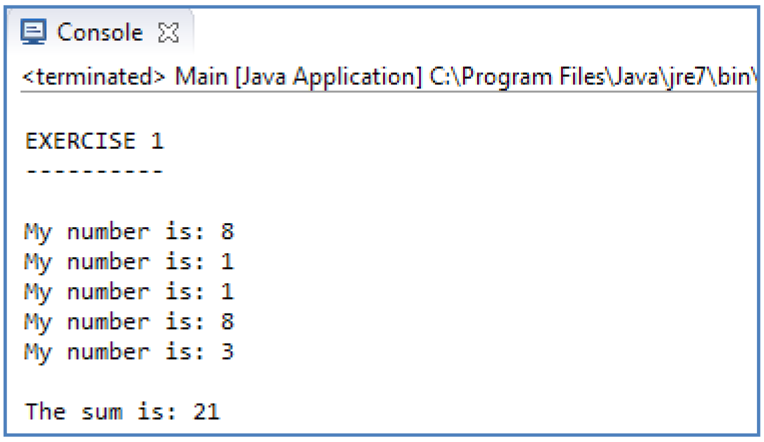
PARALLELISM AND CONCURRENCY
JAVA THREADS. EXERCISES

EXERCISE 1

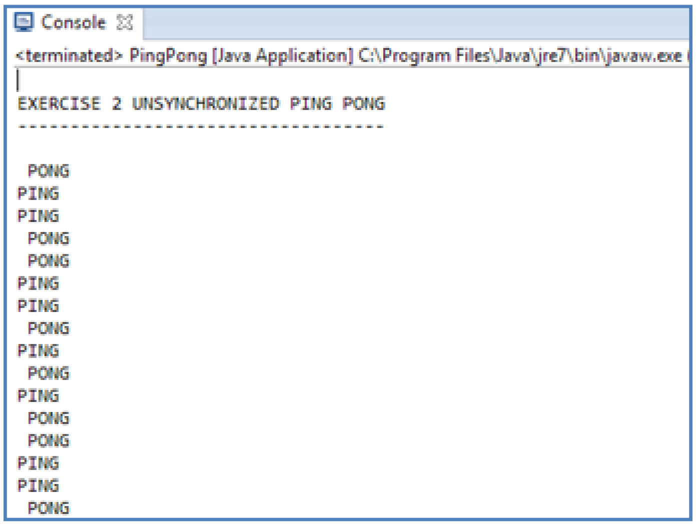
Write a program that runs 5 threads, each thread randomizes a number between 1 and 10, writes it on the console and stores it in a private attribute (provide a getter for that attribute). The "main" thread waits for all the others to finish, calculates the sum of the numbers which were randomized and prints the sum. Use method `join`.



```
<terminated> Main [Java Application] C:\Program Files\Java\jre7\bin\

EXERCISE 1
----------

My number is: 8
My number is: 1
My number is: 1
My number is: 8
My number is: 3

The sum is: 21
```

EXERCISE 2: THE PING-PONG PROBLEM

a) Write 2 subclasses of Thread, one that endlessly prints *PING* and sleeps for a number of milliseconds (e.g. 10) and another that endlessly prints *PONG* and sleeps for the same number of milliseconds.
   Then, write a "main" program that launches one instance of the ping thread and one instance of the pong thread. Make the main program wait for 5 seconds and then stop the launched threads. Observe the result...



```
<terminated> PingPong [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe

EXERCISE 2 UNSYNCHRONIZED PING PONG
-----------------------------------

 PONG
PING
PING
 PONG
 PONG
PING
PING
 PONG
PING
 PONG
PING
 PONG
 PONG
PING
PING
 PONG
```

b) You will probably notice that sometimes two (or more) consecutive PONGS (or PINGS) appear. Add a “synchronization mechanism” that makes the interleaving of PINGS and PONGS perfect (each ping is preceded and followed by a pong, each pong is preceded and followed by a ping). Consider these two possibilities:

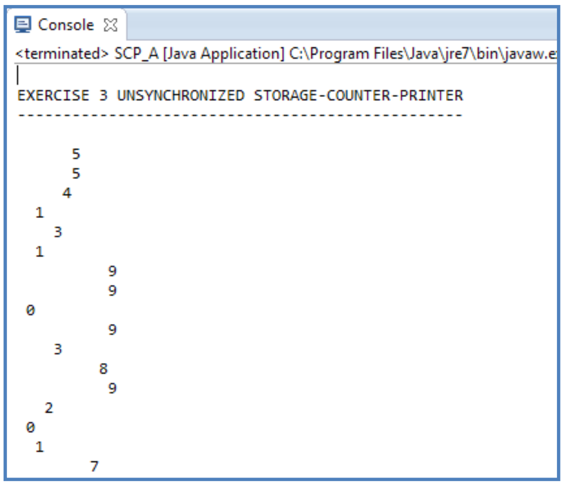
   a. Each thread has a publicly accessible canPrint attribute the value of which is true if and only if the thread is allowed to print its message. Before printing, the thread busy waits until its canPrint attribute is true. After printing, the thread changes its canPrint attribute to false, its companion’s to true and goes to sleep for a while (Provide a setCompanion method so that the main thread can set each thread’s companion). WITH THIS SOLUTION EACH THREAD HAS TO KNOW THE OTHER ONE (THREADS HAVE TO KNOW EACH OTHER)

   b. Both threads share an object that has an int attribute. If its value is 1, PING can print its line, if its value is 2, PONG can print its line. (WITH THIS SOLUTION, THREADS DO NOT HAVE TO KNOW EACH OTHER).

EXERCISE 3: THE STORAGE-COUNTER-PRINTER PROBLEM

a) Create three Classes: Storage, Counter and Printer. The Storage class has to passively[1] store an integer. Instances of the Counter class have to endlessly count from 0 to 9 and store each value in a Storage instance. The instances of the Printer class have to keep reading the value in the Storage instance and printing it on the console.

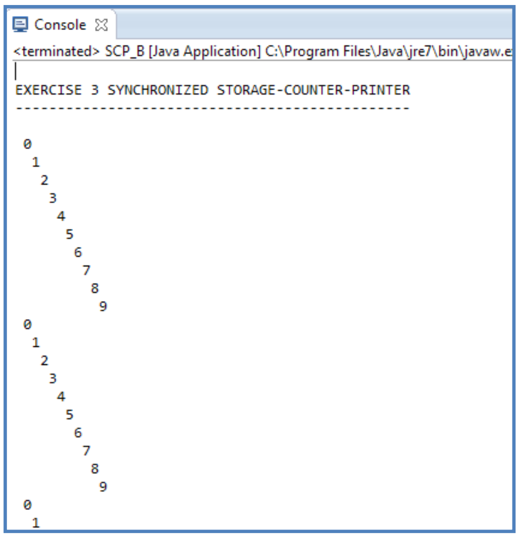
   Write a program that creates an instance of the Storage class, and sets up a Counter and a Printer object to operate on it. Let the program run for a limited time (e.g. 5 seconds).



```
<terminated> SCP_A [Java Application] C:\Program Files\Java\jre7\bin\javaw.e
|
EXERCISE 3 UNSYNCHRONIZED STORAGE-COUNTER-PRINTER
-------------------------------------------------

      5
      5
     4
  1
    3
  1
          9
          9
 0
          9
    3
         8
          9
   2
 0
  1
        7
```

[1] A “passive” object is an object that is not a Thread

b) Modify the program to ensure that each number is printed exactly once, by adding suitable synchronization. Consider using an object that allows Printer and Counter communicating with each other. Try to avoid busy-waiting. Instead, make threads sleep for a short time (e.g. some nanoseconds), check for the desired condition to be true and sleep again if it is false.

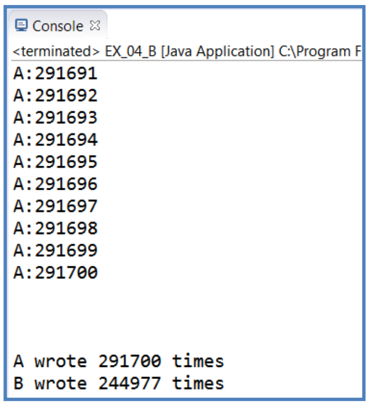


```
<terminated> SCP_B [Java Application] C:\Program Files\Java\jre7\bin\javaw.e

EXERCISE 3 SYNCHRONIZED STORAGE-COUNTER-PRINTER
-----------------------------------------------

 0
  1
   2
    3
     4
      5
       6
        7
         8
          9
 0
  1
   2
    3
     4
      5
       6
        7
         8
          9
 0
  1
```

c) Let the Storage objects take care of the synchronization issues. Modify the getValue and the setValue methods so that they keep the invoking thread waiting if the value cannot be got/set.

<u>EXERCISE 4</u>

a) Write a subclass of Thread the instances of which <u>endlessly</u> print a String composed of a line (given as a parameter) and a number reflecting the times the main loop has been performed (counter).
Writer a "Launcher" program that creates two instances of the threads previously described, starts them, waits (sleeps) for a number of seconds (e.g. 5), kills the threads and writes the total number of times each thread has performed its main loop.



```
<terminated> EX_04_B [Java Application] C:\Program F
A:291691
A:291692
A:291693
A:291694
A:291695
A:291696
A:291697
A:291698
A:291699
A:291700



A wrote 291700 times
B wrote 244977 times
```

b) Change the "Launcher" program so that before starting the threads it gives MAX_PRIORITY to one of them and MIN_PRIORITY to the other one. Execute several times to see whether the given priorities have any effect on the number of iterations performed or not.