2017학년도 1학기

JAVA 프로그래밍 II

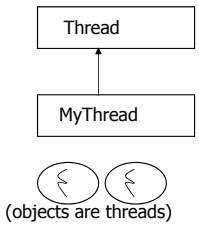
514770-1
2017년 봄학기
5/24/2017
박경신

Lab #6 (Multi-Thread)

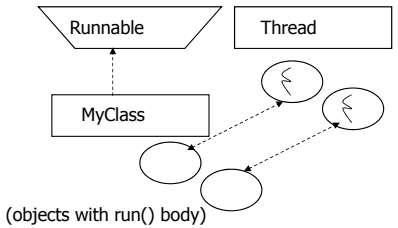
- 기존 요구사항 분석
  - Lab #5는 AWT/SWING을 이용한 GUI 프로그래밍
  - **Lab #6는 Thread, Runnable과 SwingWorker를 이용한 다양한 멀티스레드 기능을 사용**
- Thread, Mutex, Semaphore, Monitor, Worker thread (a.k.a. background thread)
- Thread 클래스, Runnable 인터페이스, SwingWorker 추상클래스

Lab #6_1 extends Thread vs implements Runnable

1. Create a class that extends the Thread class
2. Create a class that implements the Runnable interface



[a]



[b]

Lab #6_1 extends Thread vs implements Runnable

- **Lab#6_1에서는 Thread 클래스를 상속 받는 Multithread 클래스를 구현한다.**
  - ***public class Multithread extends Thread {***

```
    private static int count = 0; // global counter
// 중간생략..
    @Override
    public void run() {
        while(true) {
            System.out.println("global counter=" + count);
            count++;
        }
    }
}
```

Lab #6_1 extends Thread vs implements Runnable

- Lab#6_1에서는 Runnable 인터페이스를 상속 받는 MultithreadRunnable 클래스를 구현한다.

```
public class MultithreadRunnable implements Runnable {
    private static int count = 0; // global counter
// 중간생략..
    @Override
    public void run() {
        while(true) {
            System.out.println("global counter=" + count);
            count++;
        }
    }
}
```

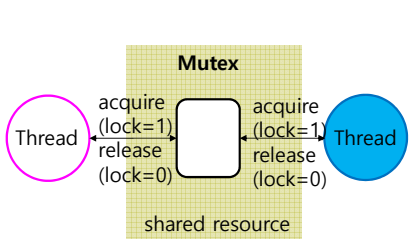
Lab #6_1 extends Thread vs implements Runnable

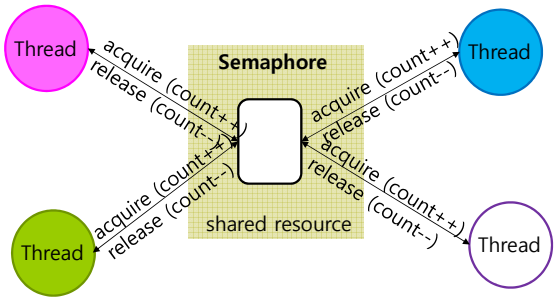
- Lab#6_1에서는 Thread 클래스 또는 Runnable 인터페이스를 상속 받는 간단한 멀티스레드 프로그램을 구현한다.

```
public class MultithreadTest {
    public static void main(String[] args) {
        new Multithread("Thread1 :").start();
        new Multithread("Thread2 :").start();
        new Multithread("Thread3 :").start();
        new Thread(new MultithreadRunnable("Thread1 :")).start();
        new Thread(new MultithreadRunnable("Thread1 :")).start();
        new Thread(new MultithreadRunnable("Thread1 :")).start();
    }
}
```

Mutex vs Semaphore vs Monitor

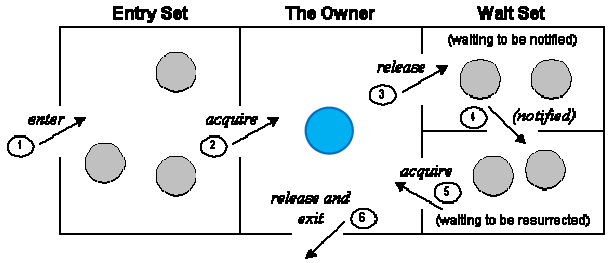
- **Mutex** (mutual execlusion) semaphore controls **only one thread at a time** executing on the shared resource.
- **Semaphore** controls **the number of threads** executing on the shared resources.





Mutex vs Semaphore vs Monitor

- **Monitor** controls **only one thread at a time**, and can execute in the **monitor (shared object)**



Lab #6_2 Mutex (Mutual Exclusion Semaphore)

- Lab#6_2에서는 Mutex를 사용하여 멀티 스레드가 공유 자원(shared resource)을 하나씩 access하도록 한다. Lab6_2 실행 결과를 Lab6_1과 비교한다.

```
private static Semaphore mutex = new Semaphore(1); // binary semaphore
public void run() {
        while (true) {
                try {
                        mutex.acquire(); // mutex lock
                        System.out.println(getName() + " global count=" + count);
                        count++;
                        mutex.release(); // mutex unlock
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                        return;
                }
        }
```

Lab #6_2 Mutex (Mutual Exclusion Semaphore)

- Lab#6_2에서는 Mutex를 사용하여 멀티 스레드가 공유 자원(shared resource)을 하나씩 access하도록 한다. Lab6_2 실행 결과를 Lab6_1과 비교한다.

```
public class MultithreadMutexTest {
    public static void main(String[] args) {
            new MultithreadMutex("Thread1 :").start();
            new MultithreadMutex("Thread2 :").start();
            new MultithreadMutex("Thread3 :").start();
            new Thread(new MultithreadRunnableMutex("Thread1 :")).start();
            new Thread(new MultithreadRunnableMutex("Thread1 :")).start();
            new Thread(new MultithreadRunnableMutex("Thread1 :")).start();
    }
}
```

- Java2-lab6_2 폴더에 저장 후 제출

Lab #6_3 Semaphore (ticketSeller)

- Lab#6_3에서는 Semaphore를 사용하여 4개의 seller(즉, 멀티스레드)에서 공유 자원(즉, ticket)을 동시에 3개까지 판매 가능한 ticketSeller를 구현한다.

```
private static Semaphore semaphore = new Semaphore(3); // semaphore with max access count = 3
public void run() {
        while (!done) {
                semaphore.acquire(); // semaphore aquire
                if (numTicket == 0) done= true;
                else {
                        sold++;
                        numTicket--;
                        System.out.println(getName() + " sold=" + sold + " ticket=" + numTicket);
                }
                semaphore.release(); // semaphore release
        }
}
```

Lab #6_4 Monitor

- Lab#6_4에서는 Monitor를 사용하여 멀티 스레드 프로그램을 구현한다.

```
public class SynchronizedMultithread extends Thread {
    SharedCounter counter; // global counter (using monitor)
// 중간생략..
    @Override
    public void run() {
            int i = 0;
            while(true) {
                    counter.nextCount(i);
                    i++;
            }
    }
}
```

Lab #6_4 Monitor

- 모든 클래스는 Monitor로 구현되어 있다. synchronized 키워드를 사용하여 공유 자원 (shared object)을 동기화 (synchronization) 한다.

```
public class SharedCounter {
    private int count = 0;
    public SharedCounter() { }
    public synchronized void nextCount(int i) {
            Thread.yield();
            System.out.println(Thread.currentThread().getName() + " cycle=" + i  + "
global count=" + count);
            count++;
    }
}
```

Lab #6_4 Monitor

- 공유객체 counter는 Monitor로 동기화 (synchronization) 된다.

```
public class MonitorTest {
    public static void main(String[] args) {
            SharedCounter counter = new SharedCounter();
            new SynchronizedMultithread("SynThread1 :", counter).start();
            new SynchronizedMultithread("SynThread2 :", counter).start();
            new SynchronizedMultithread("SynThread3 :", counter).start();
    }
}
```

Lab #6_4 Monitor (Producer-Consumer Problem)

- Producer-Consumer Problem (Monitor)

```
public class BoundedBuffer {
    private volatile String[] buffer;
    private volatile int tail, head, count = 0;
    public synchronized void put(String data) { // put data into the buffer
            while (count >= buffer.length) { wait(); } // wait (buffer is full)
            System.out.println("produce=" + data);
            buffer[tail] = data;
            tail = (tail + 1) % buffer.length;
            count++;
            notifyAll(); // signal
    }
```

Lab #6_4 Monitor (Producer-Consumer Problem)

- Producer-Consumer Problem (Monitor)

```
    public synchronized String take() { // take data from the buffer
            while (count <= 0) { wait(); } // wait (buffer is empty)
            String data = buffer[head];
            head = (head + 1) % buffer.length;
            count--;
            System.out.println("produce=" + data);
            notifyAll(); // signal
            return data;
    }
}
```

Lab #6_4 Monitor (Producer-Consumer Problem)

- **Producer-Consumer Problem (Monitor)**

```
public class Producer extends Worker {
    private static volatile int count = 0; // global counter among producer threads
    public Producer(String name, BoundedBuffer data) {
            super(data);
    }
    public void run() {
            while (isRunning()) {
                    String str = "gcount=" + count);
                    data.put(str); // put data into the buffer
                    setCurrentItem(str);
            }
    }
}
```

Lab #6_4 Monitor (Producer-Consumer Problem)

- **Producer-Consumer Problem (Monitor)**

```
public class Consumer extends Worker {
    private static volatile int count = 0; // global counter among producer threads
    public Consumer(String name, BoundedBuffer data) {
            super(data);
    }
    public void run() {
            while (isRunning()) {
                    String str = data.take(); // take data from the buffer
                    setCurrentItem(str);
                    System.out.println(Thread.currentTread().getName() + str);
            }
    }
}
```

Lab #6_4 Monitor (Producer-Consumer Problem)

- **공유객체 counter는 Monitor로 동기화 (synchronization) 된다.**

```
public class MonitorTest {
    public static void main(String[] args) {
            BoundedBuffer data = new BoundedBBuffer(5); // buffer size=5
            new Thread(new Producer("Producer1 :", data)).start();
            new Thread(new Producer("Producer2 :", data)).start();
            new Thread(new Producer("Producer3 :", data)).start();
            new Thread(new Consumer("Consumer1 :", data)).start();
            new Thread(new Consumer("Consumer2 :", data)).start();
            new Thread(new Consumer("Consumer3 :", data)).start();
            new Thread(new Consumer("Consumer4 :", data)).start();
    }
}
```

Lab #6_3 Semaphore (Producer-Consumer Problem)

```
Producer() {
        while (1) { <<< Produce item >>>
                P(empty); // Get an empty buffer (decrease count) , block if unavailable
                P(mutex); // acquire critical section: shared buffer
                <<< critical section: Put item into shared buffer >>>
                V(mutex); // release critical section
                V(full); // increase number of full buffers
        }
    }
Consumer() {
        while (1) {
                P(full);
                P(mutex);
                <<< critical section: Take item from shared buffer >>>
                V(mutex);
                V(empty);
        }
}
```

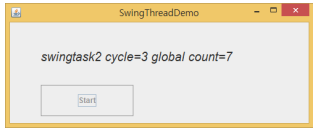
Concurrency in Swing

- Initial Thread – the threads that execute initial application code.
- Event Dispatch Thread – where all event-handling code is executed.
- Work Thread – also known as background threads, where time-consuming background tasks are executed.

Lab #6_5 SwingWorker

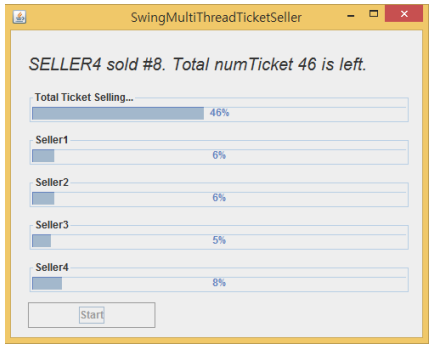
- **Lab#6_5에서는 SwingWorker를 사용하여 GUI label에 count를 출력한다.**

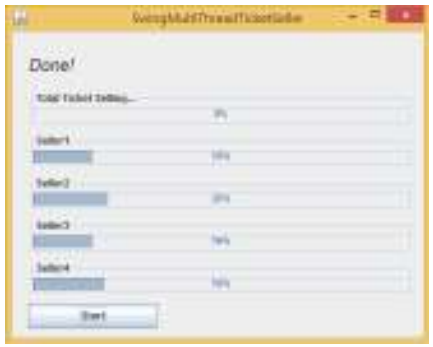
```
class Task extends SwingWorker<Boolean, String> {
    public Boolean doInBackground() { // executed in background thread
        while (count < 100) {
            String status = this.name + " global count=" + count;
            publish(status);
            count++;
        } return true;
    }
    public void done() { // executed in event dispatching thread when the background task is done
        try {
            if (get()) { label.setText("Done!₩n"); }
        } catch (InterruptedException e) {
        } catch (ExecutionException e) { }
    }
    public void process(List<String> chunks) { // executed in event dispatching thread
        String status = chunks.get(chunks.size() - 1);
        label.setText(status);
    }
}
```



Lab #6_5 SwingWorker

- **Lab#6_5에서는 SwingWorker를 사용하여 progressBar에 ticket seller를 출력한다.**





Lab #6_5 SwingWorker

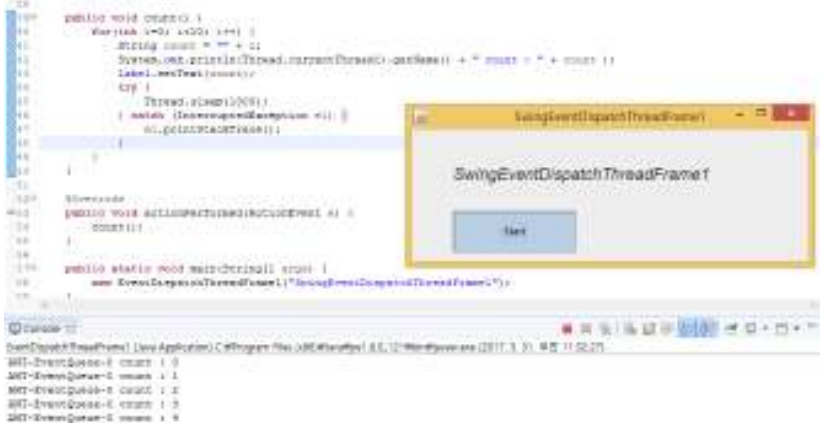
- **Lab#6_5에서는 SwingWorker를 사용하여 Producer-Consumer Problem을 textArea에 출력한다.**

Lab #6_5 SwingWorker

- Lab#6_5에서는 SwingWorker를 사용하여 image를 background로 loading후 화면에 보여준다.

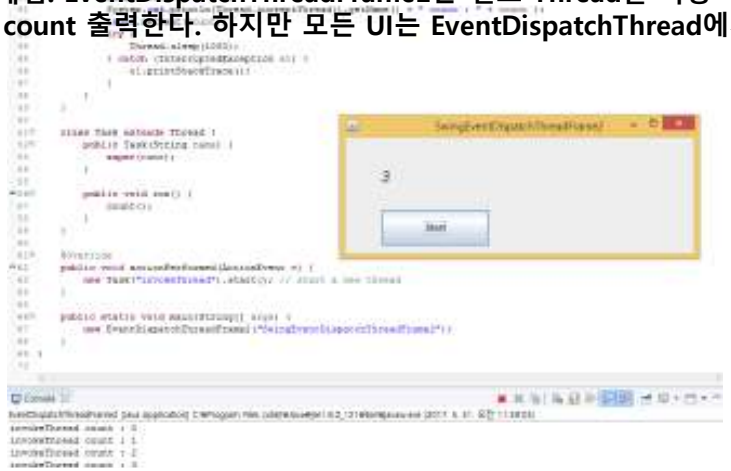
Lab #6_6 EventDispatchThread

- 문제: EventDispatchThreadFrame1은 버튼 actionPerformed하면 레이블에 count 출력을 원하나, 마지막 9 숫자만 화면 레이블에 나타나게 된다.



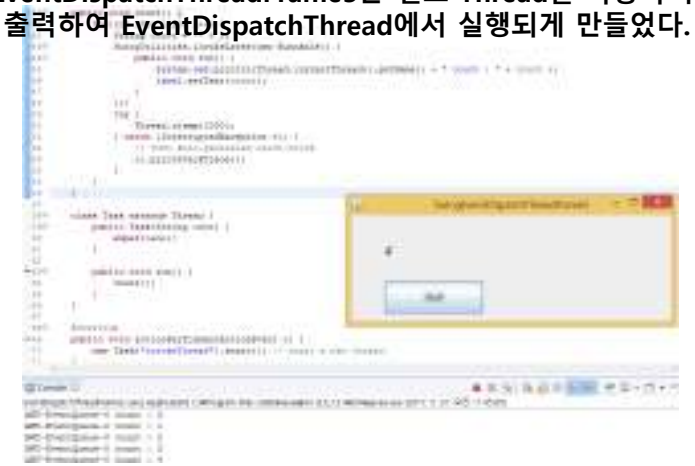
Lab #6_6 EventDispatchThread

- 단순해법: EventDispatchThreadFrame2은 별도 Thread를 사용하여 화면 레이블에 count 출력한다. 하지만 모든 UI는 EventDispatchThread에서 실행되어야 한다.



Lab #6_6 EventDispatchThread

- 해법: EventDispatchThreadFrame3은 별도 Thread를 사용하여 화면 레이블에 count 출력하여 EventDispatchThread에서 실행되게 만들었다.



과제 제출

- Lab06_1 ~ Lab06_6와 보고서를 전체적으로 묶어서 e-learning에 과제 제출
- 각 Lab마다 **본인이 추가로 작성한 코드**와 설명을 중점적으로 보고할 것!