Ordered Printing

This problem is about imposing an order on thread execution.

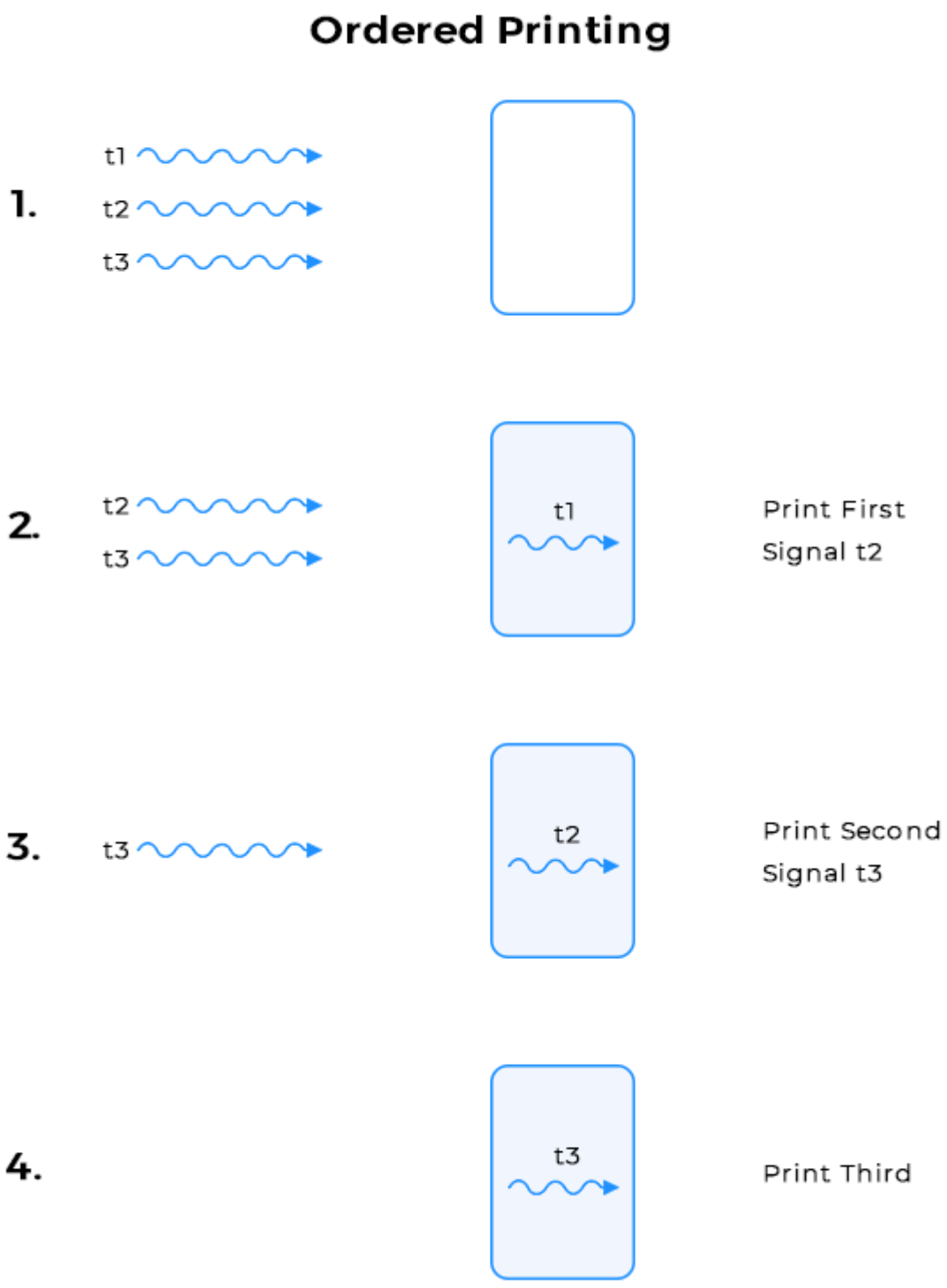
Problem

Suppose there are three threads t1, t2 and t3. t1 prints **First**, t2 prints **Second** and t3 prints **Third**. The code for the class is as follows:

```
public class OrderedPrinting {

    public void printFirst() {
       System.out.print("First");
    }

    public void printSecond() {
       System.out.print("Second");
    }

    public void printThird() {
       System.out.print("Third");
    }

}
```

Thread t1 calls printFirst(), thread t2 calls printSecond(), and thread t3 calls printThird(). The threads can run in any order. You have to synchronize the threads so that the functions **printFirst()**, **printSecond()** and **printThird()** are executed in order.

The workflow of the program is shown below:



Workflow

Solution

We present two solutions for this problem; one using **volatile** variable and the other using **CountDownLatch**.

Solution 1

In this solution, we have a class `OrderedPrinting` that consists of a private volatile variable `flag`. To recap, a **volatile** variable is the one that gets stored in the main memory and is never cached. It is a primitive to achieve synchronization and concurrency in Java. Read/write operations are only done in the main memory ensuring visibility of changes to variables across threads. The basic structure of our class is as follows:

```
class OrderedPrinting {
    private volatile int flag;

    public OrderedPrinting() {
        flag = 1;
    }
}
```

In the constructor, `flag` is initialized with 1. Next we will explain the printFirst() function below:

```
public void printFirst() throws InterruptedException {
    //infinite loop keeps checking in case flag doesn't match
    for(;;) {
        if (flag==1) {
            System.out.println("First");
            //for second printing, increment flag
            flag = 2;
            break;
        }
    }
}
```

The `printFirst()` method prints “First” when the value of `flag` is 1. After printing, the value of `flag` gets incremented and the loop is broken. This code is kept in an infinite loop so that it keeps checking for the condition that matches the current `flag` value.

```
public void printSecond() throws InterruptedException {
    for(;;) {
        if (flag==2) {
            System.out.println("Second");
            //for third printing, increment flag
```

```
            flag=3;
            break;

        }
    }
}
```

```
public void printThird() throws InterruptedException {
    for(;;) {
        if (flag==3) {
            System.out.println("Third");
            //reset flag value to ensure ordered printing
            flag = 1;
            break;
        }
    }
}
```

`printSecond()` and `printThird()` operate in the same manner as `printFirst()` but the only difference in `printThird()` is that the `flag`'s value is reset to 1 rather than getting incremented. This is done to ensure proper printing order of the code.

To run our proposed solution, we will create another class to achieve multi-threading. When we extend `Thread` class, each of our thread creates a unique object and associates with the parent class. This class has two variables: one is the object of `OrderedPrinting` and the other is a string variable `method`. The string parameter checks the method to be invoked from `OrderedPrinting`.

```
class OrderedPrintingThread extends Thread {
    private OrderedPrinting obj;
    private String method;

    public OrderedPrintingThread(OrderedPrinting obj, String method)
{
        this.method = method;
        this.obj = obj;
    }

    public void run() {
        //for printing "First"
        if ("first".equals(method)) {
            try {
```

```
                obj.printFirst();
            }

            catch(InterruptedException e) {

            }
        }
        //for printing "Second"
        else if ("second".equals(method)) {
            try {
                obj.printSecond();
            }
            catch(InterruptedException e) {

            }
        }
        //for printing "Third"
        else if ("third".equals(method)) {
            try {
                obj.printThird();
            }
            catch(InterruptedException e) {

            }
        }
    }
}
```

We will be creating 3 threads in the `Main` class for testing each solution. Each thread will be passed the same object of `OrderedPrinting`. **t1** will call `printFirst()`, **t2** will call `printSecond()` and **t3** will call `printThird()`. The output shows printing done in the proper order i.e first, second and third irrespective of the calling order of threads.

```
1  class OrderedPrinting
2  {
3
4      private volatile int flag;
5
6      public OrderedPrinting()
7      {
8          flag = 1;
9      }
10
11     public void printFirst() th
12     {
13         for(;;) {
```

```
            if (flag==1) {
                System.out.pri
                flag = 2;
                break;
            }
        }
    }

    public void printSecond() {
    {
        for(;;)
        {
            if (flag==2)
            {
                System.out.pri
                flag=3;
                break;
            }
```




Solution 2

The second solution includes the use of **CountDownLatch**; a synchronization utility used to achieve concurrency. It manages multithreading where a certain sequence of operations or tasks is required. Everytime a thread finishes its work, `countdown()` is invoked, decrementing the counter by 1. Once this count reaches zero, `await()` is notified and control is given back to the main thread that has been waiting for others to finish.

The basic structure of the class `OrderedPrinting` is the same as presented in solution 1 with the only difference of using **countdownlatch** instead of **volatile** variable. We have 2 **countdownlatch** variables that get initialized with 1 each.

```
class OrderedPrinting {
    CountDownLatch latch1;
    CountDownLatch latch2;

    public OrderedPrinting() {
        latch1 = new CountDownLatch(1);
        latch2 = new CountDownLatch(1);
    }
}
```

In `printFirst()` method, `latch1` decrements and reaches 0, waking up the waiting threads consequently. In `printSecond()`, if `latch1` is free (reached 0), then the printing is done and `latch2` is decremented. Similarly in the third method `printThird()`, `latch2` is checked and printing is done. The latches here act like switches/gates that get closed and opened for particular actions to pass.

```
public void printFirst() throws InterruptedException {
    //print and notify waiting threads
    System.out.println("First");
    latch1.countDown();
}
```

```
public void printSecond() throws InterruptedException {
    //wait if "First" has not been printed yet
    latch1.await();
    //print and notify waiting threads
    System.out.println("Second");
    latch2.countDown();
}
```

```
public void printThird() throws InterruptedException {
    //wait if "Second" has not been printed yet
    latch2.await();
    System.out.println("Third");
}
```

As in the previous solution, we create `OrderedPrintingThread` class which extends the `Thread` class. Details of this class are explained at length above.

```
import java.util.concurrent.CountDownLatch;

class OrderedPrinting
{
    CountDownLatch latch1;
    CountDownLatch latch2;

    public OrderedPrinting()
    {
        latch1 = new CountDownLatch(1);
        latch2 = new CountDownLatch(1);
    }
```

```
    public void printFirst() throws InterruptedException
    {
        System.out.println("First");
        latch1.countDown();
    }

    public void printSecond() throws InterruptedException
    {
        latch1.await();
        System.out.println("Second");
        latch2.countDown();
    }

    public void printThird() throws InterruptedException
    {
        latch2.await();
        System.out.println("Third");
    }
}

class OrderedPrintingThread extends Thread
{
    private OrderedPrinting obj;
    private String method;

    public OrderedPrintingThread(OrderedPrinting obj, String method)
    {
        this.method = method;
        this.obj = obj;
    }

    public void run()
    {
        if ("first".equals(method))
        {
            try
            {
                obj.printFirst();
            }
            catch(InterruptedException e)
            {

            }
        }
        else if ("second".equals(method))
        {
            try
            {
                obj.printSecond();
            }
            catch(InterruptedException e)
            {

            }
        }
        else if ("third".equals(method))
        {
            try
            {
                obj.printThird();
            }
```

```
            catch(InterruptedException e)
            {

            }
        }
    }
}

public class Main
{
        public static void main(String[] args)
    {
        OrderedPrinting obj = new OrderedPrinting();

        OrderedPrintingThread t1 = new OrderedPrintingThread(obj, "first");
        OrderedPrintingThread t2 = new OrderedPrintingThread(obj, "second");
        OrderedPrintingThread t3 = new OrderedPrintingThread(obj, "third");

        t3.start();
        t2.start();
        t1.start();

    }
}
```