Producer/Consumer Problem

Introduction to the producer/consumer problem

Producer

- Writes on the buffer

Consumer

- Reads from the buffer

The Producer Consumer Problem

- A Producer and a consumer - two threads
  - trying to concurrently access a shared object
- The object is used to store data made by the producer
  - and used by the consumer.
- The producer should not produce new data until the consumer has consumed the last piece of data.
- The consumer should not consume the same data value twice.

Producer/Consumer Problem

Example (**Init_V1 folder**):

- Two threads:
  - The first is trying to store values (producer)
  - and the second is trying to fetch the stored values (consumer)

- We will look at five evolving examples showing the difficulty in synchronizing access to data .

```
class Main { // This class is used to set things in
   motion.
   public static void main(String argv[]) {
        MyData data = new MyData();
        new Thread(new Producer(data)).start();
        new Thread(new Consumer(data)).start();
   }
}
```

Producer

```
class Producer implements Runnable {
   MyData data;
   public Producer(MyData data) {
     this.data = data;
   }
   public void run() {
     int i;
     for (i=0;;i++) {
        data.store(i);
       System.out.println ("Producer: "+i);
       try {
         // doze off for a random time (0 to 0.5 sec)
         Thread.sleep ((int) (Math.random()*500));
       } catch (InterruptedException e) { }
     }
    }
}
```

Consumer

```
class Consumer implements Runnable { // The consumer
   MyData data;
   public Consumer(MyData data) {
       this.data=data;
   }
   public void run() {
   for (;;) {
       System.out.println ("Consumer: "+data.load());
       try {
           // sleep for a random time (0 to 0.5 sec)
          Thread.sleep ((int) (Math.random()*500));
       } catch (InterruptedException e) { }
   }
}
```

Producer Consumer – Goal output

Goal Output:

Producer: 0

Consumer: 0

Producer: 1

Consumer: 1

Producer: 2

Consumer: 2

Producer: 3

Consumer: 3

Producer: 4

Consumer: 4

Producer: 5

…..

- Two threads take turns putting information into the shared object and taking information back out of the shared object

Producer Consumer – Actual Output

Actual Output:

```
Producer: 0
                        Consumer: 0
Producer: 1
                        Consumer: 1
                        Consumer: 1
Producer: 2
Producer: 3
                        Consumer: 3
Producer: 4
Producer: 5
```

- The producer has no way to know that the consumer did not consume the data yet, and therefore he overwrites it.
- The consumer has no way of knowing if the value it reads is a new value or the old one.

Solution 1 - MyData (Sol1_V2)

- Use ordinary Boolean variables as flags to control data access:
  - The *Ready* flag
    - will signify that new data was produced and is ready to be consumed;
  - The *Taken* flag
    - will signify that the data was consumed and it is okay to overwrite it.
- Issue:
  - The *store*() and *load*() methods are using *busy-loops*.
  - The threads are constantly checking the flags to see if their value has been changed.

```
class MyData {
   private int data;
   private boolean ready;
   private boolean taken;

   public MyData() {
        ready = false;
        taken = true;
   }
   public void store(int data) {
        while (!taken);

        this.data = data;
        taken = false;
        ready = true;
   }
   public int load() {
        int data;
        while (!ready);
        data = this.data; // save the
   value because after  Taken turns
   "true" it may  change at any time.

        ready = false;
        taken = true;
        return data;
   }
}
```

Solution 2 – MyData (Sol2_V3)

Declaring the methods synchroniz-ed removes the need for storing the value of the Data variable in the load() method. The load() and store() will not be able to execute at the same time in different threads anymore.

Problem:

- When one thread will engage the busy-wait loop, still owning the monitor. The other thread will never be able to execute its code because it can not acquire the monitor. The two threads are in a deadlock: one of them waits for the value to change while blocking the other one from changing it.

```
class MyData {
   private int data;
   private boolean ready;
   private boolean taken;

   public MyData() {
         ready = false;
         taken = true;
   }

   public synchronized void store(int
   data) {
         while (!taken);
         this.data = data;
         taken = false;
         ready = true;
   }

   public synchronized int load() {
         while (!ready);
         ready = false;
         taken = true;
         return this.data;
   }
}
```

Solution 3

- We need is to acquire the monitor ***after*** waiting for the flag.

- Therefore, we use the `synchronized` keyword on a code segment (as opposed to an entire method), thus protecting only the *critical code segment* that needs mutual-exclusion.

- When using the synchronized keyword on a code segment, an object to synchronize upon needs to be supplied. This is the object whose monitor is to be used for the critical section.

Solution 3 (cont) – MyData (Sol3_V4)

```
class MyData {
   private int data;
   private boolean ready;
   private boolean taken;
   public MyData() {
        ready = false;
        taken = true;
   }
   public void store(int data) {
        while (!taken);
        synchronized (this) {
                this.data = data;
                taken = false;
                ready = true;
        }
   }
   public int load() {
        while (!ready);
        synchronized (this) {
                ready = false;
                taken = true;
                return this.data;
        }
   }
}
```

Problem:

- the busy-wait loop:
  - It is considered bad practice for a thread to use a busy-wait loop:
  - it is processor-expensive on a preemptive implementation, and might cause deadlock on a non-preemptive one.

- In order to improve thread efficiency and to help avoid deadlock, the following methods are used:
  - `java.lang.Object.wait()`
  - `java.lang.Object.notify()`
  - `java.lang.Object.notifyAll()`

Final Producer/Consumer Solution
MyData (Final_V5)

```
class MyData {
   private int Data;
   private boolean Ready;
   public MyData() {
      Ready=false;
   }
   public synchronized void store(int
   Data) {
      while (Ready)
     try {
         wait();
     } catch (InterruptedException e) {
   }
     this.Data=Data;
     Ready=true;
     notify();
   }
   public synchronized int load() {
       while (!Ready)
       try {
         wait();
   } catch (InterruptedException e) { }
   Ready=false;
   notify();
   return this.Data;
   }
}
```

- The wait() method makes a thread release the monitor and shifts from the *runnable* state to the *non-runnable* state.
- The thread will stay in a non-runnable state until it is waken up by a call to `notify()`.
  - When a thread stops wait()ing, it re-acquires the monitor.
- The `notify`() method arbitrarily chooses a thread from those that are wait()ing and releases it from its wait state.