JAVA PROGRAMMING II

Module 2, 3: More on Threads & java.lang package
Lab Guide for Lab2

Session Objectives

In this session, you will

- Create multi-threading program
- Use isAlive() and join() methods
- Apply thread synchronization.
- Apply wait-notify mechanism
- Use java.lang package
- Use Wrapper classes

Part 1 – Getting started (30 minutes)

Exercise 1: Create threads with shared resources

The following program simulates a Counter object that works as shared resource, used by 2 threads.

Scan the code first, copy the code, compile, run and observe the result.

File: **Counter.java**

```
public class Counter {
    int value;
       // The forward-count method, that increases the value from 0 to
9,
    // print out the value in each steps (with 300ms delay)
    public void count() {
        value = 0;
        while (value < 10) {
            System.out.println("Count: " + (++value));
            pause(300);
```

```
        }
    }

    // The downward-count method, that dereases the value from 10 to 0,
       // print out the value in each steps (with 200ms delay)
    public void countdown() {
        value = 10;
        while (value >= 0) {
            System.out.println("Countdown: " + (value--));
            pause(200);
        }
    }
       // Sleep method
    private void pause(long time) {
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {}
    }
}
```

File **SyncDemo.java**

```
public class SyncDemo {
    public static void main(String[] args) {
        final Counter ct = new Counter();
              // Create new runnable object t1,
        // therefore t1 is a thread is using Counter object ct
        Runnable t1 = new Runnable() {
            public void run() {
                ct.count();
            }
        };
        // Create new runnable object t2,
        // Therefore t2 is a thread using Counter object ct.
        // Both t1 and t2 are using ct at the same time,
        // so ct is called shared-resource
```

```
        Runnable t2 = new Runnable() {
            public void run() {
                ct.countdown();
            }
        };
        new Thread(t1).start();
        new Thread(t2).start();
    }
}
```

Perform the following steps and answer these questions:

1. Compile and run file **SyncDemo.java**. What's the result?

2. Add the *synchronized* keyword before the *count()* and *countdown()* methods, so they will be:

```
public synchronized void count() {
    //...
}
public synchronized void countdown {
    //...
}
```

   Compile and run. Scan and explain the result.

3. In another way, we DO NOT add the *synchronized* keyword as within step 2, we synchronize shared-resource **ct** by adding **synchronized** keyword as below:

```
...
synchronized(ct) {
    ct.count();
}
...
synchronized(ct) {
    ct.countdown();
}
...
```

   Compile and run. Scan and explain the result.

Part 2 – Workshops (30 minutes)

- Quickly look at workshops of Module 2 and Module 3 for reviewing basic steps for creating and managing threads with synchronization concepts and wait-notify mechanisms.
- Try to compile, run and observe the output of sample code provided for related workshops. Discuss with your class-mate and your instructor if needed.

Part 3 – Lab Assignment (60 minutes)

Do the assignment for Module 2 and Module 3 carefully. Discuss with your class-mates and your instructor if needed.

Part 4 – Do it yourself

Exercise 1:

Write a program to simulate the working process in a factory:

- There is task list that defines all the tasks to be performed
- A manager keeps on adding new tasks to the list
- There are several workers who check the task list regularly and take one of the tasks listed there to carry out. Each task should be taken by only one worker.

You should implement the manager and workers as concurrent threads.

Notes: A task can be represented simply as a String holding the task's name. Performing a task is as simple as printing out the worker's name and the task's name.

Hints:

- The task list is the shared resource
- To ensure any single task is taken by only one worker (i.e. there is no conflict), you may have to synchronize the threads
- The worker threads may have to wait for the manager thread to define new tasks.