**Program to print even and odd thread**

**class** Printer {  
  
 **private** AtomicInteger **num** = **new** AtomicInteger(1);  
  
 **public void** printEven() **throws** InterruptedException {  
  
 **synchronized** (**this**) {  
 **while** (**num**.get() < 100) {  
 wait();  
 print();  
 notify();  
 }  
  
 }  
 }  
  
 **private void** print() {  
 System.***out***.println(Thread.*currentThread*().getName() + **"\t"** + **num**.getAndIncrement());  
 }  
  
 **public void** printOdd() **throws** InterruptedException {  
  
 **synchronized** (**this**) {  
 **while** (**num**.get() <= 100) {  
 print();  
 notify();  
 wait();  
 }  
 }  
 }  
}  
  
**public class** OddEvenPrinter {  
  
 **public static void** main(String[] args) {  
  
 Printer printer = **new** Printer();  
  
 **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **try** {  
 printer.printEven();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }, **"Even"**).start();

**new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **try** {  
 printer.printOdd();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }, **"Odd"**).start();  
  
 }  
}

**Executor Service**

**public class** ExecutorServiceDemo {  
  
 **static class** CallableTask **implements** Callable<Integer>  
 {  
  
 **public** Integer call() **throws** Exception {  
  
 **int** timeToSleep = **new** Random().nextInt(20000);  
 System.***out***.println(**"About to sleep for : "**+timeToSleep+**", milli seconds"**);  
  
 **if**(timeToSleep> 20000)  
 {  
 **throw new** IllegalArgumentException(**"Time to sleep is too long"**);  
 }  
  
 Thread.*sleep*(timeToSleep);  
 System.***out***.println(**"Awake from sleep"**);  
 **return** timeToSleep;  
 }  
 }  
  
 **public static void** main(String[] args) {  
  
 **ExecutorService executorService = Executors.*newFixedThreadPool*(1);** Future<Integer> future = executorService.submit(**new** CallableTask());  
  
 System.***out***.println(**"Just submitted callable tasks"**);  
 Integer val = **null**;  
 **try** {  
 val = future.get();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **catch** (ExecutionException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(val);  
  
 executorService.shutdown();  
  
 }  
}

**CountDownLatch**

**import** java.util.concurrent.CountDownLatch;  
**import** java.util.concurrent.ExecutorService;  
**import** java.util.concurrent.Executors;

**public class** CountDownLatchDemo {  
  
 **public static void** main(String[] args) **throws** InterruptedException {  
  
 CountDownLatch latch = **new** CountDownLatch(3);  
  
 ExecutorService executorService = Executors.*newFixedThreadPool*(3);  
 **for** (**int** i = 0; i < 3; i++) {  
 executorService.submit(**new** Processor(latch));  
 }  
 executorService.shutdown();  
 latch.await();  
 System.***out***.println(**"Main Completed"**);  
 }  
}  
**class** Processor **implements** Runnable {  
  
 **private** CountDownLatch **latch**;  
  
 **public** Processor(CountDownLatch latch) {  
 **this**.**latch** = latch;  
 }  
  
 **public void** run() {  
  
 **try** {  
 Thread.*sleep*(10000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 **latch**.countDown();  
 System.***out***.println(**"Completed thread "**+ Thread.*currentThread*().getId());  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

**N Threads printing number (all the thread should print 1, then 2 and so on). Solve using wait and notify**

**public class** NThreadsProblems {  
  
  
 **public static void** main(String[] args) {  
  
 **new** NThreadsProblems().printNumber(3);  
 }  
  
 **public void** printNumber(**int** numberOfThreads) {  
  
 NumberPrinter printer = **new** NumberPrinter(numberOfThreads);  
  
 **for** (**int** i = 0; i < numberOfThreads; i++) {  
  
 **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **try** {  
 printer.print();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }, **"Thread "** + i).start();  
 }  
 }  
  
}  
  
  
**class** NumberPrinter {  
  
 **private volatile int threadCounter**;  
 **private int numberOfThreads**;  
 **private volatile int num** = 1;  
 **private static final int *NUM\_LIMIT*** = 5;  
  
 **public** NumberPrinter(**int** numberOfThreads) {  
 **this**.**numberOfThreads** = numberOfThreads;  
 }  
  
 **public void** print() **throws** InterruptedException {  
  
 **synchronized** (**this**) {  
  
 **while** (**num** <= ***NUM\_LIMIT***) {  
 **threadCounter**++;  
 **if** (**threadCounter** == **numberOfThreads**) {  
 **threadCounter** = 0;  
 notifyAll();  
 System.***out***.println(Thread.*currentThread*().getName() + **"\t"** + **num**++);  
 } **else** {  
 System.***out***.println(Thread.*currentThread*().getName() + **"\t"** + **num**);  
 wait();  
 }  
 }  
 }  
 }  
}

**Implementation of same N thread problem using Cyclic Barrier**

**public class** NThreadsTaskCompletion {  
  
 **private static final int *NO\_OF\_THREADS*** = 10;  
  
 **static class** WorkerThread **implements** Runnable {  
  
 CyclicBarrier **barrier**;  
  
 **public** WorkerThread(CyclicBarrier barrier) {  
 **this**.**barrier** = barrier;  
 }  
  
 @Override  
 **public void** run() {  
  
 **int** printCounter = 1;  
  
 **while** (printCounter <= ***NO\_OF\_THREADS***) {  
 System.***out***.println(**"Thread Name : "** + Thread.*currentThread*().getName() + **" "** + printCounter++);  
 **try** {  
 **barrier**.await();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **catch** (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
  
 **public static void** main(String[] args) {  
  
 CyclicBarrier barrier = **new** CyclicBarrier(***NO\_OF\_THREADS***);  
 ExecutorService executorService = Executors.*newFixedThreadPool*(***NO\_OF\_THREADS***);  
  
 **for** (**int** i = 0; i < ***NO\_OF\_THREADS***; i++) {  
 executorService.submit(**new** WorkerThread(barrier));  
 }  
 executorService.shutdown();  
 }  
}

**Producer Consumer using BlockingQueue**

**public class** ProducerConsumerBlockingQueue {  
  
 **private static** BlockingQueue<Integer> *queue* = **new** ArrayBlockingQueue<Integer>(10);  
  
 **static void** produce() **throws** InterruptedException {  
  
 Random random = **new** Random();  
 **while** (**true**) {  
 *queue*.put(random.nextInt(100));  
 }  
 }  
  
 **static void** consume() **throws** InterruptedException {  
  
 Random random = **new** Random();  
 **while** (**true**) {  
 Thread.*sleep*(100);  
 **if** (random.nextInt(10) == 0) {  
 Integer valueTaken = *queue*.take();  
 System.***out***.println(**"Queue Size "** + *queue*.size() + **", Value processing by consume : "** + valueTaken);  
 }  
  
 }  
 }  
  
 **public static void** main(String[] args) {  
  
 **new** Thread(**new** Runnable() {  
 **public void** run() {  
 **try** {  
 *produce*();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }).start();  
  
 **new** Thread(**new** Runnable() {  
 **public void** run() {  
 **try** {  
 *consume*();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }).start();  
  
 }  
}

**Print numbers in sequence by N threads, for eg: 1st thread will print 1, second will print 2 and so on**

**public class** SequencePrinter {  
  
 **public static void** main(String[] args) {  
 **new** SequencePrinter().printInSequence(4);  
  
 }  
  
 **public void** printInSequence(**int** numOfThreads) {  
  
 ArrayList<Worker> threadList = **new** ArrayList<>();  
  
 Worker previous = **null**;  
 Worker current = **null**;  
 Worker first = **null**;  
  
 *// Create the workers and chain them in a round robin fashion* **for** (**int** i = 0; i < numOfThreads; i++) {  
 current = **new** Worker();  
 **if** (previous != **null**) {  
 previous.setNext(current);  
 } **else** {  
 first = current;  
 }  
 previous = current;  
 threadList.add(current);  
 }  
 current.**next** = first;  
  
 **for** (**int** i = 1; i <= numOfThreads; i++) {  
 Thread t = **new** Thread(threadList.get(i-1),**"Thread -"**+i);  
 t.start();  
 }  
  
 *// Seed the first current* first.accept(1);  
 }  
}  
  
**class** Worker **implements** Runnable {  
  
 BlockingQueue<Integer> **q** = **new** LinkedBlockingQueue<Integer>();  
 Worker **next** = **null**; *// next worker in the chain* **public void** setNext(Worker t) {  
 **this**.**next** = t;  
 }  
  
 **public void** accept(**int** i) {  
 **q**.add(i);  
 }  
  
 @Override  
 **public void** run() {  
 **while** (**true**) {  
 **try** {  
 **int** i = **q**.take(); *// blocks till it receives a number* System.***out***.println(Thread.*currentThread*().getName() +**"\t"**+ i);  
  
 Thread.*sleep*(1000); *// delay to slow the printing* **if** (**next** != **null**) {  
 **next**.accept(i + 1); *// pass the next number to the next worker* }  
  
 } **catch** (InterruptedException e) {  
 System.***err***.println(Thread.*currentThread*().getName() + **" interrrupted."**);  
 }  
 }  
 }  
}

**Deadlock Demo**

**public class** DeadlockDemo {  
  
 **private** Object **lock1** = **new** Object();  
 **private** Object **lock2** = **new** Object();  
  
 **public static void** main(String[] args) {  
 **new** DeadlockDemo().startThreads();  
 }  
  
 **private void** startThreads() {  
 **new** Thread(**new** Runnable() {  
 **public void** run() {  
  
 **synchronized** (**lock1**)  
 {  
 System.***out***.println(**"Acquired lock1 by first thread"**);  
  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 **synchronized** (**lock2**)  
 {  
 System.***out***.println(**"Acquired lock2 by first thread "**) ;  
 }  
 }  
 }  
 }).start();  
  
 **new** Thread(**new** Runnable() {  
 **public void** run() {  
  
 **synchronized** (**lock2**)  
 {  
 System.***out***.println(**"Acquired lock2 by second thread"**);  
  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 **synchronized** (**lock1**)  
 {  
 System.***out***.println(**"Acquired lock1 by second thread "**) ;  
 }  
 }  
 }  
 }).start();  
 }  
  
  
}

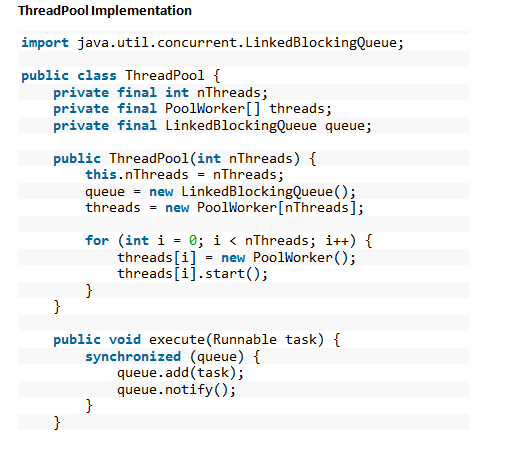
**Producer Consumer using wait notify**

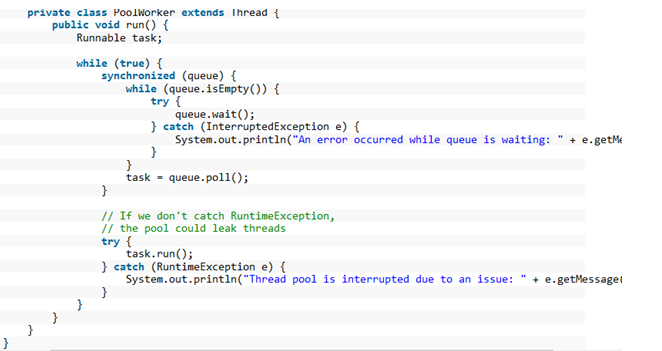
**public class** Dropbox {  
  
 **private int number**;  
 **private boolean empty** = **true**;  
 **private boolean evenNumber** = **false**;  
  
 **public synchronized int** take(**final boolean** even) {  
 **while** (**empty** || **evenNumber** != even) {  
 **try** {  
 System.***out***.format(**"%s is waiting ... %n"**, even ? **"Even"** : **"Odd"**);  
 wait();  
 } **catch** (InterruptedException e) { }  
 }  
 System.***out***.format(**"%s took %d.%n"**, even ? **"Even"** : **"Odd"**, **number**);  
 **empty** = **true**;  
 notifyAll();  
  
 **return number**;  
 }  
  
 **public synchronized void** put(**int** number) {  
 **while** (!**empty**) {  
 **try** {  
 System.***out***.println(**"Producer is waiting ..."**);  
 wait();  
 } **catch** (InterruptedException e) { }  
 }  
 **this**.**number** = number;  
 **evenNumber** = number % 2 == 0;  
 System.***out***.format(**"Producer put %d.%n"**, number);  
 **empty** = **false**;  
 */\*\*  
 \* This will demonstrate the usage of notifyAll, change notifyAll to notify and see the difference  
 \*/* notifyAll();  
 }  
 }

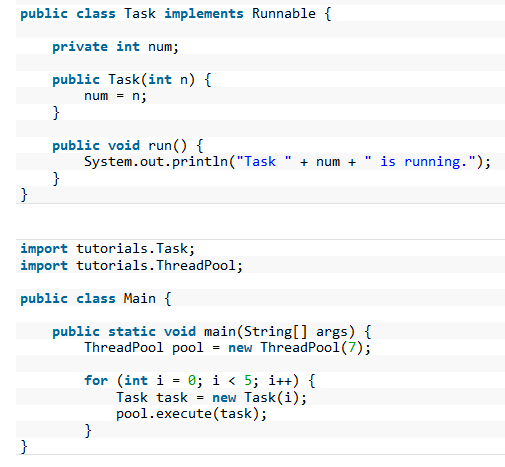
**public class** Producer **implements** Runnable {  
  
 **private** Dropbox **dropbox**;  
  
 **public** Producer(Dropbox dropbox) {  
 **this**.**dropbox** = dropbox;  
 }  
  
 **public void** run() {  
 Random random = **new** Random();  
 **while** (**true**) {  
 **int** number = random.nextInt(10);  
 **try** {  
 Thread.*sleep*(random.nextInt(100));  
 **dropbox**.put(number);  
 } **catch** (InterruptedException e) { }  
 }  
 }  
}

**public class** Consumer **implements** Runnable {  
  
 **private final** Dropbox **dropbox**;  
 **private final boolean even**;  
  
 **public** Consumer(**boolean** even, Dropbox dropbox) {  
 **this**.**even** = even;  
 **this**.**dropbox** = dropbox;  
 }  
  
 **public void** run() {  
 Random random = **new** Random();  
 **while** (**true**) {  
 **dropbox**.take(**even**);  
 **try** {  
 Thread.*sleep*(random.nextInt(100));  
 } **catch** (InterruptedException e) { }  
 }  
 }  
}

**public class** ProducerConsumerExample {  
 **private static boolean** *Even* = **true**;  
 **private static boolean** *Odd* = **false**;  
  
 **public static void** main(String[] args) {  
 Dropbox dropbox = **new** Dropbox();  
 (**new** Thread(**new** Consumer(*Even*, dropbox))).start();  
 (**new** Thread(**new** Consumer(*Odd*, dropbox))).start();  
 (**new** Thread(**new** Producer(dropbox))).start();  
 }  
}







Use **put()** and **take()** method instead of **add()** and **poll()** so that we can get rid of synchronization**.**