Part 2

Thinking about Volatile

```
1.  temp = curr.p;
2.  synchronized(temp){
3.          synchronized(curr){
4.                  if(temp == curr.p){
5.                          synchronized(curr.n){
6.                           …
7.                           }
8.                  }
9.          }
10.  }
```

a) Should p be marked volatile in the class definition for Element

Answer: Yes

Explanation: Following table considers both the situations, where "p" is volatile or not

If "p" is not Volatile	If "P" is Volatile
Step 1 the temp is assigned the reference for Obj1 from curr.p.	Step 1 the temp is assigned the reference for Obj1 from curr.p.
Between Step 1 and 2 the curr.p could be made to point to Obj2 by Thread2. But the change has not been communicated to Thread1 yet.	Between Step 1 and 2 the curr.p could be made to point to Obj2 by Thread2. The change would be communicated to Thread1 instantly.
Step 2 We locked Obj1	Step 2 We locked Obj1
Step3 We locked Obj3 to which curr points	Step3 We locked Obj3 to which curr points
In Thread1 curr.p may still point to Obj1 so producing false positive for test condition in Step4.	curr.p will have latest value so will be pointing to Obj2. There will be no false positive for test condition in Step4.

b) Should n be marked volatile in the class definition for Element

Answer: No

Explanation: Following table considers both the situations, where "p" is volatile or not

If "n" is not Valatile	If "n" is Volatile
Step 1 the temp is assigned the reference for Obj1.	Step 1 the temp is assigned the reference for Obj1.
Suppose between Step 1 and 2 the curr.p could be made to point to Obj2 by some other Thread.	Suppose between Step 1 and 2 the curr.p could be made to point to Obj2 by some other Thread.
Step 3 makes sure that the object pointed by curr is locked	Step 3 makes sure that the object pointed by curr is locked

Condition in Step4 makes sure that temp and curr.p still points to same object. i.e. Obj1 == Obj2	Condition in Step4 makes sure that temp and curr.p still points to same object. i.e. Obj1 == Obj2
After Step 3 executed successfully curr.n can't be accessed/changed by more than 1 Thread. So no point of making it Volatile.	No added advantage of using Volatile for curr.n

c) Should temporary variable temp be marked volatile?

Answer: No

Explanation: Following table considers both the situations, where "p" is volatile or not

If "temp" is not Volatile	If "temp" is Volatile
Step 1 the temp is assigned the reference for Obj1.	Step 1 the temp is assigned the reference for Obj1.
Suppose between Step 1 and 2 the curr.p could be made to point to Obj2 by some other Thread.	Suppose between Step 1 and 2 the curr.p could be made to point to Obj2 by some other Thread. Additionally temp could be updated by some other Thread to point to Obj3.
Step 3 makes sure that the object pointed by curr is locked	Step 3 makes sure that the object pointed by curr is locked
Condition in Step4 makes sure that temp and curr.p still points to same object. i.e. Obj1 == Obj2	Condition in Step4 may produce false negative even if curr.p was not changed since temp was made to point to Obj3 by some other Thread in its Step1.

Part 3

Synchronized, blocks or methods?

Table 1: Difference between Synchronized blocks or methods

	Synchronized blocks	Synchronized methods
Choice of Locked Object	We can choose which object to acquire lock upon.	We acquire lock on the "this", i.e. caller of the synchronized method.
Choice of code locked	Provides flexibility to lock exactly the amount of code we want in a method.	Locks all the code inside the method making it exclusive for a thread.
Performance	Higher and as least as the synchronized methods. Since we may allow concurrency to some extent in the method with some code outside synchronized block.	Lower than synchronized methods since we made the complete method code exclusive for a thread.
Code complexity	Higher than method where we blindly locked all the code of method.	Lower than the synchronized block.

Usability	The synchronized blocks can serve any purpose a synchronized method can without changing the programming logic. i.e.:
	
	synchronized boolean insertAfter() {
	…….
	}
	Can be replaced by
	boolean insertAfter() {
	synchronized(this) {
	…….
	}
	}
	Without changing the program logic
		Reverse may not be true. In our code snippet for the previous part of the question we are locking three objects curr, curr.p and curr.n. This would not be possible using synchronized methods, without changing program logic.

Fine Grained Handling of Complex situations like Reader-Writer situation	If we want to create a specific locking mechanism where we allow multiple locks on same objects in some situation and exclusive locks in some other than synchronized blocks allow us to encode the functionality.	Synchronized methods provide can't help us in these situations since synchronized method won't allow 2 threads to enter critical section simultaneously.

We can achieve functionality of Synchronized methods by Synchronized blocks if we include all the code of a method inside a synchronized block which is synchronized at "this".

Example:

Following **Synchronized Method:**

```
synchronized boolean insertafter() {
```

```
        // The method is exclusive for a single Thread.
         ……..……..
}
```

May be changed to following **synchronized block:**

```
boolean insertAfter() {
      synchronized (this){
            // The  code is exclusive for a Thread with Lock on calling Object.
             ……..……..
       }
}
```

Both the above implementations are logically the same and provide same functionality.