BSCCS2005: Graded with Solutions
Week 11

1. Consider the code given below.

```
public class PrlSequence extends Thread{
    int init;
    public PrlSequence(int i) {
        init = i;
    }
    public void run(){
        for(int i = init; i <= init + 5; i++) {
            System.out.print(i + " ");
            try {
                sleep(500);
            }
            catch(InterruptedException e) {}
        }
    }
}

public class FClass{
    public static void main(String[] args) throws InterruptedException{
        Thread th1 = new  PrlSequence(10);
        Thread th2 = new  PrlSequence(20);
        Thread th3 = new  PrlSequence(30);
        th1.start();
        th2.start();
        th1.join();
        th2.join();
        th3.start();
    }
}
```

Choose the correct option regarding the code.

- ○ It may print 10 to 15, 20 to 25, and 30 to 35 in an interleaved manner.
- [✓ It may print 10 to 15 and 20 to 25, in an interleaved manner, followed by printing 30 to 35.]
- ○ It prints 10 to 15 first, followed by 20 to 25, and followed by printing 30 to 35.
- ○ It prints 10 to 15 first, followed by 20 to 25, and 30 to 35 in an interleaved manner.

Solution: The `th1.join()` method ensures that the main thread wait for `th1` to complete,

The `th2.join()` method ensures that the main thread wait for `th2` to complete, then starts `th3`.
Thus, it prints 10 to 15 and 20 to 25 in interleaved fashion, and then prints 30 to 35.

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

```
class CakeOperation {
    int available_cake_count=0;
    boolean eating_permit = false;

    synchronized void eatCake() {
        while(eating_permit == false) {
            try {
             wait();
            }catch(InterruptedException e) {}
        }
        available_cake_count--;
        System.out.println("Ate one cake. Cakes left: "+ available_cake_count);

        // SEGMENT 1
    }
    synchronized void makeCake() {

        // SEGMENT 2

        available_cake_count++;
        System.out.println("Made one cake. Cakes left: "+ available_cake_count);
        eating_permit = true;
        notify();
    }
}
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 Test2 {
    public static void main(String args[]) {
        CakeOperation obj  = new CakeOperation();
        Baker b1 = new Baker(obj);
        Consumer c1 = new Consumer(obj);
    }
}
```

If the given code always generates the following output then what should be written in place of `SEGMENT 1` and `SEGMENT 2` ?

Output

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

SEGMENT 1:

```
√ eating_permit = false;
  notify();
```

SEGMENT 2:

```
if(eating_permit == true) {
    try {
        wait();
    }catch(InterruptedException e) {}
}
```

◯ *SEGMENT 1:*
```
eating_permit = true;
notify();
```

SEGMENT 2:
```
if(eating_permit == true) {
    try {
        wait();
    }catch(InterruptedException e) {}
}
```

◯ *SEGMENT 1:*
```
eating_permit = false;
notify();
```

SEGMENT 2:
```
if(eating_permit == false) {
    try {
        wait();
    }catch(InterruptedException e) {}
}
```

◯ *SEGMENT 1* and *SEGMENT 2* can be left blank, the code will produce the required output because the methods are synchronized.

Solution: Detailed sol

3. Consider the following code which converts temperatures given in Centigrade scale to Fahrenheit scale and vice versa, using two threads. Once the conversion is finished, the program should check whether there is a temperature value in the array which is the same in both the scales.

```
import java.util.*;
class Centigrade extends Thread{
    double[] f_temp = null;
    public Centigrade(double[] arr) {
        f_temp = arr;
    }
    public void run() {
        // converts the temperatures in f_temp to centigrade
        for(int i=0;i<f_temp.length;i++) {
            f_temp[i] = ((f_temp[i] - 32)/9.0)*5;
        }
    }
}
class Fahrenheit extends Thread{
    double[] c_temp = null;
    public Fahrenheit(double[] arr) {
        c_temp = arr;
    }
    public void run() {
        // converts the temperatures in c_temp to fahrenheit
        for(int i=0;i<c_temp.length;i++) {
            c_temp[i] = ((9.0/5.0)*c_temp[i])+32.0;
        }
    }
}
public class Test{
    public static void main(String args[]) {
        double temp_in_fahrenheit[] = {29.67, -40.0, 50.0, 32.0};
        double temp_in_centigrade[] = {100.0, -92.54, 32.0, -40.0};
        Centigrade cobj = new Centigrade(temp_in_fahrenheit);
        Fahrenheit fobj = new Fahrenheit(temp_in_centigrade);
        cobj.start();
        fobj.start();

        //***** SEGMENT 1 ******

        // Checking if a temperature is same in both centigrade and fahrenheit
        for(int i=0;i<4;i++) {
            for(int j=0;j<4;j++) {
                if(temp_in_fahrenheit[i] == temp_in_centigrade[j])
                    System.out.println(temp_in_centigrade[j]);
            }
        }
    }
}
```

- 40 is the only temperature which is same in both the scales. Choose all the correct options which should be written in place of SEGMENT 1, so that this program produces the result as -40 on every execution.

[✓
```
try {
    cobj.join();
    fobj.join();
}catch(InterruptedException e) {}
```
]

○
```
try {
    Thread.currentThread().wait();
}catch(InterruptedException e) {}
```

○
```
try {
    Thread.sleep(2);
}catch(InterruptedException e) {}
```

○ `SEGMENT 1` can be left blank, the code will anyway produce -40 on every execution.

Solution: Detailed sol

4. Consider the code given below.

```
class TicketMyshow implements Runnable{
    int available = 10;
    int wanted;
    public TicketMyshow(int w) {
        wanted = w;
    }
    public synchronized void run() {
        String name = Thread.currentThread().getName();
        System.out.println("AvailableTickets : " + available + " Wanted : "+wanted);
        if (wanted <= available){
            System.out.println(name+" booked " + wanted + " tickets");
            available = available - wanted;
        }
        else{
            System.out.println("Sorry no tickets for " + name);
        }
    }
}
public class BookTicket{
    public static void main(String[] args){
        TicketMyshow ticket = new TicketMyshow(5);
        Thread thread1 = new Thread(ticket,"Jock");
        thread1.start();
        Thread thread2 = new Thread(ticket, "John");
        thread2.start();
        Thread thread3 = new Thread(ticket, "Virat");
        thread3.start();
    }
}
```

Which of the following options is/are possible result/s of the above code?

◯
```
AvailableTickets : 10 Wanted : 5
Virat booked 5 tickets
AvailableTickets : 5 Wanted : 5
Jock booked 5 tickets
AvailableTickets : 0 Wanted : 5
Sorry no tickets for John
```

◯
```
AvailableTickets : 10 Wanted : 5
Jock booked 5 tickets
AvailableTickets : 5 Wanted : 5
John booked 5 tickets
```

```
AvailableTickets : 0 Wanted : 5
Sorry no tickets for Virat
```

○

```
AvailableTickets : 10 Wanted : 5
John booked 5 tickets
AvailableTickets : 5 Wanted : 5
Virat booked 5 tickets
AvailableTickets : 0 Wanted : 5
Sorry no tickets for Jock
```

[✓ All of the above]

Solution: We cannot predict order of thread's execution, but all threads are synchronized, among three threads one thread will not get any ticket.

5. Consider the code given below.

```
class BookTicket{
    int available=3;
    synchronized void extract(String name,int book){
        if (book<=available){
            System.out.println(name + " booked " + book + " ticket.");
            available = available - book;
        }
        else {
            System.out.println(name + ", you can not book " + book + " ticket.");
            System.out.println("Tickets available: " + available);
        }
    }
}
class Passenger implements Runnable{
    BookTicket bt;
    String name;
    int number;
    Passenger(BookTicket b, String n, int num){
        this.bt = b;
        this.name = n;
        this.number = num;
    }
    public void run(){
        bt.extract(name, number);
    }
}
public class Test {
    public static void main(String[] args){
        BookTicket obj = new BookTicket();
        BookTicket obj1 = new BookTicket();
        BookTicket obj2 = new BookTicket();
        Thread t1 = new Thread(new Passenger(obj, "Sun", 2));
        Thread t2 = new Thread(new Passenger(obj1, "Moon", 1));
        Thread t3 = new Thread(new Passenger(obj2, "Earth", 2));
        t1.start();
        t2.start();
        t3.start();
    }
}
```

What will be the possible output(s)?

[✓
```
Moon booked 1 ticket.
Sun booked 2 ticket.
Earth booked 2 ticket.
```
]

○
```
Sun booked 2 ticket.
Moon booked 1 ticket.
Earth, you can not book 2 ticket.
Tickets available: 0
```

○ This program generates `InterruptedException` at runtime.

○ This program generates `IllegalMonitorStateException` at runtime.

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

In this program multiple threads are associated with multiple `Objects`. So at a time, 1 thread from each object will get an opportunity to invade the critical region and create race condition. It is highly unlikely that this program will generate correct output fight shying the race condition, but this can not be guaranteed. Hence to prevent race condition class level lock is required.

6. Consider the code given below.

```
import java.util.concurrent.locks.*;
class DemoLock{
    ReentrantLock lck = new ReentrantLock();
    public void display(String name){
        lck.lock();
        try{
            for(int i = 1; i < 4; i++){
            System.out.print(i + ":" + name + " ");
            }
            System.out.print("\n");
        }
        finally{
            lck.unlock();
        }
    }
}
class Example extends Thread{
    DemoLock l_obj;
    String str;
    Example(DemoLock o, String str){
        this.l_obj = o;
        this.str = str;
    }
    public void run(){
        l_obj.display(str);
    }
}
public class FClass{
    public static void main(String[] args){
        DemoLock obj = new DemoLock();
        Example e = new Example(obj, "Sun");
        Example e2 = new Example(obj, "Moon");
        Example e3 = new Example(obj, "Earth");
        e.start();
        e2.start();
        e3.start();
    }
}
```

What is/are the possible output/s?

- ✓ `1:Sun 2:Sun 3:Sun`
  `1:Earth 2:Earth 3:Earth`
  `1:Moon 2:Moon 3:Moon`
- ◯ `1:Moon 2:Moon 1:Earth 1:Sun 2:Earth 3:Moon`
  `3:Earth`
  `2:Sun 3:Sun`
- ◯ `1:Earth 1:Sun 1:Moon`
  `2:Moon 2:Sun 2:Earth`
  `3:Sun 3:Moon 3:Earth`
- ✓ `1:Sun 2:Sun 3:Sun`
  `1:Moon 2:Moon 3:Moon`
  `1:Earth 2:Earth 3:Earth`

7. Consider the code given below.

```
import java.util.logging.*;
class Example extends Thread{
    Thread t;
    Example(Thread t){
        this.t = t;
    }
    public void run(){
        try {
            t.join();
        } catch (InterruptedException ex) {
            Logger.getGlobal().log(Level.SEVERE, ex.getMessage());
        }
        for(int i = 1; i < 3; i++){
            System.out.print(i + " ");
        }
    }
}
public class FClass{
    public static void main(String[] args) throws InterruptedException{
        Thread t1 = Thread.currentThread();
        Example t2 = new Example(t1);
        t2.start();
        for(int i = 3; i <= 5; i++){
            System.out.print(i + " ");
        }
    }
}
```

What is a possible output?

- ◯ `1 2 3 4 5`
- [✓ `3 4 5 1 2`]
- ◯ `3 4 5`
- ◯ All of them

8. Consider the code given below.

```
import java.util.*;
class PrlTest extends Thread{
    Map<String, Integer> mp;
    Thread th;
    PrlTest(Map<String, Integer> ic,Thread t){
        this.mp = ic;
        this.th = t;
    }
    public void run(){
        try {
            th.join();
        } catch (InterruptedException ex) {}
        mp.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, t1);
        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 is undetermined.

9. Consider the code given below.

```
import java.util.*;
import java.util.concurrent.*;
class Example extends Thread{
    Map siMap;
    Example(Map m){
        this.siMap = m;
    }
    public void run(){
        siMap.put("D",4);
    }
}
public class FClass{
    public static void main (String[] args) {
        Map<String, Integer> siMap = new ConcurrentHashMap();
        String[] str = {"A", "B", "C"};
        Integer[] arr = {1, 2, 3};
        for(int i = 0; i < str.length; i++){
                siMap.put(str[i],arr[i]);
        }
        Example t = new Example(siMap);
        t.start();
        Set s = siMap.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`.]

◯
```
A => 1
B => 2
C => 3
D => 4
```

◯
```
D => 4
A => 1
B => 2
C => 3
```

◯
```
A => 1
B => 2
C => 3
```