```
    public synchronized void method1()
    {
      //Access ia1 and ia2
    }
    public synchronized void method2()
    {
      //Access ia1 and ia2
    }
    public synchronized void method3()
    {
      //Access da1 and da2
    }
    public synchronized void method4()
    {
      //Access da1 and da2
    }
    //...
}
```

This class is certainly thread safe. Each method must be declared `synchronized` in order to ensure the arrays are not corrupted by multiple threads accessing this object concurrently. For example, because `method1` and `method2` both access and potentially alter the arrays `ia1` and `ia2`, access to them must be synchronized. The same is true of `method3` and `method4`.

Notice, however, that although `method1` and `method2` must be synchronized with each other, they do not need to be synchronized with either `method3` or `method4`. This is because `method1` and `method2` do not operate on data that `method3` and `method4` operate on. This is similarly true for `method3` and `method4` with regard to `method1` and `method2`.

Unfortunately, this is how instance methods are sometimes synchronized in classes. However, synchronization in Java is not very granular. Synchronization provides you with only one lock per object. In the previous code, if you create an object of class `Test` and call `method1` on the main thread and `method3` on a secondary thread, you pay an unnecessary performance penalty. These methods synchronize with one another even though there is no need for them to do so. Remember that when a method is declared `synchronized`, the lock obtained is the lock for the object on which the method is invoked. Therefore, both methods attempt to get the same lock.

To fix the problem in the previous code you need multiple locks per object. Because Java does not provide this, you must furnish your own mechanism. One way to accomplish this is to create objects as instance data that serve only to pro-