



**19BIT0292**

**Bhaumik Tandan**

**ASSESSMENT-2**

**ADVANCED JAVA**  
**PROGRAMMING**

**ITE2005**

**L25+L26**

**Q1)** Implement the producer consumer problem for the bounded buffer of size 'n'.

## **CODE**

```
import java.util.*;
```

```
class Buffer
```

```
{
```

```
    int size,index;
```

```
    int [] buffer;
```

```
    int capacity;
```

```
    Buffer(int size)
```

```
    {
```

```
        this.size=size;
```

```
        buffer=new int[size];
```

```
        index=-1;
```

```
        capacity=size;
```

```
    }
```

```
}
```

```
class Consumer extends Thread{
```

```
    Buffer b;
```

```
    int consumerNo;
```

```
    Consumer(Buffer x,int no)
```

```
    {
```

```
        b=x;
```

```
        consumerNo=no;
```

```

    }

    public void run()
    {
        while(true){
            synchronized(b)
            {
                try{
                    while(b.capacity==b.size)

                    b.wait();

                    System.out.println("Consumer "+consumerNo+" consumed
"+b.buffer[b.index]+" from index "+b.index);

                    if(b.index==0)
                        b.index=b.size-1;

                    else
                        b.index--;

                    b.capacity++;

                    b.notifyAll();

                    Thread.sleep(1000);

                }

                catch(Exception e)

                {

                    System.out.println("Exception in consumer "+consumerNo);

                }

            }

        }

    }

}

}

```

```

class Producer extends Thread{

    Buffer b;

    int producerNo;

    Producer(Buffer x,int no)

    {

        b=x;

        producerNo=no;

    }


    public void run()

    {

        while(true){

            synchronized(b)

            {

                try{

                    while(b.capacity==0)

                    b.wait();

                    b.index=(b.index+1)%b.size;

                    b.buffer[b.index]=(int)(Math.random()*1000);

                    b.capacity--;

                    System.out.println("Producer "+producerNo+" produced "+b.buffer[b.index]+"

at index "+b.index);

                    b.notifyAll();

                    Thread.sleep(1000);

                }

                catch(Exception e){

                    System.out.println("Exception in producer "+producerNo);

                }

            }

        }

    }

}

```

}

}

```
public class Q1 {  
    public static void main(String[] args)  
    {  
        Scanner sc=new Scanner(System.in);  
        System.out.print("Enter the size of buffer: ");  
        int size=sc.nextInt();  
        Buffer b=new Buffer(size);  
        System.out.print("Enter the number of producer: ");  
        int p=sc.nextInt();  
        System.out.print("Enter the number of consumer: ");  
        int c=sc.nextInt();  
        Producer[] p1=new Producer[p];  
        Consumer [] c1=new Consumer[c];  
        for(int i=0;i<p;i++)  
        {  
            p1[i]=new Producer(b,i+1);  
            p1[i].start();  
        }  
        for(int i=0;i<c;i++)  
        {  
            c1[i]=new Consumer(b,i+1);  
            c1[i].start();  
        }  
    }  
}
```

}

}

# OUTPUT

```
Enter the size of buffer: 10
Enter the number of producer: 3
Enter the number of consumer: 3
Producer 1 produced 232 at index 0
Producer 1 produced 17 at index 1
Producer 1 produced 657 at index 2
Producer 1 produced 868 at index 3
Producer 1 produced 200 at index 4
Producer 1 produced 797 at index 5
Producer 1 produced 649 at index 6
Producer 1 produced 282 at index 7
Producer 1 produced 953 at index 8
Producer 1 produced 240 at index 9
Consumer 3 consumed 240 from index 9
Consumer 3 consumed 953 from index 8
Consumer 3 consumed 282 from index 7
Consumer 3 consumed 649 from index 6
Consumer 3 consumed 797 from index 5
Consumer 3 consumed 200 from index 4
Consumer 3 consumed 868 from index 3
Consumer 3 consumed 657 from index 2
Consumer 3 consumed 17 from index 1
Consumer 3 consumed 232 from index 0
Producer 2 produced 152 at index 0
Producer 2 produced 385 at index 1
Producer 2 produced 378 at index 2
Producer 2 produced 223 at index 3
Producer 2 produced 839 at index 4
Producer 2 produced 675 at index 5
Producer 2 produced 405 at index 6
Producer 2 produced 551 at index 7
Producer 2 produced 525 at index 8
Producer 2 produced 99 at index 9
Consumer 2 consumed 99 from index 9
```

**Q2)** Lets assume the case of 2 threads (denoted A and B) that communicate using a circular array (a basic one-dimensional array "int[]") . More precisely, thread A generates and writes data into this array, and thread B reads these data from the array. Reading and the updating of variables for holding the start value and quantity are performed in synchronized blocks on the same object. If the shared array is full, then thread A waits until thread B finish reading data from it. If the array is empty, then thread B waits until thread A finish writing some data into it. Since the array can't be full and empty at the same time, it is impossible for both threads to be waiting at the same time. Also, since thread A can only write in the free part of the array, and thread B can only read from the occupied part of the array, then the threads will never access the same cell at the same time, i.e. the threads are working on 2 independent parts of the array. Implement the above scenario and display the output at each sequence.

## **CODE**

```
import java.util.Scanner;
```

```
class Array
```

```
{
```

```
    int size,index;
```

```
    int [] arr;
```

```
int capacity;  
    Array(int size)  
    {  
        this.size=size;  
        arr=new int[size];  
        index=-1;  
        capacity=size;  
    }
```

```
}
```

```
class A extends Thread{
```

```
    Array obj;
```

```
    A(Array x)
```

```
    {
```

```
        obj=x;
```

```
    }
```

```
    public void run()
```

```
    {
```

```
        while(true){
```

```
            synchronized(obj)
```

```
            {
```

```
                try{
```

```
                    while(obj.capacity==0)
```

```
                        obj.wait();
```

```
                        obj.index=(obj.index+1)%obj.size;
```

```
                        obj.arr[obj.index]=(int)(Math.random()*100);
```

```
                        obj.capacity--;
```



```

        System.out.println("A wrote "+obj.arr[obj.index]+" at index "+obj.index);
        obj.notifyAll();
        Thread.sleep(1000);
    }

    catch(Exception e){
        System.out.println("Exception in A");
    }
}

}

}
}

```

```

class B extends Thread

```

```

{
    Array obj;
    B(Array x)
    {
        obj=x;
    }
    public void run()
    {
        while(true){
            synchronized(obj)
            {
                try{
                    while(obj.capacity==obj.size)
                        obj.wait();

                    System.out.println("B read "+obj.arr[obj.index]+" at index "+obj.index);

```

```

        if(obj.index==0)
        obj.index=obj.size-1;
        else
        obj.index--;
        obj.capacity++;
        obj.notifyAll();
        Thread.sleep(1000);
    }
    catch(Exception e){
        System.out.println("Exception in B");
    }
}
}
}
}

```

```

public class Q2 {
    public static void main(String[] args)throws Exception {
        Scanner sc=new Scanner(System.in);
        System.out.print("Enter the size of the array: ");
        int size=sc.nextInt();
        Array obj=new Array(size);
        A a=new A(obj);
        B b=new B(obj);
        a.start();
        b.start();
    }
}

```

# OUTPUT

```
Enter the size of the array: 5
A wrote 82 at index 0
A wrote 40 at index 1
A wrote 98 at index 2
A wrote 32 at index 3
A wrote 49 at index 4
B read 49 at index 4
B read 32 at index 3
B read 98 at index 2
B read 40 at index 1
B read 82 at index 0
A wrote 25 at index 0
A wrote 87 at index 1
A wrote 98 at index 2
A wrote 94 at index 3
A wrote 6 at index 4
B read 6 at index 4
B read 94 at index 3
B read 98 at index 2
B read 87 at index 1
B read 25 at index 0
A wrote 91 at index 0
A wrote 46 at index 1
A wrote 34 at index 2
A wrote 93 at index 3
A wrote 82 at index 4
B read 82 at index 4
B read 93 at index 3
B read 34 at index 2
B read 46 at index 1
B read 91 at index 0
A wrote 62 at index 0
```

**Q3)** Write a method that takes a string and returns the number of unique characters in the string. It is expected that a string with the same character sequence may be passed several times to the method. Since the counting operation can be time consuming, the method should cache the results, so that when the method is given a string previously encountered, it will simply retrieve the stored result. Use collections and maps where appropriate.

## **CODE**

```
import java.util.*;

public class Q3 {

    static Map<String,Integer> cache = new HashMap<String,Integer>();

    static int countUnique(String s) {

        if(cache.containsKey(s))

            return cache.get(s);

        Map<Character,Boolean> alphabetFrequency = new
HashMap<Character,Boolean>();

        for(int i=0;i<s.length();i++)

            alphabetFrequency.put(s.charAt(i),true);

        int count = alphabetFrequency.size();

        cache.put(s,count);

        return count;

    }

    static void printResults(String s) {

        System.out.println("String: "+s);
```

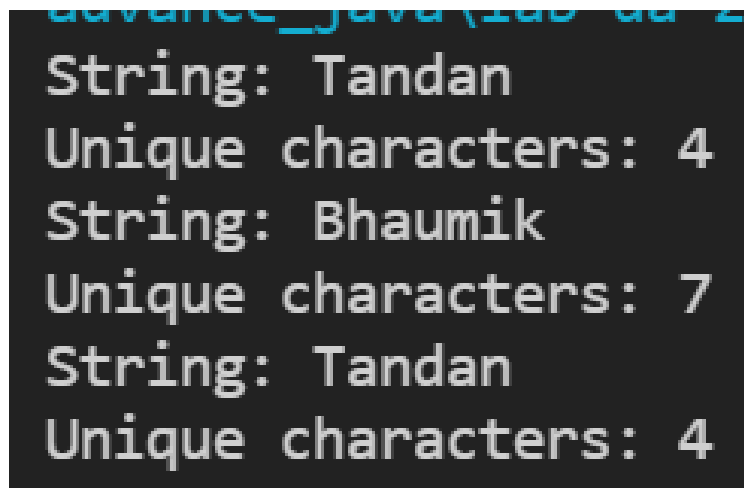
```

        System.out.println("Unique characters: "+countUnique(s));
    }

    public static void main(String[] args)
    {
        printResults("Tandan");
        printResults("Bhaumik");
        printResults("Tandan");
    }
}

```

## **OUTPUT**



```

String: Tandan
Unique characters: 4
String: Bhaumik
Unique characters: 7
String: Tandan
Unique characters: 4

```

**Q4)** Make a Map that associates the following employee IDs with names. Keys and values of Maps can be any Object type, so in real life you would probably have the key be a String and the associated value be a Person or Employee object. To make things simpler on this exercise, you can use String for both the ID and the name, rather than bothering to create a Person or Employee class. The point here is to associate keys with values, then retrieve values later based on keys.

ID	Name
a1234	Steve Jobs
a1235	Scott McNealy
a1236	Jeff Bezos
a1237	Larry Ellison
a1238	Bill Gates

Make test cases where you test several valid and invalid ID's and print the associated name.

## CODE

```
import java.util.*;

public class Q4 {

    static Map<String,String> map = new HashMap<String,String>();

    static void addData()
    {
        map.put("a1234","Steve Jobs");
        map.put("a1235","Scott McNealy");
        map.put("a1236","Jeff Bezos");
        map.put("a1237","Larry Ellison");
        map.put("a1238","Bill Gates");
    }

    static void test()
    {
        System.out.println("\nEnter the key: ");
        Scanner sc = new Scanner(System.in);
        String key = sc.next();
```

```
if(map.containsKey(key))
```

```
    System.out.println("The value is: " + map.get(key));
```

```
else
```

```
    System.out.println("The key is not present in the map");
```

```
System.out.println("1. Get the value of the key");
```

```
System.out.println("2. Exit");
```

```
int choice = sc.nextInt();
```

```
switch(choice)
```

```
{
```

```
    case 1: test();
```

```
        break;
```

```
    case 2: System.exit(0);
```

```
        break;
```

```
    default: System.out.println("Invalid choice");
```

```
        test();
```

```
}
```

```
}
```

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

```
{
```

```
    addData();
```

```
    test();
```

```
}
```

```
}
```

# OUTPUT

```
Enter the key: a1234
The value is: Steve Jobs
1. Get the value of the key
2. Exit
1

Enter the key: dsd
The key is not present in the map
1. Get the value of the key
2. Exit
1

Enter the key: a1235
The value is: Scott McNealy
1. Get the value of the key
2. Exit
2
```

**Q5)** Make a coin-flipping class that implements Runnable. The run method should flip 1000 coins and print out whenever it sees 3 or more consecutive heads. Make a task queue, and put 5 separate instances of the Runnable class in the queue. In the printouts, you can use `Thread.currentThread().getName()` to identify the thread. You are following variation 1 of the basic threading approach (separate classes that implement Runnable), so your code will look something like this (or, you could call execute from a loop):



```

public class Foo implements Runnable {
    public void run() { loop, flip coins, check for 3+ heads in a row }
}
-----
public class Driver {
    public static void main(String[] args) {
        ExecutorService tasks = ...
        tasks.execute(new Foo()); // Multiple instances of Foo
        tasks.execute(new Foo());
        tasks.execute(new Foo());
    }
}

```

## CODE

```

import java.util.*;

class Coin_Flipping implements Runnable{

    boolean flipCoin()

    {

        //0 means head

        //1 means tail

        return Math.random()<0.5;

    }

    @Override

    public void run()

    {

        try{

            int c=0;

            for(int i=0;i<1000;i++)

            {

                boolean x=flipCoin();

                if(x==false)

                {

                    c++;

                    continue;

                }

            }

        }

    }

}

```

```

        if(c>=3){

            System.out.println(c+" consecutive heads were given by
"+Thread.currentThread().getName());

            Thread.sleep(1000);

        }

        c=0;

    }

}

catch(InterruptedException e){

    System.out.println(Thread.currentThread().getName()+" interrupted");

}

}

}

```

```

class TaskQueue{

    void execute(Coin_Flipping c)

    {

        Thread t=new Thread(c);

        t.start();

    }

}

public class Q5 {

    public static void main(String[] args)

    {

        TaskQueue t=new TaskQueue();

        for(int i=0;i<5;i++)

            t.execute(new Coin_Flipping());

    }

}

```

# OUTPUT

```
advance_java(100, 2, 1) { java -c 2-
3 consecutive heads were given by Thread-2
4 consecutive heads were given by Thread-1
6 consecutive heads were given by Thread-4
3 consecutive heads were given by Thread-0
3 consecutive heads were given by Thread-3
3 consecutive heads were given by Thread-1
3 consecutive heads were given by Thread-2
6 consecutive heads were given by Thread-4
3 consecutive heads were given by Thread-0
3 consecutive heads were given by Thread-3
3 consecutive heads were given by Thread-2
4 consecutive heads were given by Thread-0
3 consecutive heads were given by Thread-4
4 consecutive heads were given by Thread-1
3 consecutive heads were given by Thread-3
3 consecutive heads were given by Thread-0
3 consecutive heads were given by Thread-2
4 consecutive heads were given by Thread-4
4 consecutive heads were given by Thread-1
5 consecutive heads were given by Thread-3
5 consecutive heads were given by Thread-0
4 consecutive heads were given by Thread-2
5 consecutive heads were given by Thread-4
3 consecutive heads were given by Thread-1
5 consecutive heads were given by Thread-3
3 consecutive heads were given by Thread-0
4 consecutive heads were given by Thread-4
3 consecutive heads were given by Thread-1
3 consecutive heads were given by Thread-2
3 consecutive heads were given by Thread-3
```



**Q6)** Make an “infinite” stream that generates random doubles between 0 and 10. Use it to

- Print 5 random doubles
- Make a List of 10 random doubles
- Make an array of 20 random doubles

Note: in general, if you are dealing with numbers, DoubleStream is preferred over Stream because DoubleStream uses primitives and has more convenient methods (e.g., min, max, sum, average). In this case, however, use Stream because it is hard to turn a DoubleStream into a List and because it is hard to print a double[] but easy to print a Double[] (e.g., pass the array to Arrays.asList and print the resultant List). So, for this part of the exercises, use Stream.generate, not DoubleStream.generate

## **CODE**

```
import java.util.*;

import java.util.stream.*;

public class Q6 {

    static int printCount = 0, listCount=0, arrayCount=0;

    static List<Double> list = new ArrayList<Double>();

    static double[] array = new double[20];

    static Double randomDouble() {

        return Math.random()*10;

    }

    static void task(Double e) {

        if(printCount==0)

        {

            System.out.println("5 random numbers are: ");
```

```
}

if(printCount<5) {
    System.out.println(e);
    printCount++;
    return;
}

if(listCount<10)
{
    list.add(e);
    listCount++;
    return;
}

if(listCount==10)
{
    System.out.println("\nList of 10 random numbers is: ");
    list.stream().forEach(System.out::println);
    listCount++;
}

if(arrayCount<20)
{
    array[arrayCount]=e;
    arrayCount++;
    return;
}

if(arrayCount==20)
{
    System.out.println("\nArray of 20 random numbers is: ");
    for(int i=0;i<array.length;i++)
    {
        System.out.println(array[i]);
    }
    arrayCount++;
}
```

```

    }

    public static void main(String[] args)
    {
        Stream<Double> stream = Stream.generate(Q6::randomDouble);

        stream.forEach(Q6::task);
    }
}

```

## OUTPUT

5 random numbers are:

```

1.3998969094772795
5.242263438902841
1.6600264025641864
4.068982267198383
6.481923494667697

```

List of 10 random numbers is:

```

2.0323683444156937
7.575591388152738
1.852621332457487
9.596067833223827
5.2799584953985015
5.275170657016199
6.441662816075145
6.145318349119378
9.00528995229474
1.761194834772274

```

Array of 20 random numbers is:

```

6.039697601125696
2.060912676077389
7.570413545446682

```

Array of 20 random numbers is:

```

6.039697601125696
2.060912676077389
7.570413545446682
3.5709799240743525
7.815328965662324
7.121092916601541
7.0387143164859225
1.824041397386993
5.870221045321466
1.6776172527419142
8.20125328502332
9.300363268626784
1.7005213309690825
8.21855227503778
8.296587277185429
5.631469426021569
5.276958109880012
1.7056443443390235
6.976124449451256
6.852160924728704

```

**CLICK HERE**  
**FOR GITHUB**  
**LINK**