Sample Answer

# Q1:

3, 2, 3, 2, 1, 4, 1, 2, (First, second, third quadrant), 4

# Q2:

One thread tries to set the color to 0, 0, 0, another tries to set it to 255, 255, 255; the result might be 0, 255, 255.

(b)

public class RGBColor {

private int r;

private int g;

private int b;

public RGBColor(int r, int g, int b) {

checkRGBVals(r, g, b);

this.r = r;

this.g = g;

this.b = b;

}

public synchronized void setColor(int r, int g, int b) {

checkRGBVals(r, g, b);

this.r = r;

this.g = g;

this.b = b;

}

public synchronized int[] getColor() {

int[] retVal = new int[3];

retVal[0] = r;

retVal[1] = g;

retVal[2] = b;

return retVal;

}

public synchronized void invert() {

r = 255 - r;

g = 255 - g;

b = 255 - b;

}

private static void checkRGBVals(int r, int g, int b) {

if (r < 0 || r > 255 || g < 0 || g > 255 ||

b < 0 || b > 255) {

throw new IllegalArgumentException();

}

}

(c)

public class SafeRGBColor {

private RGBColor color;

public SafeRGBColor(int r, int g, int b) {

color = new RGBColor(r, g, b);

}

public synchronized void setColor(int r, int g, int b) {

color.setColor(r, g, b);

}

/\*\*

\* returns color in an array of three ints: R, G, and B

\*/

public synchronized int[] getColor() {

return color.getColor();

}

public synchronized void invert() {

color.invert();

}

}

# Q3:

class CountDownLatchCustom{

private int count;

/\*\*

\* CountDownLatch is initialized with given count.

\* count specifies the number of events that must occur

\* before latch is released.

\*/

public CountDownLatchCustom(int count) {

this.count=count;

}

/\*\*

\* Causes the current thread to wait until one of the following things happens-

- latch count has down to reached 0, or

- unless the thread is interrupted.

\*/

public synchronized void await() throws InterruptedException {

//If count is greater than 0, thread waits.

if(count>0)

this.wait();

}

/\*\*

\* Reduces latch count by 1.

\* If count reaches 0, all waiting threads are released.

\*/

public synchronized void countDown() {

//decrement the count by 1.

count--;

//If count is equal to 0, notify all waiting threads.

if(count == 0)

this.notifyAll();

}

}

# Q4:

(a)

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class testInterface2 {

public static void main (String [] args) throws Exception {

ExecutorService exec = Executors.newFixedThreadPool(2);

//TODO

final Cup mine = new Cup (50);

final Cup yours = new Cup (50);

exec.execute(new Runnable() {

public void run() {

//TODO

mine.pour(yours, 20);

}

}

);

exec.execute(new Runnable() {

public void run() {

//TODO:

yours.pour(mine, 20);

}

}

);

}

}

class Cup {

private int water = 0;

public Cup (int i) {

water = i;

}

public synchronized void top (int quantify) {

water = water + quantify;

}

public synchronized void pour (int quantify) {

water = water - quantify;

}

public synchronized void pour (Cup other, int quantify) {

pour(quantify);

other.top(quantify);

}

}

(b)

class MyCup {

private int water = 0;

private final ReentrantLock tryLock = new ReentrantLock();

public MyCup (int i) {

water = i;

}

public void top (int quantify) {

while (tryLock.tryLock()) {

water = water + quantify;

}

}

public void pour (int quantify) {

while (tryLock.tryLock()) {

water = water - quantify;

}

}

public void pour (MyCup other, int quantify) {

while (tryLock.tryLock()) {

pour(quantify);

other.top(quantify);

}

}

}

/\*class MyCup {

private int water = 0;

private static final Object tieLock = new Object();

public MyCup (int i) {

water = i;

}

public synchronized void top (int quantify) {

water = water + quantify;

}

public synchronized void pour (int quantify) {

water = water - quantify;

}

public void pour (MyCup other, int quantify) {

int myHash = System.identityHashCode(this);

int otherHash = System.identityHashCode(other);

if (myHash < otherHash) {

pour(quantify);

other.top(quantify);

}

else if (myHash > otherHash) {

other.top(quantify);

pour(quantify);

}

else {

synchronized(tieLock) {

pour(quantify);

other.top(quantify);

}

}

}

}\*/

# Q5:

(a)

private final Object a = new Object ();

private final Object b = new Object ();

public void methodA() {

synchronized (a) {

somethingA();

}

}

public void methodB() {

synchronized (b) {

somethingB();

}

}

(b)

private final AtomicInteger a = new AtomicInteger ();

private final object x;

public void methodA() {

syn{

flag = a.addAndGet(1)>0

}

if (a.addAndGet(1) > 0) {

somethingA();

a.addAndGet(-1)

}

}

public void methodB() {

if (a.addAndGet(-1) < 0) {

somethingB();

}

}

(c)

private final Semaphore a = new Semaphore (10);

public void methodA() throws InterruptedException {

a.acquire();

try {

somethingA();

}

finally {

a.release();

}

}

public void methodB() throws InterruptedException {

a.acquire();

try {

somethingB();

}

finally {

a.release();

}

}