13   public Parametro(String xml) {
14     Parametro p = (Parametro)(new XStream()).fromXML(xml);
15     numeroCampioni = p.numeroCampioni;
16     stabilitaCampioni = p.stabilitaCampioni;
17     sommaCampioni = p.sommaCampioni;
18   }
19
20   public void aggiungiCampione (double c) {
21     if ((Math.round(sommaCampioni/numeroCampioni) !=
22          Math.round((sommaCampioni+c)/(numeroCampioni+1)))
23     ) { sommaCampioni+=c; numeroCampioni++; }
24     else
25       stabilitaCampioni = true;
26   }
27
```

```
28   public boolean stabilitaCampioni() {
29     return stabilitaCampioni;
30   }
31
32   public double calcolaParametro() {
33     return Math.round(sommaCampioni/numeroCampioni);
34   }
35
36   public String toString() {
37     return (new XStream()).toXML(this);
38   }
39 }
40
```

```
1  // Terminale.java
2  import java.net.*;
3  import java.io.*;
4
5  public class Terminale {
6
7    public static void inviaComeStringa(int porta, Parametro p) {
8      try (Socket sock = new Socket("localhost", porta);
9        DataOutputStream dos =
10         new DataOutputStream(sock.getOutputStream());
11     ) {  dos.writeUTF(p.toString());
12     } catch (IOException e) { e.printStackTrace();}
13     System.out.println("- invio a " + porta); //2
14   }
15
16   public static Parametro riceviComeStringa(int porta) {
17     Parametro p = null;
18     try (
19       ServerSocket servsock = new ServerSocket(porta);
20       Socket sock = servsock.accept();
21       DataInputStream dis =
22          new DataInputStream(sock.getInputStream());
23     ) {
24       p = new Parametro(dis.readUTF());
25     } catch (IOException e) { e.printStackTrace();}
26     System.out.println("- ricevo\n" + p); //2
27     return p;
28   }
29
30   public static void main(String[] args) {
31     System.out.println("- sono " + args[1]); //2
32     Parametro p;
33     if (args[1].equals("8080")) { // è la radice
34       p = new Parametro(Double.parseDouble(args[0])); //1
35       System.out.println(p); //2
36       inviaComeStringa(Integer.parseInt(args[2]), p); //3
37       p = riceviComeStringa(Integer.parseInt(args[1])); //6
38       System.out.println(p.calcolaParametro()); //7
39     }
40     else {  //non è la radice
41       p = riceviComeStringa(Integer.parseInt(args[1])); //4
42       if (p.stabilitaCampioni()) {  //5.a
43         inviaComeStringa(Integer.parseInt(args[2]), p); //5.a.1
44       } else { //5.b
45         p.aggiungiCampione(Double.parseDouble(args[0])); //5.b.1
46         System.out.println("- aggiunto campione\n" + p); //5.b.2
```

```
47          if (args.length < 4) { //5.b.3.c foglia
48            inviaComeStringa(Integer.parseInt(args[2]), p); //5.b.3.c.1
49          } else { // 5.b.3.d nodo
50            if (!p.stabilitaCampioni()) { //5.b.3.d.e nodo e campioni instabili
51              inviaComeStringa(Integer.parseInt(args[3]), p); //5.b.3.d.e.1
52              p = riceviComeStringa(Integer.parseInt(args[1])); //5.b.3.d.e.2
53            }
54            inviaComeStringa(Integer.parseInt(args[2]), p); //  //5.b.3.d.1
55          }
56        }
57      }
58    }
59 }
60
```