BSCCS2005: Practice with Solutions
Week 11

1. Consider the code given below.

```
public class PrlSequence1 extends Thread{
    public void run(){
        try {
            for(int i = 11; i <= 20; i++){
                if(isInterrupted())
                    break;
                System.out.print(i + " ");
                Thread.sleep(500);
            }
        }catch(InterruptedException e) {}
    }
}

public class PrlSequence2 extends Thread{
    Thread other;
    PrlSequence2(Thread t){
        other = t;
    }
    public void run(){
        try {
            for(int i = 21; i <= 30; i++){
                System.out.print(i + " ");
                if(i == 25)
                    other.interrupt();
                Thread.sleep(500);
            }
        }
        catch(InterruptedException e) {}
    }
}

public class FClass{
    public static void main(String[] args) throws InterruptedException{
        Thread th1 = new PrlSequence1();
        Thread th2 = new PrlSequence2(th1);
        th1.start();
        th2.start();
    }
}
```

Choose the correct option regarding the code.

- ◯ Thread `th1` prints 11 to 20, and thread `th2` prints 21 to 30 in an interleaved way.
- ◯ The thread `th1` first prints 11 to 20, and followed by the thread `th2` prints 21 to 30.
- ◯ The thread `th1`, and the thread `th2` initially may print the numbers in an interleaved way. However, the thread `th2` gets terminated after printing 25
- √ The thread `th1`, and the thread `th2` initially may print the numbers in an interleaved way. However, the thread `th2` after printing 25, causes thread `th1` to terminate.

Solution:

2. Consider the following code which tries to simulate a producer-consumer relationship on cakes.

```java
class CakeOperation {
    int available_cake_count=0;

    synchronized void eatCake() {
        available_cake_count--;
        System.out.println("Ate one cake. Cakes left: "+ available_cake_count);
    }
    synchronized void makeCake() {
        available_cake_count++;
        System.out.println("Made one cake. Cakes left: "+ available_cake_count);
    }
}
class Baker implements Runnable {
    CakeOperation obj;
    Baker(CakeOperation o) {
        obj = o;
        Thread Producer = new Thread(this);
        Producer.start();
    }
    public void run() {
        int cake_number = 1;
        while(cake_number <= 2) {
            obj.makeCake();
            cake_number++;
        }
    }
}
class Consumer implements Runnable {
    CakeOperation obj;
    Consumer(CakeOperation o) {
        obj = o;
        Thread consumer = new Thread(this);
        consumer.start();
    }
    public void run() {
        int iteration = 0;
        while(iteration < 2) {
            obj.eatCake();
            iteration++;
        }
    }
}
public class Test {
    public static void main(String args[]) {
        CakeOperation obj  = new CakeOperation();
        Baker b1 = new Baker(obj);
        Consumer c1 = new Consumer(obj);
    }
}
```

Which of the following options is/are NOT possible result/s of the above code?

✓
```
Made one cake. Cakes left: 1
Made one cake. Cakes left: 1
Ate one cake. Cakes left: 0
Ate one cake. Cakes left: 0
```

○
```
Made one cake. Cakes left: 1
Ate one cake. Cakes left: 0
Made one cake. Cakes left: 1
Ate one cake. Cakes left: 0
```

○
```
Made one cake. Cakes left: 1
Ate one cake. Cakes left: 0
Ate one cake. Cakes left: -1
Made one cake. Cakes left: 0
```

○
```
Made one cake. Cakes left: 1
Made one cake. Cakes left: 2
Ate one cake. Cakes left: 1
Ate one cake. Cakes left: 0
```

Solution: Detailed sol goes here

3. Consider the code given below.

```
class Bank{
    int total = 100;
    synchronized void extract(String name,int withdrawal){
        if (total >= withdrawal){
            System.out.println(name + " has withdrawn " + withdrawal);
            total = total - withdrawal;
        }
        else {
            System.out.println(name + " can not withdraw "+ withdrawal);
            System.out.println("Current balance is: " + total);
        }
    }
}
class Withdrawal implements Runnable{
    Bank bk;
    String name;
    int money;
    Withdrawal(Bank b, String n, int m){
        this.bk = b;
        this.name = n;
        this.money = m;
    }
    public void run(){
        bk.extract(name, money);
    }
}
public class Test {
    public static void main(String[] args){
        Bank obj = new Bank();
        Thread t1 = new Thread(new Withdrawal(obj, "Sun", 20));
        Thread t2 = new Thread(new Withdrawal(obj, "Moon", 40));
        Thread t3 = new Thread(new Withdrawal(obj, "Earth", 80));
        t1.start();
        t2.start();
        t3.start();
    }
}
```

What is/are the possible output/s?

√
```
Sun has withdrawn 20
Earth has withdrawn 80
Moon can not withdraw 40
Current balance is: 0
```

√
```
Sun has withdrawn 20
Moon has withdrawn 40
Earth can not withdraw 80
Current balance is: 40
```

○
```
Moon has withdrawn 40
Sun has withdrawn 20
Earth has withdrawn 80
```

○ This program generates `IllegalMonitorStateException` at runtime.

Solution: The non `static synchronization` provides an object level lock and allows only 1 thread at a time to enter the critical region. An object can have multiple threads but the sensitive area can only be accessed by 1 thread at a time.

4. Consider the code given below.

```
class ThExample extends Thread{
    int a = 1, b = 1, c = 0;
    public void run(){
       synchronized(this){
           for(int n = 0; n < 5; n++){
               c = a + b;
               a = b;
               b = c;
            }
            this.notify();
        }
    }
}

public class FClass{
    public static void main(String[] args) throws InterruptedException{
        ThExample t = new ThExample();
        t.start();
        synchronized(t){
            t.wait();
            System.out.println(t.c);
        }

     }
}
```

Choose the correct option regarding the given code.

- [✓ This code may generate the output `13`]
- ◯ This code may generate the output `8`
- ◯ This code may generate the output `0`
- ◯ This code may generate any Fibonacci number within the closed interval [2, 13].

5. Consider the code given below.

```
import java.util.concurrent.locks.*;
class Bank{
    ReentrantLock lck = new ReentrantLock();
    int balance = 100;
    void extract(String name,int withdrawal){
        lck.lock();
        try{
            if (balance>=withdrawal){
                System.out.println(name + " has withdrawn " + withdrawal);
                balance = balance - withdrawal;
            }
            else {
                System.out.println(name + " cannot withdraw " + withdrawal);
                System.out.println("Because current balance is " + balance);
            }
        }
        finally{
            lck.unlock();
        }
    }
}
class Withdrawal implements Runnable{
    Bank l_obj;
    String name;
    int money;
    Withdrawal(Bank o, String n, int m){
        this.l_obj = o;
        this.name = n;
        this.money = m;
    }
    public void run(){
        l_obj.extract(name, money);
    }
}
public class FClass{
    public static void main(String[] args){
        Bank obj = new Bank();
        Thread t1 = new Thread(new Withdrawal(obj, "Sun", 20));
        Thread t2 = new Thread(new Withdrawal(obj, "Moon", 40));
        Thread t3 = new Thread(new Withdrawal(obj, "Earth", 80));
        t1.start();
        t2.start();
        t3.start();
    }
}
```

What is/are the possible output/s?

✓
```
Sun has withdrawn 20
Earth has withdrawn 80
Moon cannot withdraw 40
Because current balance is 0
```

◯
```
Earth has withdrawn 80
Moon has withdrawn 40
Sun has withdrawn 20
```

✓
```
Sun has withdrawn 20
Moon has withdrawn 40
Earth cannot withdraw 80
Because current balance is 40
```

◯
```
Moon has withdrawn 40
Sun has withdrawn 20
Earth has withdrawn 80
```

6. Consider the code given below.

```
public class Test{
    public static void main(String[] args){
        Thread t=Thread.currentThread();
        t.interrupt();
        System.out.println(Thread.interrupted());
        while(t.isInterrupted()){
            System.out.println("Infinite");
        }
    }
}
```

What will the output be?

- √ `true`
- ◯ `false`
- ◯ `true`
  `Infinite`
- ◯ This program prints `true` followed by infinite loops printing `Infinite`.

Solution: The `interrupted` method prints the interruption status and change it to `false` if the status is `true`.

7. Consider the code given below.

```
import java.util.*;
class PrlTest extends Thread{
    Map<String, Integer> icMap;
    public PrlTest(Map<String, Integer> ic){
        this.icMap = ic;
    }
    public void run(){
        icMap.put("D", 4);
    }
}
public class FClass{
    public static void main (String[] args) throws InterruptedException{
        Thread t1 = Thread.currentThread();
        Map<String, Integer> icMap = new LinkedHashMap<String, Integer>();
        String[] str = {"A", "B", "C"};
        Integer[] arr = {1, 2, 3};
        for(int i = 0; i < str.length; i++){
            icMap.put(str[i], arr[i]);
        }
        PrlTest t2 = new PrlTest(icMap);
        t2.start();
        t2.join();
        for(Map.Entry m : icMap.entrySet()){
            System.out.println(m.getKey() + " => "+ m.getValue());
        }
    }
}
```

Choose the correct option regarding the code.

- ○ This program prints
  ```
  A => 1
  B => 2
  C => 3
  ```
- ○ This program results in a deadlock.
- ✓ This program prints
  ```
  A => 1
  B => 2
  C => 3
  D => 4
  ```
- ○ This program prints all 4 elements: `A => 1, B => 2, C => 3` and `D => 4`. However, the order of the elements can't be undetermined.

8. Consider the code given below.

```
import java.util.*;
class Example extends Thread{
    Map mp;
    Example(Map m){
        this.mp = m;
    }
    public void run(){
        mp.put("D",4);
    }
}
public class Test{
    public static void main (String[] args){
        Map<String, Integer> map = new LinkedHashMap();
        String[] str = {"A", "B", "C"};
        Integer[] arr = {1, 2, 3};
        for(int i = 0;i < str.length; i++){
           map.put(str[i],arr[i]);
        }
        Example t = new Example(map);
        t.start();
        Set s = map.entrySet();
        Iterator itr = s.iterator();
        while(itr.hasNext()){
           Map.Entry m = (Map.Entry)itr.next();
           System.out.println(m.getKey() + " => " + m.getValue());
        }
    }
}
```

Which of the following is NOT true about the given code.

- ○ This program may generate `ConcurrentModificationException`.
- ○ This program may generate the output
```
A => 1
B => 2
C => 3
```
- ○ This program may generate the output
```
A => 1
B => 2
C => 3
D => 4
```

√ This program may generate the output

```
D => 4
A => 1
B => 2
C => 3
```

9. In the following program, there are two user-defined threads named `Dijkstra` and `Turing`. The program prints different possible thread states that each of these threads can be in, during the execution of program.

```
import java.util.*;
public class ThreadState implements Runnable   {

    public synchronized void run() {
        try {
            Thread.sleep(800);
        }catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName()+
                        " is inside run method but not sleeping");
    }
    public static void main(String args[]) throws InterruptedException  {
        ThreadState obj = new ThreadState();
        Thread t1= new Thread(obj,"Dijkstra");
        Thread t2= new Thread(obj,"Turing");
        Thread.currentThread().setPriority(10);
        System.out.println("Dijkstra before calling start(): "+t1.getState());
        System.out.println("Turing before calling start(): "+t2.getState());
        t1.start();
        t2.start();
        System.out.println("Dijkstra after calling start(): "+t1.getState());
        System.out.println("Turing after calling start(): "+t2.getState());

        Thread.sleep(200);
        System.out.println("Dijkstra before joining: "+t1.getState());
        System.out.println("Turing before joining: "+t2.getState());

        t1.join();
        t2.join();
        System.out.println("Dijkstra after joining: "+t1.getState());
        System.out.println("Turing after joining: "+t2.getState());
    }
}
```

Note:
1. A thread remains in `TIMED_WAITING` state when it is sleeping.
2. A thread is in `BLOCKED` state when it is waiting for the object lock to be released by some other thread which is using that object.

Choose all the options corresponding to an output, that would NEVER occur as a result of this program.

√
```
Dijkstra before calling start(): NEW
Turing before calling start(): NEW
Dijkstra after calling start(): RUNNABLE
Turing after calling start(): RUNNABLE
Dijkstra before joining: TIMED_WAITING
Turing before joining: TIMED_WAITING
Turing is inside run method but not sleeping
Dijkstra is inside run method but not sleeping
Dijkstra after joining: TERMINATED
Turing after joining: TERMINATED
```

○
```
Dijkstra before calling start(): NEW
Turing before calling start(): NEW
Dijkstra after calling start(): RUNNABLE
Turing after calling start(): RUNNABLE
Dijkstra before joining: TIMED_WAITING
Turing before joining: BLOCKED
Dijkstra is inside run method but not sleeping
Turing is inside run method but not sleeping
Dijkstra after joining: TERMINATED
Turing after joining: TERMINATED
```

○
```
Dijkstra before calling start(): NEW
Turing before calling start(): NEW
Dijkstra after calling start(): RUNNABLE
Turing after calling start(): RUNNABLE
Dijkstra before joining: BLOCKED
Turing before joining: TIMED_WAITING
Turing is inside run method but not sleeping
Dijkstra is inside run method but not sleeping
Dijkstra after joining: TERMINATED
Turing after joining: TERMINATED
```

○
```
Dijkstra before calling start(): NEW
Turing before calling start(): NEW
Dijkstra after calling start(): TIMED_WAITING
Turing after calling start(): BLOCKED
Dijkstra before joining: TIMED_WAITING
Turing before joining: BLOCKED
Dijkstra is inside run method but not sleeping
Turing is inside run method but not sleeping
Dijkstra after joining: TERMINATED
Turing after joining: TERMINATED
```

◯
```
Dijkstra before calling start(): NEW
Turing before calling start(): NEW
Dijkstra after calling start(): TIMED_WAITING
Turing after calling start(): RUNNABLE
Dijkstra before joining: TIMED_WAITING
Turing before joining: BLOCKED
Dijkstra is inside run method but not sleeping
Turing is inside run method but not sleeping
Dijkstra after joining: TERMINATED
Turing after joining: TERMINATED
```

Solution:
A thread is in `TIMED_WAITING` state when it is sleeping, the only part of code where either `Dijkstra` or `Turing` can execute a `Thread.sleep()` instruction, is inside the `run` method. Now, in the first option both the threads are together in `TIMED_WAITING` state, which means both have been inside the `run` method together and thus mutual exclusion couldn't be achieved, but `run` method is synchronized in this example and hence it must guarantee mutual exclusion on an object of `ThreadState` class.

Thus we reach at a contradiction which clearly indicates that the output given in option 1 can never occur.