**lock Statement (C# Reference)**

The **lock** keyword marks a statement block as a critical section by obtaining the mutual-exclusion lock for a given object, executing a statement, and then releasing the lock. This statement takes the following form:

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl01other');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl01other');)

Object thisLock = new Object();

lock (thisLock)

{

// Critical code section.

}

For more information, see [Thread Synchronization (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/ms173179.aspx).

http://i.msdn.microsoft.com/Global/Images/clear.gif Remarks

The **lock** keyword ensures that one thread does not enter a critical section of code while another thread is in the critical section. If another thread tries to enter a locked code, it will wait, block, until the object is released.

The section [Threading (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/ms173178.aspx) discusses threading.

The **lock** keyword calls [Enter](http://msdn.microsoft.com/en-us/library/de0542zz.aspx) at the start of the block and [Exit](http://msdn.microsoft.com/en-us/library/system.threading.monitor.exit.aspx) at the end of the block.

In general, avoid locking on a **public** type, or instances beyond your code's control. The common constructs lock (this), lock (typeof (MyType)), and lock ("myLock") violate this guideline:

* lock (this) is a problem if the instance can be accessed publicly.
* lock (typeof (MyType)) is a problem if MyType is publicly accessible.
* lock(“myLock”) is a problem because any other code in the process using the same string, will share the same lock.

Best practice is to define a **private** object to lock on, or a **private static** object variable to protect data common to all instances.

http://i.msdn.microsoft.com/Global/Images/clear.gif Example

The following sample shows a simple use of threads without locking in C#.

C#

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl15CSharp');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl15CSharp');)

//using System.Threading;

class ThreadTest

{

public void RunMe()

{

Console.WriteLine("RunMe called");

}

static void Main()

{

ThreadTest b = new ThreadTest();

Thread t = new Thread(b.RunMe);

t.Start();

}

}

// Output: RunMe called

The following sample uses threads and **lock**. As long as the **lock** statement is present, the statement block is a critical section and balance will never become a negative number.

C#

[[http://i.msdn.microsoft.com/Global/Images/clear.gif](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl16CSharp');)Copy Code](javascript:CopyCode('ctl00_rs1_mainContentContainer_ctl16CSharp');)

// using System.Threading;

class Account

{

private Object thisLock = new Object();

int balance;

Random r = new Random();

public Account(int initial)

{

balance = initial;

}

int Withdraw(int amount)

{

// This condition will never be true unless the lock statement

// is commented out:

if (balance < 0)

{

throw new Exception("Negative Balance");

}

// Comment out the next line to see the effect of leaving out

// the lock keyword:

lock (thisLock)

{

if (balance >= amount)

{

Console.WriteLine("Balance before Withdrawal : " + balance);

Console.WriteLine("Amount to Withdraw : -" + amount);

balance = balance - amount;

Console.WriteLine("Balance after Withdrawal : " + balance);

return amount;

}

else

{

return 0; // transaction rejected

}

}

}

public void DoTransactions()

{

for (int i = 0; i < 100; i++)

{

Withdraw(r.Next(1, 100));

}

}

}

class Test

{

static void Main()

{

Thread[] threads = new Thread[10];

Account acc = new Account(1000);

for (int i = 0; i < 10; i++)

{

Thread t = new Thread(new ThreadStart(acc.DoTransactions));

threads[i] = t;

}

for (int i = 0; i < 10; i++)

{

threads[i].Start();

}

}

}