Demo Double Checked Locking

Issue Description (Data Race, Requires Volatile)

- Double Checked Locking is a design pattern (or anti-pattern) from POSA (patterns of software architecture).
- The pattern is broken if implemented one-to-one in languages like C# (also Java and others).
- Two threads may call Singleton.Instance: Only one may acquire the lock and create the instance.
- While writing the instance in the lock, another thread may read the unsynchronized instance variable in the first if-condition.
- Both accesses can happen concurrently and are not mutually synchronized. Only one side holds the lock, the other not.
- This is a formal data race.
- In practice, parts of the constructor execution could be reordered after the assignment of the new created object to instance. The latter may happen with an optimizing compiler, runtime system (liberty allowed by the language specification) or a processor such as ARM. (Intel processor write instructions are not reordered.) It can thus lead to real race condition and is thus considered broken.

```
if (instance == null)
{
    lock (locker)
    {
        if (instance == null)
        {
            instance = new Singleton();
        }
    }
}
```

Checker Output (1 Issue, 2 Locations)

```
Issue: #0 Data race on DoubleCheckedLocking.Singleton.instance
  caused by write at "instance = new Singleton()" in Program.cs line 27
    caused by access DoubleCheckedLocking.Singleton.Instance at "Singleton.Instance" in Program.cs line 40
      caused by thread or task at "() => Console.WriteLine(Singleton..." in Program.cs line 40
        caused by call DoubleCheckedLocking.Program.Main()
          caused by initial thread at "Main" in Program.cs line 38
  caused by read at "instance" in Program.cs line 21
    caused by access DoubleCheckedLocking.Singleton.Instance at "Singleton.Instance" in Program.cs line 41
      caused by thread or task at "() => Console.WriteLine(Singleton..." in Program.cs line 41
        caused by call DoubleCheckedLocking.Program.Main()
          caused by initial thread at "Main" in Program.cs line 38
```

Problem Fixing (2 Options)

Option 1: Volatile Variable

Declare the variable instance as volatile, imposing memory fences/barriers on the accesses. Certain reordering is thereby prohibited, i.e. the constructor must complete before the write on instance.

```
private static volatile Singleton instance;
```

Option 2: Explicit Volatile Accesses

Use explicit Volatile read and write accesses:

Acquire a monitor lock the in BankAccount.Balance Getter. Use a full getter statement block.

```
if (Volatile.Read(ref instance) == null)
{
   lock (locker)
   {
      if (instance == null)
      {
         Volatile.Write(ref instance, new Singleton());
      }
   }
}
return instance;
```