Multi-Threading

① Synchronize ! — method — Non-Static (lock on object level)
— Static (Lock on Class Level)
— Block Statement

② Wait - Notify ! ① Inter-Thread Comm.
② Avoid Polling / Reduce CPU Utilization

wait() and notify(): wait() tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and calls notify().

```
Class Q {
    int n;
    boolean empty = false;
    Synchronized int get() {
        while (empty) {
            try { wait(); }
            catch (InterruptedException e) {
                e.getMessage();
            }
        }
        SOP("value: " + n);
        empty = true;
        notify();
    }
    Synchronized void set (int x) {
        while (!empty) {
            try { wait(); }
            catch (InterruptedException e) {
                e.getMessage();
            }
        }
        n = x;
        empty = false;
        notify();
    }
```

```
class Producer implements Runnable {
    Q q;
    Producer(Q q) { this.q = q; }
    public void run() {
        int i = 0;
        while (true) { q.put(i++); }
    }
}
class Consumer implements Runnable {
    Q q;
    Consumer(Q q) { this.q = q; }
    public void run() {
        while (true) { q.get(); }
    }
}
class Demo {
    Queue q = new Q();
    new Thread(new Producer(q)).start();
    new Thread(new Consumer(q)).start();
}
```

ii) CountDownLatch (A) Thread has to wait for one or more events to be occurred

(B) Not reusable

```
class CDLDemo {
    main() {
    CountDownLatch cdl = new CountDownLatch(5);
    SOP("Starting");
        new MyThread(cdl);
    try { cdl.await(); }
    catch(InterruptedException e) { e.getMessage(); }
    SOP("Done");

}

class MyThread implements Runnable {
    CountDownLatch cd;
    MyThread(CountDownLatch cdl) { this.cd = cdl;
        new Thread(this).start();
    }
    public void run() {
        for(i=1; i<=5; i++) {
            cd.countDown();
        }
    }
}
```

(3) iii) Cyclic Barrier: (A) Set of one or more threads has wait at a predetermined point until all threads in the set have reached

(B) Can be Reusable

```
Class CBDemo{
    CyclicBarrier cb = new CyclicBarrier (3, newBarAction());
    SOP("Starting");
    new MyThread(cb, "A");
    new MyThread(cb, "B");
    new MyThread(cb, "C");
  }
}
class MyThread implements Runnable{
    CyclicBarrier cb;
    MyThread(CyclicBarrier cb){
        this.cb = cb;
        new Thread(this).start();
    }
    public void run(){
        try{ cb.await(); }
        catch(BrokenBarrierException e){ .. }
        catch(InterruptedException e){ .. }
    }
}
class BarAction implements Runnable{
    public void run(){
        SOP("Done");
    }
}
```

Synchronizers — Semaphore
— CountDownLatch
— Cyclic Barrier

③ i) Semaphore: (Kernal variables, If counter is greater than zero, then only access is allowed)
Use counter. Permit to Access.

```
main() {
    Semaphore sem = new Semaphore(1);
    new IncThread(sem);
    new DecThread(sem);
}
Class SharedCell {
    public static int x = 0;
}
Class IncThread {
    Semaphore sem;
    IncThread(Semaphore sem) {
        this.sem = sem;
        new Thread(this).start();
    }
    public void run() {
        sem.acquire();
        for(i=1; i<=5; i++) {
            SharedCell.x++;
        }
        sem.release();
    }
}
```

```
class DecThread {
    Semaphore sem;
    DecThread(Semaphore sem) {
        this.sem = sem;
        new Thread(this).start();
    }

    public void run() {
        sem.acquire();
        for(i=1; i<=5; i++) {
            SOP(SharedCell.x--);
        }
        sem.release();
    }
}
```

Executors: (A) Manages execution of threads

(B) Executor, ExecutorService

ThreadPoolExecutor ScheduledThreadPoolExecutor

(C)

```
Static ExecutorService newCatchedThreadPool
Static ExecutorService newFixedThreadPool
Static ExecutorService newScheduledThreadPool
```

Atomic Operation: (java.util.concurrent.atomic)

This package offers methods that get, set or compare the value of a variable in one uninterruptible (that is, atomic) operation.

↳ AtomicInteger
↳ AtomicLong

methods
- get()
- set()
- getAndSet()
- compareAndSet()
- incrementAndGet()

@ThreadSafe

```
public class CountingFactorizer implements Servlet {
    private final AtomicLong count = new AtomicLong(0);
    public long getCount() { return count.get(); }
    public void service(ServletRequest req, ServletResponse res) {
        count.incrementAndGet();
        // Do work here
    }
}
```

volatile Variable: When a variable is declared volatile the compiler and runtime are put on notice that this variable is shared and that operations on it should not be reordered with other memory operations. Volatile variable are not cached in register or in caches where they are hidden from other processors, so a read of a volatile variable always returns the most recent write by any thread. It only guarantees visibility not Atomicity.