Because these methods are invoked on separate threads, they attempt to execute concurrently. You might think that because both methods are declared `synchronized`, the execution of the two threads cannot interleave between the two methods. You might also think that the output of this code is either the string `printM1` or `printM2`, depending on which thread enters its `synchronized` method first and obtains the lock. After all, both methods are declared with the `synchronized` keyword.

Despite both methods' being declared `synchronized`, they are not thread safe. This is because one is a `synchronized static` method and the other is a `synchronized` instance method. Thus, they obtain two different locks. The instance method `printM1` obtains the lock for the `Foo` object. The `static` method `printM2` obtains the lock for the `Class` object of class `Foo`. These two locks are different and do not affect one another. When this code is executed, both strings are printed to the screen. In other words, execution interleaves between the two methods.

What if this code needs to be synchronized? For example, both methods might share a common resource. To protect the resource, the code must be properly synchronized in order to avoid a conflict. There are two options to solve this problem:

1. Synchronize on the common resource.
2. Synchronize on a special instance variable.

Option 1 involves simply adding `synchronized` statements in the methods in order to synchronize on the shared resource. For example, if these two methods were updating the same object, they would synchronize on it. The code might look like this:

```
class Foo implements Runnable
{
  private SomeObj someobj;
  public void printM1()
  {
    synchronized(someobj) {
      //The code to protect.
    }
  }

  public static void printM2(Foo f)
  {
    synchronized(f.someobj) {
      //The code to protect.
```