Lab Session 1-Report

Concurrency and Parallelism

* Submitted by

Cyril Naves Samuel

# Exercise 1: The *Thread* class

The goal of this very first exercise is to get used to the usage of the *Thread* class in Java (*java.lang*).

1/ Write a program where two threads print concurrently on the standard output (*System.out*) a different message 10000 times each.

2/ After launching the threads, the main will also print 10000 times a message.

**Summary for the Thread Class implementation:**

1. Main Class is extending Thread Class from the java.lang package.
2. Thread Class implements Runnable Interface and extends Object Class
3. Here two New Threads are created and then started to print different messages for 10000 times
4. Also the main method has a provided logic to print the text 10000 times

**Code:**

**package** masterInt.CandP.exo1;

**public** **class** Exo1 **extends** Thread {

String message;

/\*\*

\* Constructor

\*

\* **@param** \_message

\* The message to print

\*/

**public** Exo1(String inpmessage) {

message = inpmessage;

}

@Override

**public** **void** run() {

// Print the message 10000 times

**for** (**int** i = 0; i < 10000; i++) {

System.***out***.println(message + "no:" + (i + 1));

}

}

/\*\*

\* Main method to print concurrent messages through two different threads

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

Exo1 exo1 = **new** Exo1("Thread1Printing");

exo1.start();

Exo1 exo2 = **new** Exo1("Thread2Printing");

exo2.start();

**for** (**int** i = 0; i < 10000; i++) {

System.***out***.println("main" + "no:" + (i + 1));

}

}

}

**Sample Output: Since all the 30000 printout are not captured due to space constraint**

Thread2Printingno:9976

Thread2Printingno:9977

Thread2Printingno:9978

Thread1Printingno:9979

Thread1Printingno:9980

Thread2Printingno:9981

# Exercise 2: The *Runnable* interface

Write a program that prints a frame (*javax.swing.JFrame*) containing two text fields (*javax.swing.JTextField*) and two buttons (*javax.swing.JButton*). Each button is a runnable entity that is started when one clicks on it (*java.awt.event.ActionListener* and addActionListener()).

When started, it prints the content of the associated text field on the standard output every 500 ms (*Thread.sleep()*). If the text in the field changes, the printed message changes accordingly. Of course, both buttons can be activated simultaneously.

When clicking one more time on the button, it stops printing, then resumes, and so on. When the frame is closed, the two runnable entities are killed!

**Summary for the Runnable Interface implementation:**

1. Here Runnable Button Class extends JButton Class as well as implements Runnable Interface to get the properties of swing as well as Concurrent Execution of Threads
2. Run method is overridden to print the value in the Jtext field value passed through constructor
3. Exo2 has the button, Text field in the current window frame and awaits the action listener
4. When the button is clicked the button threads are then started .
5. Conditional checks are implemented in the action listener to handle the stop as well as resume the thread printing operation
6. Finally in window closing method the two threads are then killed.

**Code:**

**package** masterInt.CandP.exo2;

**import** javax.swing.JButton;

**import** javax.swing.JTextField;

**public** **class** RunnableButton **extends** JButton **implements** Runnable {

// Please ignore that

**private** **static** **final** **long** ***serialVersionUID*** = 7453535863156182464L;

JTextField textfield;

**public** RunnableButton(String text, JTextField \_tf) {

**super**(text);

**this**.textfield = \_tf;

}

@Override

**public** **void** run() {

// Print the content of the text field on the output stream

**while** (**true**) {

System.***out***.println(textfield.getText());

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

package masterInt.CandP.exo2;

import java.awt.FlowLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.WindowEvent;

import java.awt.event.WindowListener;

import javax.swing.JFrame;

import javax.swing.JTextField;

public class Exo2 extends JFrame implements ActionListener, WindowListener {

private static final long serialVersionUID = -2818867002346363736L;

private RunnableButton button1, button2;

private Thread thread1, thread2;

private String textButton1, textButton2;

public Exo2(String title) {

// Set up the frame

super(title);

this.setDefaultCloseOperation(DO\_NOTHING\_ON\_CLOSE);

this.addWindowListener(this);

textButton1 = "button1";

textButton2 = "button2";

this.setSize(300, 120);

this.setLayout(new FlowLayout());

JTextField tf1 = new JTextField("Text1", 10);

JTextField tf2 = new JTextField("Text2", 10);

// Set up button1

button1 = new RunnableButton("Text1", tf1);

button1.addActionListener(this);

// Set up button 2

button2 = new RunnableButton("Text2", tf2);

button2.addActionListener(this);

// Add button 1 and 2 to the frame

this.add(tf1);

this.add(button1);

this.add(tf2);

this.add(button2);

}

// This method is called when any of the button is clicked

@SuppressWarnings("deprecation")

// At the first click, the corresponding thread is started

// At the subsequent, the state of the thread is changed

// from active to inactive or vice versa

@Override

public void actionPerformed(ActionEvent e) {

if (e.getSource() instanceof RunnableButton) {

RunnableButton button = (RunnableButton) e.getSource();

// Identify the button

if (thread1 == null && button.getText() == "Text1") {

thread1 = new Thread(button1);

thread1.start();

try {

Thread.sleep(500);

} catch (InterruptedException e1) {

// TODO Auto-generated catch block

e1.printStackTrace();

}

} else if (thread2 == null && button.getText() == "Text2") {

thread2 = new Thread(button2);

thread2.start();

try {

thread2.sleep(500);

} catch (InterruptedException e1) {

// TODO Auto-generated catch block

e1.printStackTrace();

}

} else if (thread1 != null && thread1.getState().toString().equalsIgnoreCase("TIMED\_WAITING")

&& thread1.getName().equalsIgnoreCase("suspend") && button.getText().equalsIgnoreCase("Text1")) {

thread1.resume();

thread1.setName("");

} else if (thread2 != null && thread2.getState().toString().equalsIgnoreCase("TIMED\_WAITING")

&& thread2.getName().equalsIgnoreCase("suspend") && button.getText().equalsIgnoreCase("Text2")) {

thread2.resume();

thread2.setName("");

} else if (thread1 != null && thread1.getState().toString().equalsIgnoreCase("TIMED\_WAITING")

&& button.getText().equalsIgnoreCase("Text1")) {

thread1.suspend();

thread1.setName("suspend");

} else if (thread2 != null && thread2.getState().toString().equalsIgnoreCase("TIMED\_WAITING")

&& button.getText().equalsIgnoreCase("Text2")) {

thread2.suspend();

thread2.setName("suspend");

}

}

}

public static void main(String[] arg) {

Exo2 main = new Exo2("Exo1b");

main.setVisible(true);

}

// Kill the thread before closing the windows

@Override

public void windowClosing(WindowEvent e) {

if (thread1 != null) {

thread1.stop();

}

if (thread2 != null) {

thread2.stop();

}

this.setVisible(false);

System.exit(0);

}

@Override

public void windowActivated(WindowEvent e) {

}

@Override

public void windowClosed(WindowEvent e) {

}

@Override

public void windowDeactivated(WindowEvent e) {

}

@Override

public void windowDeiconified(WindowEvent e) {

}

@Override

public void windowIconified(WindowEvent e) {

}

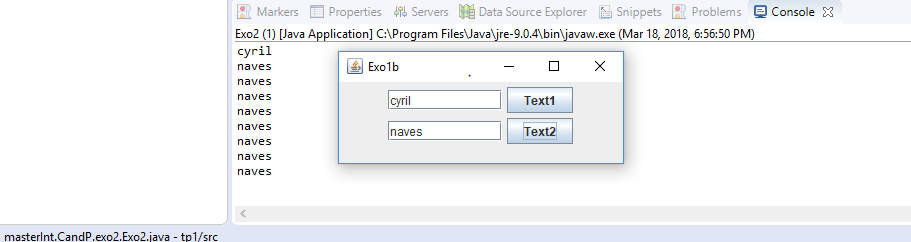
@Override

public void windowOpened(WindowEvent e) {

}

}

**Output**:



# Exercise 3: PipedStream

Write a program where a reader thread catches the stream of characters typed on the keyboard (*System.in*). When a full line has been typed (the key *<*enter*>* has been pressed) (*java.util.Scanner*), the line is put into a pipe (*java.io.PipedOutputStream*).

At the other end of the pipe, another thread reads the content of the pipe (*PipedInputStream*) and prints it in the text area (*javax.swing.JTextArea*) of a frame, line by line.

Could you kill the reader when the frame is closed?

**Summary for the PipedStream implementation:**

1. Reader Class handles the input of the text from the console which gets the input from the console through Scanner Class
2. In Reader PipedOutPutStream is used to send the message through the Pipe which is passed thorugh the constructor instantiation from the main Exo3 Class
3. In Writer PipedInputStream is used to receive the message from the pipe in bytes form (int) then converted to char which is then appended to string.
4. Exo3 Class contains the instantiation of PipedInputStream , PipedOutPutStream, as well as the instance variable are passed to the object constructors of Writer and Reader Class.
5. PipedInputStream is connected to the PipedOutPutStream through the connect method.
6. Then ExecutorService is used to create a threadpool with the threads of Reader and Writer objects
7. Threads are started through Executor Service.
8. Window Closing Event is handled in the where the current thread running is closed.

**Code:**

package masterInt.CandP.exo3;

import java.awt.FlowLayout;

import java.awt.event.WindowEvent;

import java.awt.event.WindowListener;

import java.io.BufferedReader;

import java.io.ByteArrayInputStream;

import java.io.DataInputStream;

import java.io.IOException;

import java.io.InputStream;

import java.io.InputStreamReader;

import java.io.PipedInputStream;

import javax.swing.JFrame;

import javax.swing.JTextArea;

public class Printer extends JFrame implements WindowListener, Runnable {

private static final long serialVersionUID = 4835711038057686272L;

PipedInputStream pipedInputStream = null;

private JTextArea textarea;

public Printer(PipedInputStream \_pipe) {

this.pipedInputStream = \_pipe;

// Set up the window

this.setSize(250, 200);

this.setDefaultCloseOperation(DO\_NOTHING\_ON\_CLOSE);

this.setLayout(new FlowLayout());

this.addWindowListener(this);

// Set up the text area

textarea = new JTextArea(9, 20);

textarea.setEditable(false);

this.add(textarea);

this.setVisible(true);

}

// Read when available

// And print in the Text area

@Override

public void run() {

while (true) {

StringBuilder fina = new StringBuilder();

char text;

int c;

try {

int count = pipedInputStream.available();

for (int i = 0; i < count; i++) {

text = (char) pipedInputStream.read();

fina.append(text);

}

if (!fina.toString().isEmpty()) {

textarea.append(fina.toString() + "\n");

}

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

public String convertto(InputStream is) {

StringBuilder sb = null;

String a;

BufferedReader br = new BufferedReader(new InputStreamReader(is));

try {

while ((a = br.readLine()) != null) {

sb.append(a);

}

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return sb.toString();

}

// Kill the thread before killing the program

@Override

public void windowClosing(WindowEvent arg0) {

Thread.currentThread().interrupt();

this.setVisible(false);

System.exit(0);

}

@Override

public void windowActivated(WindowEvent arg0) {

}

@Override

public void windowClosed(WindowEvent arg0) {

}

@Override

public void windowDeactivated(WindowEvent arg0) {

}

@Override

public void windowDeiconified(WindowEvent arg0) {

}

@Override

public void windowIconified(WindowEvent arg0) {

}

@Override

public void windowOpened(WindowEvent arg0) {

}

}

**package** masterInt.CandP.exo3;

**import** java.io.IOException;

**import** java.io.PipedOutputStream;

**import** java.util.Scanner;

**public** **class** Reader **implements** Runnable {

String lines = **null**;

PipedOutputStream pipedOutputStream = **null**;

**public** Reader(PipedOutputStream pipedOutputStream) {

**this**.pipedOutputStream = pipedOutputStream;

}

// Read input stream when available

// send it in the pipe

**public** **void** run() {

**while** (**true**) {

System.***out***.println("Enter the stream");

Scanner scanner = **new** Scanner(System.***in***);

String total = "";

String input = scanner.next();

total += input;

**if** (**true**) {

**try** {

pipedOutputStream.write(total.getBytes());

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

}

package masterInt.CandP.exo3;

import java.io.IOException;

import java.io.PipedOutputStream;

import java.util.Scanner;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.io.PipedInputStream;

public class Exo3 {

public static void main(String[] args) {

PipedOutputStream pipedOutputStream = new PipedOutputStream();

PipedInputStream pipedInputStream = new PipedInputStream();

try {

pipedOutputStream.connect(pipedInputStream);

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

// Create the printer and the reader

Reader reader = new Reader(pipedOutputStream);

Printer printer = new Printer(pipedInputStream);

// Start them

Thread threadReader = new Thread(reader);

Thread threadWriter = new Thread(printer);

ExecutorService service = Executors.newFixedThreadPool(2);

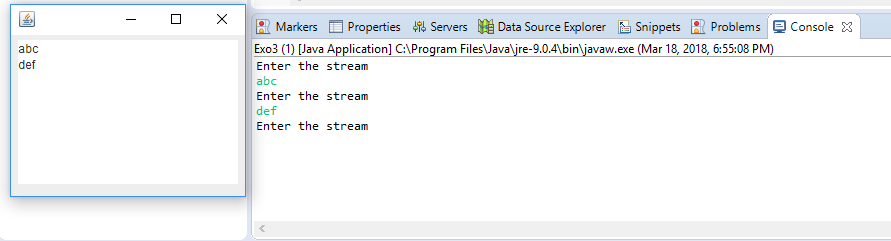
service.execute(threadReader);

service.execute(threadWriter);

}

}

**Output:**



# Exercise 4: Parallel bubble sort

Write a program which takes as input a huge array of numbers. This array is split into *n* sub-arrays and *n* threads apply a bubble sort on each of the *n* sub-arrays. Lastly, another thread merges the *n* sorted sub-arrays into one with the same size as the original array. Of course, the resulting array should be sorted.

Compare the execution of a sequential bubble sort on an array of 40000 elements with the execution of the parallel version (as described above).

**Summary for the Parallel Bubble Sort implementation:**

1. Bubble Sorter Class Implements the Runnable Interface and overrides the run() method where the bubble\_srt() method is called.
2. In Exo4 the Divide and Conquer method is applied to divide the 40000 elements to each of 10000 elements 4 times(n times)
3. Here 4 threads are run in concurrent mode and all of them are made to join (wait for the other threads to finish execution)
4. Then each of the individual rows are then merged using fusion of the input array.
5. Here fusion thread is made to execute and then made to join so that when the execution is complete the result array is then
6. Parallel Execution of threads employs dividing the computation task upto n threads and then they are finally merged using a single thread.
7. On comparing the parallel execution of the bubble sort , with the sequential bubble sort

**Sequential bubble sort time: 4087 ms**

**Parallel Sorting Time: 236 ms**

Parallel Sorting Time is much faster compared to the sequential sorting since the time to compute a much bigger elements of 10000 elements of 4 times is much easier compared to 40000 since he time to merge or fuse a sorted sub array of elements is much faster.

**Code:**

**package** masterInt.CandP.exo4;

//This class implements the bubble sorting algorithm in a thread

**public** **class** BubbleSorter **implements** Runnable {

**private** **int**[] tab;

**public** BubbleSorter(**int**[] \_tab) {

**super**();

**this**.tab = \_tab;

// ...

}

// Run the bubble sort algorithm on tab.

@Override

**public** **void** run() {

**long** start\_time = System.*currentTimeMillis*();

bubble\_srt();

System.***out***.println("Bubble Sorting time=" + (System.*currentTimeMillis*()-start\_time) + " ms");

}

// Here is the bubble sort algorithm on tab

**public** **void** bubble\_srt() {

**int** t, n = tab.length;

**for** (**int** i = 0; i < n; i++)

**for** (**int** j = 1; j < (n - i); j++)

**if** (tab[j - 1] > tab[j]) {

t = tab[j - 1];

tab[j - 1] = tab[j];

tab[j] = t;

}

}

// This main aims at trying the bubble sorting algorithm and compare the

// sequential sorting with the parallel sorting

**public** **static** **void** main(String[] args) {

// Set up the table with random values

**int** size = 40000;

**int**[] tab = **new** **int**[size];

**for** (**int** i = 0; i < size; i++) {

tab[i] = (**int**) (Math.*random*() \* size);

}

// Do the bubble sort ...

**long** start\_time = System.*currentTimeMillis*();

**int** t, n = tab.length;

**for** (**int** i = 0; i < n; i++) {

**for** (**int** j = 1; j < (n - i); j++) {

**if** (tab[j - 1] > tab[j]) {

t = tab[j - 1];

tab[j - 1] = tab[j];

tab[j] = t;

}

}

}

System.***out***.println("Bubble Sorting time=" + (System.*currentTimeMillis*() - start\_time) + " ms");

}

}

**package** masterInt.CandP.exo4;

//This class implements the fusion operation in a thread

**public** **class** Fusion **implements** Runnable {

**private** **int**[][] tab;

**private** **int**[] result;

**public** Fusion(**int**[][] \_tab) {

**this**.tab = \_tab;

}

// Run the fusion algorithm on tab

**public** **void** run() {

**long** start\_time = System.*currentTimeMillis*();

fusion();

System.***out***.println("Fusion time=" + (System.*currentTimeMillis*() - start\_time) + " ms");

}

// Implement the fusion algorithm

**private** **int**[] fusion() {

**int** size = 0;

**int** tabIndex;

// Set up the size of the resulting merged table

**for** (**int** i = 0; i < tab.length; i++)

size += tab[i].length;

**int**[] result = **new** **int**[size];

// Set up the indices table

**int**[] indices = **new** **int**[tab.length];

**for** (**int** i = 0; i < tab.length; i++)

indices[i] = 0;

// Fill up the resulting merged table

**for** (**int** i = 0; i < result.length; i++) {

// Find the first table where there is still values to merge

tabIndex = -1;

**for** (**int** j = 0; j < tab.length; j++)

**if** (indices[j] < tab[j].length) {

tabIndex = j;

**break**;

}

// Compare the value in the current table with the value in the next table

**for** (**int** j = tabIndex + 1; j < tab.length; j++) {

**if** ((indices[j] < tab[j].length) && (tab[tabIndex][indices[tabIndex]] > tab[j][indices[j]]))

tabIndex = j;

}

// Add the correct value to the resulting merged table

result[i] = tab[tabIndex][indices[tabIndex]];

// Increase the index for the correct table

indices[tabIndex]++;

}

**this**.result = result;

**return** result;

}

// Get the resulting merged table

**public** **int**[] getResult() {

**return** result;

}

}

**package** masterInt.CandP.exo4;

**public** **class** Exo4 {

/\*\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

// Set up the table with random values

**int** nb = 4; // number of tables

**int** size = 10000;// Size of each table

**int**[][] tab = **new** **int**[nb][size];

**for** (**int** i = 0; i < nb; i++) {

**for** (**int** j = 0; j < size; j++) {

tab[i][j] = (**int**) (Math.*random*() \* size);

}

}

/\*

\* //Print the original tabs for(int i=0;i<nb;i++) { for(int j=0;j<size;j++)

\* System.out.print(""+tab[i][j]); System.out.println("\n"); }

\*/

// Create the bubble sorters

BubbleSorter bubbleSorterfirst = **new** BubbleSorter(tab[0]);

Thread firstthread = **new** Thread(bubbleSorterfirst);

BubbleSorter bubbleSortersecond = **new** BubbleSorter(tab[1]);

Thread secondthread = **new** Thread(bubbleSortersecond);

BubbleSorter bubbleSorterThird = **new** BubbleSorter(tab[2]);

Thread thirdthread = **new** Thread(bubbleSorterThird);

BubbleSorter bubbleSorterfourth = **new** BubbleSorter(tab[3]);

Thread fourththread = **new** Thread(bubbleSorterfourth);

// Get the current time

**long** start\_time = System.*currentTimeMillis*();

// Do bubble sorting

firstthread.start();

secondthread.start();

thirdthread.start();

fourththread.start();

// Waiting for the Bubble sorters

**try** {

firstthread.join();

secondthread.join();

thirdthread.join();

fourththread.join();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

// Do the fusion

// Wait for the fusion to complete

Fusion fusion = **new** Fusion(tab);

Thread threadfusion = **new** Thread(fusion);

threadfusion.start();

**try** {

threadfusion.join();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**int**[] mergedTab = fusion.getResult();

// Print the total execution time

System.***out***.println("Total time=" + (System.*currentTimeMillis*() - start\_time) + "ms");

// Print result

**for** (**int** ij = 0; ij < mergedTab.length; ij++) {

System.***out***.println("" + mergedTab[ij]);

}

}

}

**Output:**

**Parallel Sorting:**

Bubble Sorting time=227 ms

Bubble Sorting time=227 ms

Bubble Sorting time=228 ms

Bubble Sorting time=230 ms

Fusion time=5 ms

Total time=236ms

**Sequential Sorting:**

Bubble Sorting time=4087 ms

1