Lab 5 - Threads, Synchronization, Thread Communications - wait/notify

This lab contains in-class exercises related to threads, synchronization, and thread communications - wait/notify

Task 1: Develop a class `BankCredit` with a method called `creditAccount` that can be used by many customers. Your customers are Chan and John.

Your `BankCredit` must be an object of the Runnable type and must be used as follows:

```
// Create a Runnable object
BankCredit tbc = new BankCredit(0);

// Create two threads which share the same bank credit
Thread th1 = new Thread(tbc, "John ");
Thread th2 = new Thread(tbc, "Chan ");
```

Here is a possible implementation:

```
public class BankCredit /* ??? */ {

   private int balance;

   // BankCredit constructor
   public BankCredit(int balance) {
       this.balance = balance;
   }

   // Implement run() method of Runnable to execute a new thread
   public void run() {

       // Get current active thread
       /* ??? */

       System.out.println(activeThread.getName() + " is done.");
   }

   // Credit account
   public void creditAccount(int credit) {

       Thread activeThread = Thread.currentThread();
       System.out.println(activeThread.getName() + " opening creditAccount().");


       /* ??? */

       System.out.println("Account balance is now: " + balance);
   }

   public static void main(String[] args) {
```

```
        // Create a Runnable object
        BankCredit tbc = new BankCredit(0);

        // Create two threads which share the same bank credit
        Thread th1 = new Thread(tbc, "John ");
        Thread th2 = new Thread(tbc, "Chan ");

        // Power up the threads via the start() method
        /* ??? */
        /* ??? */
    }
}
```

Fill out `/* ??? */` with missing code.

The output of your program should be similar to this one:

```
John  wants access
Chan  wants access
John  opening creditAccount().
Chan  opening creditAccount().
John  about to deposit 10 canadian dollars.
Account balance is now: 10
Chan  about to deposit 10 canadian dollars.
John  opening creditAccount().
Account balance is now: 20
John  about to deposit 10 canadian dollars.
Chan  opening creditAccount().
Account balance is now: 30
Chan  about to deposit 10 canadian dollars.
John  is done.
Account balance is now: 40
Chan  is done.
```

Task 2: Design and develop a program that solves the classical problem Producer-Consumer

The problem describes two objects, the producer and the consumer, who share a common, fixed-size container.

The producer's job is to generate data, one piece at a time, and put it into the container. At the same time, the consumer is consuming the data (i.e., removing it from the container), one piece at a time.

The problem is to make sure that the producer won't try to add data into the container if it has an object in it and that the consumer won't try to remove data from an empty container.

```
public class SharedResource {

   public static void main(String[] args) {
       if (args.length == 0) {
           System.out.println("usage: java SharedResource <number of resources>");
           System.exit(0);
       }

       //indicates how many integer will be generated and consumed
       int numberOfResources = 1;
       try {
           numberOfResources = Integer.parseInt(args[0]);
           if (numberOfResources < 0) System.exit(-1);
       } catch (NumberFormatException e) {
           System.out.println("Input: [" + args[0] + "] must be an integer number.");
           System.exit(-1);
       }

       //The shared container containing a shared int
       Container container = new Container();

       ProduceResource p = new ProduceResource(container, numberOfResources);
       ConsumeResource c = new ConsumeResource(container, numberOfResources);

       // Start the theads
       /* ??? */
   }
}

/**
* Definition of a Thread class ProduceResource
*/
class ProduceResource extends Thread {

   private Container container;
   private int numberOfResources;

   public ProduceResource(Container c, int n) {
       super("Producer of Resource");
       container = c;
       numberOfResources = n;
   }

   public void run() {
       for (int count = 1; count <= numberOfResources; count++) {
           // sleep for a random interval
           try {
               Thread.sleep((int) (Math.random() * 3000));
           } catch (InterruptedException e) {
               System.err.println(e.toString());
```

```
            }

           /* ??? */
        }

    }
}

/**
* Definition of a Thread class ConsumeResource
*/
class ConsumeResource extends Thread {

    private Container container;
    private int numberOfResources;

    public ConsumeResource(Container c, int m) {
        super("Consumer of Resource");
        container = c;
        numberOfResources = m;
    }

    public void run() {
        /* ??? */

    }
}

/**
* Definition of a Container
*/
class Container {
    private int sharedInt = 0;
    private boolean writeable = true;  // condition variable

    /**
     * synchronized method for setting the resource:
     * The code in the get method loops until the Producer has produced a new value.
     * Each time through the loop, the get method calls the wait method;
     * The wait method relinquishes the lock held by the Consumer on the Container
     * (thereby allowing the Producer to get the lock and update the Container)
     * and then waits for notification from the Producer.
     */
    public synchronized void setSharedResource(int val) {
        /* ??? */
        System.err.println(Thread.currentThread().getName() +
                " generates Resource value number = " + val);
        sharedInt = val;
        writeable = false;
        notify();  // tell a waiting thread to become ready
```

```
}

/**
 * synchronized method for getting the resource:
 * When the Producer puts something in the Container,
 * it notifies the Consumer by calling notifyAll.
 * The Consumer then comes out of the wait state, the loop exits,
 * and the get method returns the value in the Container.
 * The set method works in a similar fashion, waiting for the Consumer thread to consume
 * the current value before allowing the Producer to produce a new one.
 */
public synchronized int getSharedResource() {
    /* ??? */

    writeable = true;
    notify();  // tell a waiting thread to become ready
    System.err.println(Thread.currentThread().getName() +
            " uses Resource value number = " + sharedInt + "\n");
    return sharedInt;
}
```

Fill out `/* ??? */` with missing code.

Task 3: Change your classes, so that you can store generic objects of type **T** instead of only integers.

```
public class SharedResource<T> { … }

class ProduceResource<T> extends Thread { … }

class ConsumeResource<T> extends Thread { … }

class Container<T> { … }
```