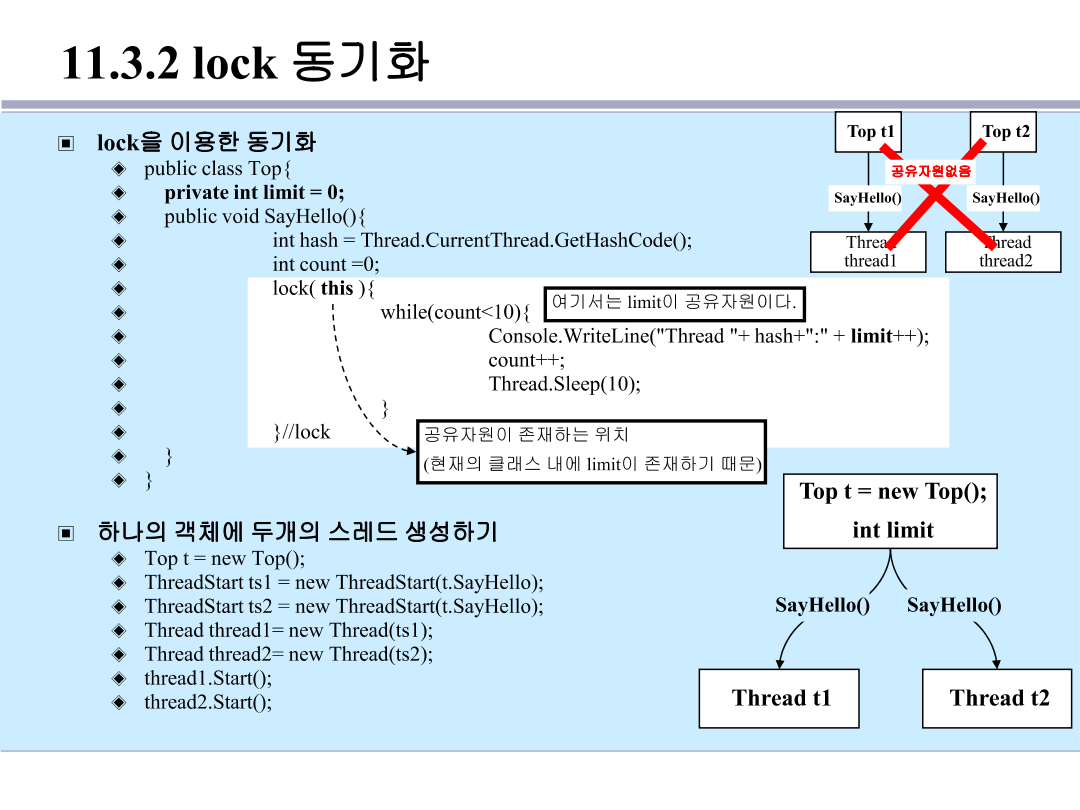


**11.3.1 멀티 스레드의 문제점과 해결방안**  
  
애플리케이션에서 스레드의 수가 많아지면 관리하기가 더 까다로워진다. 당연히 소수의 스레드를 운영할 때보다 여러 개의 스레드를 운영할 때 더 많은 문제가 발생한다. 이러한 문제는 여러 개의 스레드가 하나의 자료를 공유하기 때문에 발생하는 공유자원(Shared Data)의 문제이다. 즉 공유자원을 사용할 때 발생하는 동기화의 문제이다.  
  
**□ 스레드에서 문제가 발생하는 곳**  
◇ 여러 개의 스레드가 하나의 자료(공유자원)를 공유할 때  
  
스레드가 생성되더라도 스레드 내부에 존재하는 각각의 자료만을 사용한다면 큰 문제가 되지 않는다. 하지만 여러 개의 스레드들이 동시에 하나의 자료를 사용하다 보면 자료를 서로 차지하려는 병목현상(BottleNeck)이 발생한다. 이 현상은 병의 목처럼 한 곳에 집중되는 현상을 말한다.  
  
그리고 하나의 자원을 여러 곳에서 요청하게 될 때 자원을 한쪽에서는 무한히 사용하고 있으면 다른 한쪽에서는 무한히 대기하는 데드락(DeadLock)이 발생할 수도 있다. 데드락은 시스템에 치명적인 장애를 발생시킬 수 있다.  
  
**□ 공유자원을 사용할 때의 문제점**  
◇ 병목현상(BottleNeck)  
◇ 데드락(DeadLock)  
  
이러한 문제점들은 비단 스레드에서만 발생하는 것이 아니라 네트워크, 분산, 데이터베이스 관리 등 자원을 공유하는 곳이면 어디든지 발생한다.  
  
동기화란 여러 곳에서 자료를 요청하더라도 순서대로 자료를 사용한다는 의미를 지니고 있다. 동기화를 하기 위해서는 일단 특정 사용자가 자료를 사용한다면 다른 사용자는 기다려야 하고, 그리고 사용이 끝나면 대기하고 있던 사용자에게 알려 주어야 한다. 이러한 원리는 스레드에서도 그대로 적용된다.  
  
동기화를 화장실에 비유할 수 있다. 화장실에 변기가 하나밖에 없다고 가정하자. 사용하려는 사람은 10명이라고 가정한다면 10명이 차례대로 사용해야 한다. 만약 이를 어기면 난리가 날 것이다. 공유가 깨지는 것이다. 그래서 보통 우리는 어떻게 하는가? 문을 잠근다. lock을 거는 것이다. 아주 쉽다. 변기는 공유자원에 해당하고 잠그는 것은 lock에 해당한다.  
  
**□ 동기화(Synchronization)**  
◇ 스레드가 자원을 공유할 때 한번에 하나의 스레드만 사용  
◇ 하나의 스레드가 공유자원을 사용할 때 공유자원에 lock을 걸어준다  
  
만약 하나의 데이터를 공유하는데 동기화를 해두면 하나의 스레드만 이 데이터를 사용하게 된다. lock을 거는 것이다. 데이터를 사용하고 나면 lock을 해제할 것이고, 그리고 다른 스레드가 이 데이터를 사용하게 될 것이다. 동기화를 해두면 CLR이 알아서 처리하게 된다.  
  
그럼 본격적으로 C#에서의 동기화 방법에 어떤 것들이 있는지 알아보자. C#에서 지원하는 대표적인 동기화의 방법들은 다음과 같다.  
  
**□ 동기화를 위한 방법들**  
◇ lock 키워드의 사용  
◇ Threading 네임스페이스의 Monitor 클래스의 사용  
◇ Threading 네임스페이스의 Mutex 클래스의 사용  
  
위 세가지 방법은 비록 그 쓰이는 형태가 조금씩은 다르지만 모두 공유자원을 잠그고 해제할 수 있는 방법들을 제시한다. 다음 절부터 lock, Monitor, Mutex에 대해서 차례대로 알아보자.



**11.3.2 lock 동기화**  
  
11.2.4절의 멀티 스레드를 구현할 때 서로 다른 객체를 이용해서 ThreadStart 델리게이트를 만들었다. 이 절에서는 멀티 스레드 예제를 약간 변형해서 사용할 것이다. 하나의 객체로 두개의 ThreadStart를 생성한 후 스레드를 실행해보자.   
  
이렇게 될 경우 객체에 공유자원이 생긴다. 즉 하나의 객체에 존재하는 자원을 두개의 스레드가 동시에 액세스하기 때문에 문제가 발생한다.  
  
이러한 경우에 C#에서는 lock이라는 것을 사용해서 간단하게 스레드의 동기화를 제어할 수 있다. 우선 그 형식부터 살펴보자.  
  
**▒ lock 키워드의 형식**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3 | lock(expression) {     statement;  } |

lock 키워드는 블록(Statement Block)을 이용해서 임계영역(Critical Section)을 설정한다. expression은 lock을 걸 자원이 존재하는 위치를 의미한다. 임계영역 내의 보호자원에 접근하려고 할 때 단 하나의 스레드만이 임계영역에 들어갈 수 있으며 다른 스레드들은 임계영역에 들어가기 위해서 대기해야 한다.   
  
**□ lock 키워드**  
◇ 동기화를 구현하는 키워드  
◇ 동기화 객체는 참조타입이어야 함  
  
다음의 예제는 lock 키워드를 이용한 동기화의 기본적인 방법을 보여주고 있다.

**§ chap11\MultiThreadSyncTest.cs**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | /\*\*  공유자원에 lock을 거는 예제  \*\*/  using System;  using System.Threading;  public class MultiThreadSyncTest{      public static void Main(){          Top t = new Top();          ThreadStart ts1 = new ThreadStart(t.SayHello);          ThreadStart ts2 = new ThreadStart(t.SayHello);          Thread thread1= new Thread(ts1);          Thread thread2= new Thread(ts2);          thread1.Start();          thread2.Start();          Console.Write("\nThread " + Thread.CurrentThread.GetHashCode() + "메인 종료\n" );      }  }    public class Top{      private int limit = 0;      public void SayHello(){          int hash = Thread.CurrentThread.GetHashCode();          int count =0;          lock(this){              while(count<10){                  Console.WriteLine("Thread " + hash + ":" + limit++);                  count++;                  Thread.Sleep(10);              }          }//lock      }  }  /\*\*\*  Thread 3메인 종료  Thread 1:0  Thread 1:1  Thread 1:2  Thread 1:3  Thread 1:4  Thread 1:5  Thread 1:6  Thread 1:7  Thread 1:8  Thread 1:9  Thread 2:10  Thread 2:11  Thread 2:12  Thread 2:13  Thread 2:14  Thread 2:15  Thread 2:16  Thread 2:17  Thread 2:18  Thread 2:19    [참고] 순서대로 출력되는 이유: 하나의 스레드에서 공유자원 limit을 사용하고 있을 경우 다른 스레드에서 limit을 사용하기 위해서는 기다려야 한다. 이것은 공유자원에 lock을 걸어 두었기 때문이다.  \*\*\*/ |

위의 예제에서 Top 객체는 하나만 생성하였다. 이렇게 될 경우 Top 내의 SayHello() 함수에서 사용하는 limit은 여러분도 모르는 사이에 공유자원이 되어 버린다. 두개의 스레드가 모두 하나의 객체를 대상으로 하기 때문에 SayHello()에서 공통적으로 사용하고 있는 limit을 동시에 접근하려고 할 것이다. 이럴 경우 lock 키워드를 이용해서 limit을 동기화시키면 된다. 다음의 구문은 lock을 이용해서 해당 블록을 동기화시키는 구문이다.  
  
**▒ lock을 이용한 동기화**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3 | lock(this){     //공유자원 limit의 사용  } //lock |

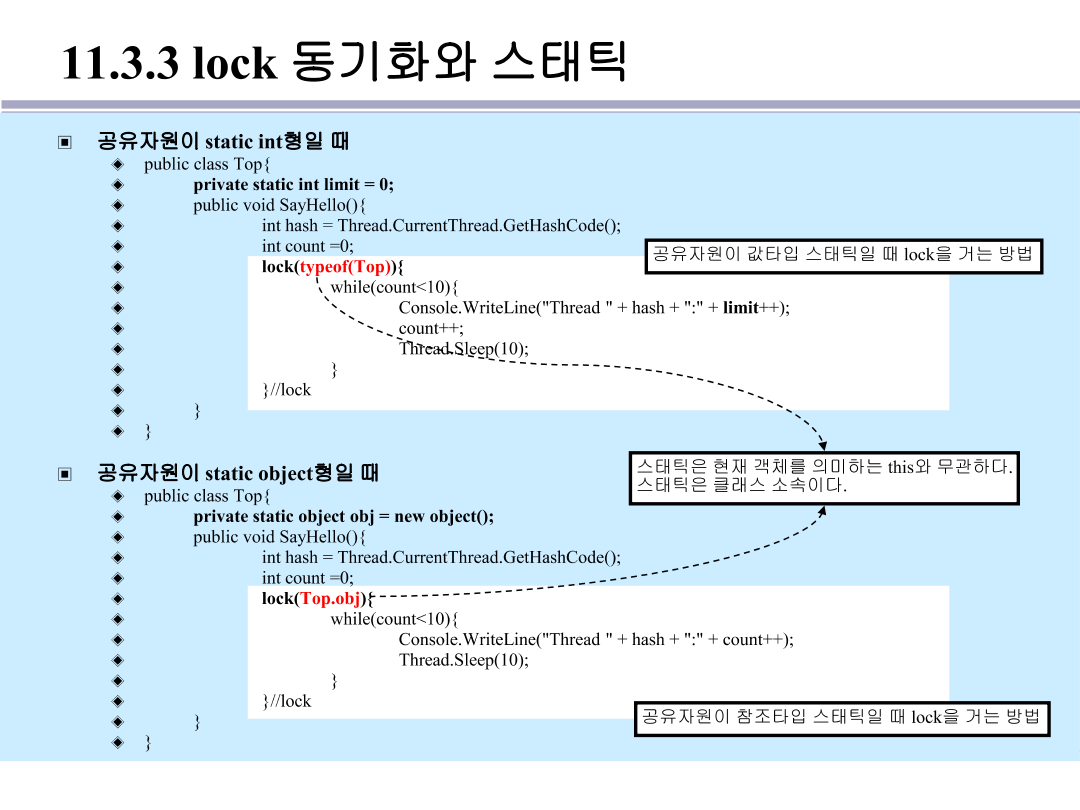
lock 블록 내에 존재하는 공유자원이 완벽하게 동기화가 된다는 것이 보장된다. 이 때 this의 의미는 공유자원이 존재하는 위치이다. 공유자원이 현재의 클래스 내에 존재한다는 것을 의미한다. 위에서 공유자원은 limit이며 limit은 현재의 클래스 내에 있는 것이다.   
  
원래 lock의 공유 대상은 limit이다. 하지만 lock(limit)과 같이 사용할 수 없다. 그 이유는 limit이 값타입이기 때문이다. lock의 대상은 반드시 참조타입이어야 한다. 그래서 limit이 존재하는 장소인 this를 사용한 것이다.  
  
lock에 접근하는 스레드가 아무리 많아도 오직 하나의 스레드만이 limit에 접근할 수 있으며, 특정 스레드가 limit을 사용하고 있으면 다른 스레드는 대기상태에서 기다려야 한다. 즉 limit이 잠겨버리는 것이다. lock에 걸린 것이다. 11.2.4절의 멀티 스레드의 생성 예와 비교해보기 바란다.  
  
만약 11.2.4의 예에 lock을 걸면 어떻게 될까? 두개의 객체를 이용해서 스레드를 생성하기 때문에 공유자원 자체가 없는 상태가 된다. 공유자원이 없는 곳에 lock을 걸면 일반적인 스레드와 별반 다를 것이 없다. 즉 lock은 있으나마나 한 존재가 된다. 다음은 공유자원이 없는 경우 lock을 거는 예제이다.

**§ chap11\MultiThreadSyncTest2.cs**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | /\*\*  공유자원이 없는 lock 테스트  \*\*/  using System;  using System.Threading;  public class MultiThreadSyncTest2{      public static void Main(){          Top t1 = new Top();          Top t2 = new Top();          ThreadStart ts1 = new ThreadStart(t1.SayHello);          ThreadStart ts2 = new ThreadStart(t2.SayHello);          Thread thread1= new Thread(ts1);          Thread thread2= new Thread(ts2);          thread1.Start();          thread2.Start();          Console.Write("\nThread " + Thread.CurrentThread.GetHashCode() + "메인 종료\n" );      } //main  } //class    public class Top{      private int limit = 0;      public void SayHello(){          int hash = Thread.CurrentThread.GetHashCode();          int count =0;          while(count<10){              lock(this){ //공유자원이 없기 때문에 필요가 없다.                  Console.WriteLine("Thread " + hash + ":" + limit++);                  count++;                  Thread.Sleep(10);              } //lock          }      }  } //class  /\*\*\*  Thread 2:0  Thread 3메인 종료  Thread 2:1  Thread 1:0  Thread 1:1  Thread 2:2  Thread 2:3  Thread 1:2  Thread 2:4  Thread 1:3  Thread 2:5  Thread 1:4  Thread 1:5  Thread 2:6  Thread 1:6  Thread 2:7  Thread 2:8  Thread 1:7  Thread 1:8  Thread 2:9  Thread 1:9    [참고] 공유자원 자체가 없기 때문에 Thread1과 Thread2는 동기화를 하더라도 전혀 영향을 받지 않는다.  \*\*\*/ |

하나의 Top 객체의 SayHello()를 이용해서 ThreadStart 델리게이트를 만들었기 때문에 공유자원이 생겼지만 두개의 Top 객체를 이용하면 공유자원이 존재하지 않는다. 위의 예제에서는 공유자원 자체가 없기 때문에 lock을 사용한 것이나 사용하지 않은 것이나 똑같은 결과가 나오게 된다. 다음 절에서는 스태틱으로 공유자원을 만드는 경우를 알아보자.



**11.3.3 lock 동기화와 스태틱**  
  
앞 절에서는 하나의 객체에 존재하는 함수로 스레드를 여러 개 만들었을 때 멤버 필드가 자동으로 공유자원이 되는 경우를 알아보았다. 이번 절에서는 스태틱의 특성상 하나의 메모리만 만들어지기 때문에 공유자원이 되는 경우를 알아보자.   
  
**□ 스태틱 멤버 필드**  
◇ 스태틱의 특성상 하나의 메모리만 만들어지기 때문에 멀티 스레드에서 스태틱에 접근하면 자동으로 스태틱은 공유자원이 된다.  
  
주의해야 되는 것은 lock을 걸 때 스태틱 멤버 필드는 this와 전혀 상관이 없다는 것이다. 그래서 스태틱 멤버 필드는 this를 사용해서 lock을 걸 수 없다. 그렇다면 직접적으로 lock의 대상을 정해주어야 한다. 만약 스태틱 멤버 필드가 참조타입이라면 참조타입 그 자체를 lock의 대상으로 사용하면 되지만 값타입이라면 골치 아파진다.  
  
**□ 스태틱 멤버 필드를 공유자원으로 사용할 경우의 문제점**  
◇ 값타입 스태틱 멤버 필드에는 this를 사용할 수 없다.  
◇ 참조타입이면 lock의 대상으로 참조타입의 참조값 그 자체를 사용하면 된다.  
  
스태틱은 클래스 명으로 접근하기 때문에 공유자원이 현재의 클래스에 있다는 this를 사용할 수 없다. 그렇다고 값타입인 limit을 그대로 lock의 대상으로 삼을 수는 없다. 이러한 경우에는 typeof(클래스명) 형식으로 lock을 잡아주면 된다. 즉 해당 클래스의 Type 객체를 넣어주면 되는 것이다.   
  
**□ 값타입 스태틱 멤버 필드의 lock**  
◇ 값타입의 Type 객체를 lock의 대상으로 넣어주면 된다.  
◇ typeof(클래스명) 형식으로 lock을 잡아주면 된다.  
  
아래의 예제는 두개의 객체를 생성한 후 각각의 스레드를 만들어주고 있다. 스태틱 멤버 필드를 사용해서 전역적인 공유자원이 되는 경우의 예를 보여주고 있다. lock의 사용법을 유심히 살펴보자.

**§ chap11\MultiThreadSyncStaticTest.cs**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | /\*\*  스태틱 값타입의 동기화  \*\*/  using System;  using System.Threading;  public class MultiThreadSyncStaticTest{      public static void Main(){          Top t1 = new Top();          Top t2 = new Top();          ThreadStart ts1 = new ThreadStart(t1.SayHello);          ThreadStart ts2 = new ThreadStart(t2.SayHello);          Thread thread1= new Thread(ts1);          Thread thread2= new Thread(ts2);          thread1.Start();          thread2.Start();          Console.Write("\nThread " + Thread.CurrentThread.GetHashCode() + "메인 종료\n" );      }  }    public class Top{      private static int limit = 0;      public void SayHello(){          int hash = Thread.CurrentThread.GetHashCode();          int count =0;          lock(typeof(Top)){              while(count<10){                  Console.WriteLine("Thread " + hash + ":" + limit++);                  count++;                  Thread.Sleep(10);              }          }//lock      }  }  /\*\*\*  Thread 3메인 종료  Thread 1:0  Thread 1:1  Thread 1:2  Thread 1:3  Thread 1:4  Thread 1:5  Thread 1:6  Thread 1:7  Thread 1:8  Thread 1:9  Thread 2:10  Thread 2:11  Thread 2:12  Thread 2:13  Thread 2:14  Thread 2:15  Thread 2:16  Thread 2:17  Thread 2:18  Thread 2:19    [참고] 값타입 스태틱 멤버필드에 lock을 거는 방법을 테스트하는 예제이다.  \*\*\*/ |

일반적인 멀티 스레드를 만드는 방식과 같은 예제이다. 하지만 멤버 필드에 값타입의 스태틱 공유자원이 존재한다.  
  
**▒ 값타입의 스태틱 공유자원**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1 | private static int limit = 0; |

스태틱 공유자원이 값타입이기 때문에 다음과 같이 사용할 수는 없다.  
  
**▒ 잘못된 lock의 사용 I**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1 | lock(Top.limit) //에러발생 |

만약 limit이 참조타입이라면 위와 같이 사용할 수 있지만 limit이 값타입이기 때문에 lock의 대상이 될 수 없다. 그리고 this를 사용해도 스태틱에는 영향을 미치지 못한다.  
  
**▒ 잘못된 lock의 사용 II**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1 | lock(this) //스태틱 공유자원의 동기화와 관련이 없음 |

이와 같이 공유자원이 스태틱이면서 값타입이면 Type 객체를 만들어서 넣어주면 해당 스태틱 값타입의 공유자원은 동기화가 보장된다.  
  
**▒ 스태틱 변수의 lock**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

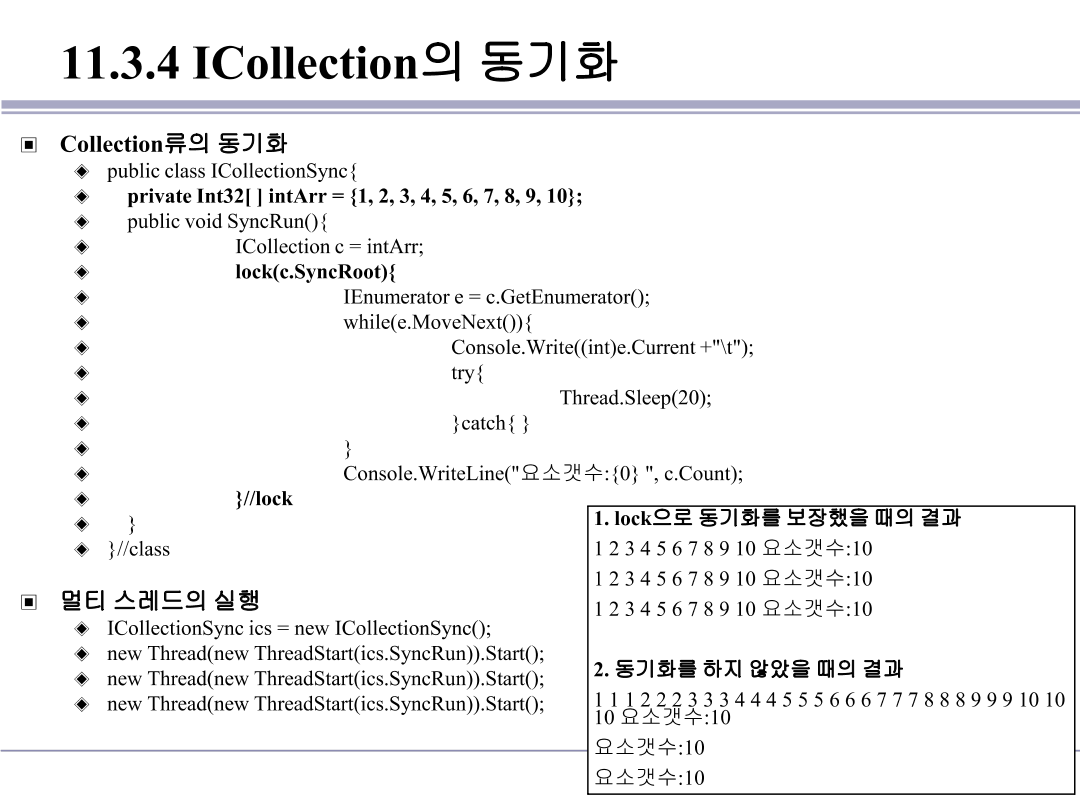
|  |  |
| --- | --- |
| 1 | lock(typeof(Top)) //동기화 보장됨 |

만약 공유자원이 스태틱이면서 참조타입이라면 다음의 예제와 같이 lock을 걸 때 해당 참조타입의 객체를 lock의 대상으로 하면 된다.

**§ chap11\MultiThreadSyncStaticTest2.cs**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | /\*\*  스태틱 참조타입의 동기화  \*\*/  using System;  using System.Threading;  public class MultiThreadSyncStaticTest2{      public static void Main(){          Top t1 = new Top();          Top t2 = new Top();          ThreadStart ts1 = new ThreadStart(t1.SayHello);          ThreadStart ts2 = new ThreadStart(t2.SayHello);          Thread thread1= new Thread(ts1);          Thread thread2= new Thread(ts2);          thread1.Start();          thread2.Start();          Console.Write("\nThread " + Thread.CurrentThread.GetHashCode() + "메인 종료\n" );      }  }    public class Top{      private static object obj = new object();      public void SayHello(){          int hash = Thread.CurrentThread.GetHashCode();          int count =0;          lock(Top.obj){              while(count<10){                  Console.WriteLine("Thread " + hash + ":" + count++);                  Thread.Sleep(10);              }          }//lock      }  }  /\*\*\*  Thread 3메인 종료  Thread 1:0  Thread 1:1  Thread 1:2  Thread 1:3  Thread 1:4  Thread 1:5  Thread 1:6  Thread 1:7  Thread 1:8  Thread 1:9  Thread 2:0  Thread 2:1  Thread 2:2  Thread 2:3  Thread 2:4  Thread 2:5  Thread 2:6  Thread 2:7  Thread 2:8  Thread 2:9    [참고] 스태틱 멤버필드가 참조타입이라면 lock의 대상으로 참조변수를 그대로 사용하면 된다.  \*\*\*/ |



**11.3.4 ICollection의 동기화**  
  
ICollection 계열의 클래스일 경우에 내부에 많은 요소(Element)를 포함할 수 있다. 이들에 대한 동기화를 위해서 ICollection에서는 SyncRoot라는 속성을 지원해준다.  
  
**□ SyncRoot**  
◇ ICollection 계열의 클래스들에 대한 동기화를 지원하는 ICollection의 속성  
  
SyncRoot의 사용법은 간단하다. ICollection 계열의 클래스일 경우 단순히 SyncRoot라는 속성 자체에 lock을 걸면 된다. lock을 거는 방법은 다음과 같다.  
  
**▒ ICollectionSync 클래스**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class ICollectionSync{     private Int32[ ] intArr = {1,2,3,4, 5, 6, 7, 8, 9, 10}; //공유자원     public void SyncRun(){        ICollection c = intArr;        lock(c.SyncRoot){           //....        }     }  } |

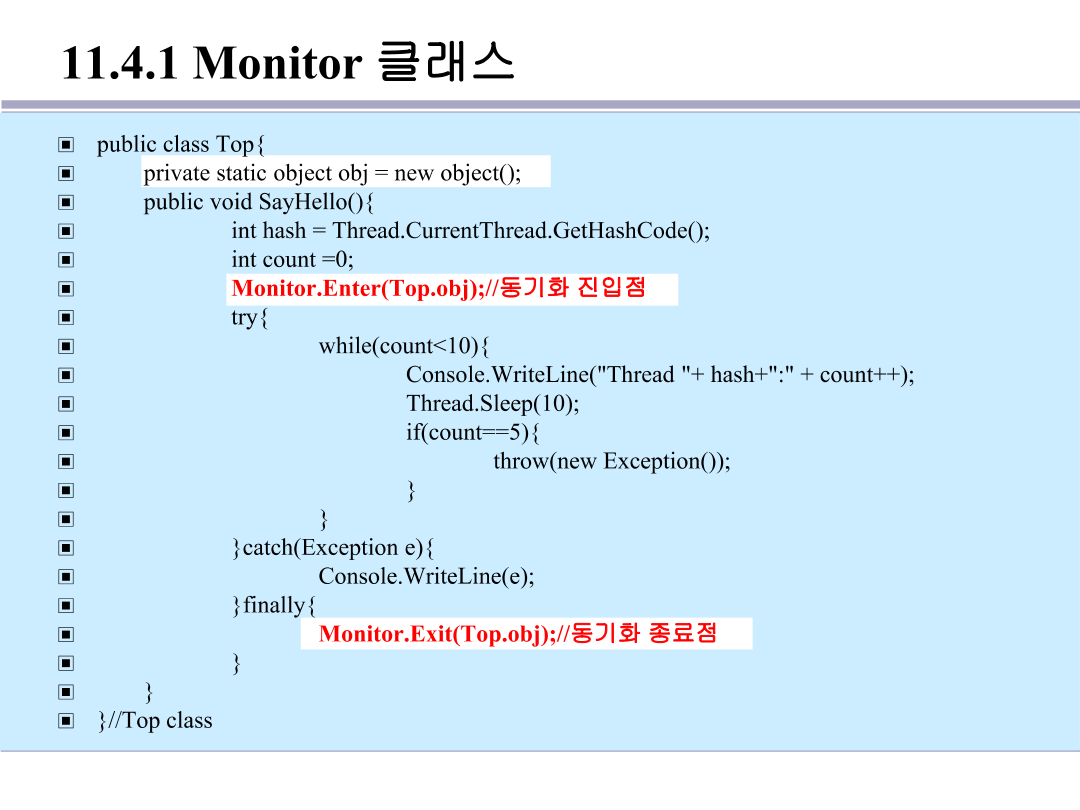
다음은 ICollection의 동기화를 테스트하는 있는 예를 보여주고 있다.

**§ chap06\ICollectionSyncTest1.cs**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | /\*\*  ICollection의 동기화를 테스트하는 예제  \*\*/  using System;  using System.Collections;  using System.Threading;  public class ICollectionSync{      private Int32[] intArr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};        public void SyncRun(){          ICollection c = intArr;//배열은 Array을 상속 -> IColleciton으로 캐스팅 가능          //lock(c.SyncRoot){              IEnumerator e = c.GetEnumerator();              while(e.MoveNext()){                  Console.Write((int)e.Current +"\t");                  try{                      Thread.Sleep(20);                  }catch{ }              }              Console.WriteLine("요소갯수:{0} ", c.Count);          //}//lock      }  }//class    public class ICollectionSyncTest1{      public static void Main(){          ICollectionSync ics = new ICollectionSync();          new Thread(new ThreadStart(ics.SyncRun)).Start();          new Thread(new ThreadStart(ics.SyncRun)).Start();          new Thread(new ThreadStart(ics.SyncRun)).Start();      }//main  }//class  /\*\*\*  1. lock으로 동기화를 보장했을 때의 결과  1   2   3   4   5   6   7   8   9   10  요소갯수:10  1   2   3   4   5   6   7   8   9   10  요소갯수:10  1   2   3   4   5   6   7   8   9   10  요소갯수:10    2. 동기화를 하지 않았을 때의 결과  1   1   1   2   2   2   3   3   3   4   4   4   5   5   5   6   6   6   7   7   7   8   8   8   9   9   9   10  10  10  요소갯수:10  요소갯수:10  요소갯수:10  \*\*\*/ |

결과를 비교해보면 동기화에 대한 느낌을 정확하게 이해할 수 있을 것이다. 동기화란 하나의 자원을 여러 스레드가 이용할 때 동시에 자원을 이용하는 것이 아니라 하나의 스레드가 자원을 점유하고 있으면 다른 스레드들은 기다리는 원리이다. 그래서 차례대로 일처리가 되는 것이다.  
  
ICollection 인터페이스에서는 이러한 동기화를 위한 자원 관리를 위해서 SyncRoot라는 속성을 제공해주고 있는 것이다.



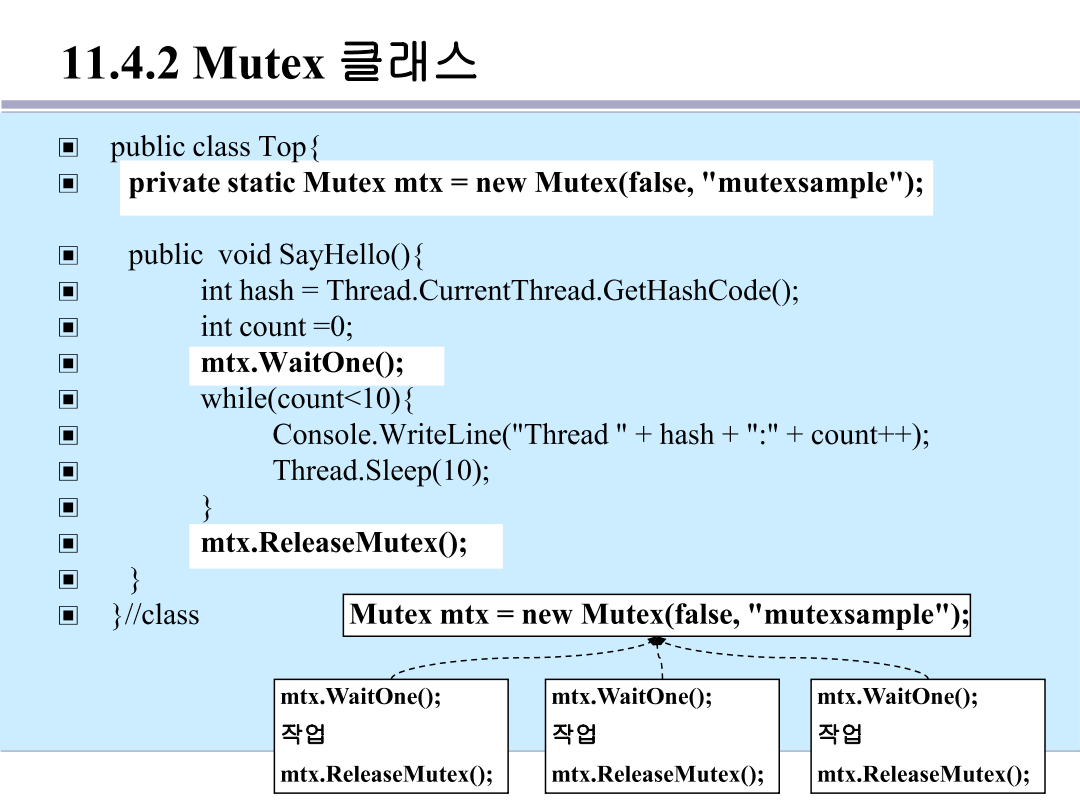
**11.4.1 Monitor 클래스**  
  
System.Threading 네임스페이스에 정의된 Monitor 클래스는 lock 키워드보다 좀더 정교한 동기화의 방법을 제공한다. Monitor는 임계영역(Critical Section)의 시작점과 끝점을 임의적으로 조정할 수 있으며, 임계영역에서 보호받을 공유자원 또한 설정할 수 있다.  
  
**□ Monitor 클래스**  
◇ lock 키워드보다 좀더 정교한 동기화의 방법을 제공  
  
Monitor 클래스는 sealed 클래스이며 lock과 아주 유사한 역할을 한다. lock 키워드를 사용해서 동기화하는 것과 무엇이 다를까?  
  
lock 키워드를 사용하면 그 효력은 블록({ })으로 감싸있는 내부 전체가 된다. 이에 반해 Monitor 클래스를 사용해서 동기화시키면 Moinitor.Enter() 함수로 동기화의 시작점을 지정할 수 있고, Monitor.Exit() 함수를 이용해서 동기화의 끝 지점을 지정할 수 있다.  
  
**□ Monitor의 특징**  
◇ lock 키워드를 사용할 때의 동기화 영역은 블록({ })으로 표시  
◇ Monitor는 Monitor.Enter() 함수로 동기화의 시작을 지정할 수 있다.  
◇ Monitor.Exit() 함수로 동기화가 끝났음을 지정할 수 있다.  
◇ Monitor는 동기화의 시작과 끝을 유연하게 조정할 수 있다.  
◇ 단순히 lock을 통해 잠금을 설정하고 해제하는 것에 비해서 기능이 많음  
◇ Monitor의 멤버는 대부분 static으로 구성되어 있다.  
  
이것은 복잡한 코드에서 동기화가 필요한 미세한 구역만을 골라서 처리해 줄 수 있다는 장점이 있지만, 문맥을 잘 따져서 언제 동기화를 풀어주는지에 대한 고민도 필요하다.  
  
lock에 비해서 좀더 나은 점은 TryEntry(), Enter(), Exit() 함수와 같이 좀더 풍부한 함수를 제공하기 때문에 단순히 lock을 통해 잠금을 설정하고 해제하는 것에 비해서 기능이 많다는 장점이 있다. TryEntry()는 해당 동기화 객체를 잠그다가 실패하면 잠그는 것을 포기하고 나오는 역할을 하는 함수이다. 이것은 lock과는 달리 결코 Blocking되지 않는 장점이 있다.  
  
예제를 보고 좀더 알아보자. 다음 예제는 앞의 lock 키워드를 사용한 동기화를 단순히 Monitor 클래스를 사용해서 동기화하는 것으로 바꾼 것이다.

**§ chap11\ThreadMonitorTest.cs**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | /\*\*  Monitor클래스를 이용한 동기화를 테스트하는 예제  \*\*/  using System;  using System.Threading;  public class ThreadMonitorTest{      public static void Main(){          Top t1 = new Top();          Top t2 = new Top();          ThreadStart ts1 = new ThreadStart(t1.SayHello);          ThreadStart ts2 = new ThreadStart(t2.SayHello);          Thread thread1= new Thread(ts1);          Thread thread2= new Thread(ts2);          thread1.Start();          thread2.Start();          Console.Write("\nThread " + Thread.CurrentThread.GetHashCode() + "메인 종료\n" );      }  }// ThreadMonitorTest class    public class Top{      private static object obj = new object();      public void SayHello(){          int hash = Thread.CurrentThread.GetHashCode();          int count =0;          Monitor.Enter(Top.obj);//동기화 진입점          try{              while(count<10){                  Console.WriteLine("Thread " + hash + ":" + count++);                  Thread.Sleep(10);                  if(count == 5){                      throw(new Exception());                  }              }          }catch(Exception e){              Console.WriteLine(e);          }finally{              Monitor.Exit(Top.obj);//동기화 종료점          }      }  }//Top class  /\*\*\*  Thread 3메인 종료  Thread 1:0  Thread 1:1  Thread 1:2  Thread 1:3  Thread 1:4  System.Exception: System.Exception 형식의 예외가 Throw되었습니다.     at Top.SayHello()  Thread 2:0  Thread 2:1  Thread 2:2  Thread 2:3  Thread 2:4  System.Exception: System.Exception 형식의 예외가 Throw되었습니다.     at Top.SayHello()  \*\*\*/ |

Moniter.Enter() 함수로 시작한 동기화는 반드시 Moniter.Exit() 함수를 만나야 끝이 난다. Moniter.Exit() 함수를 생략하면, 동기화의 종료점이 없기 때문에 즉 해당 스레드가 작업할 기회를 얻지 못하므로 프로그램은 데드락(Deadlock)이 발생할 수 있다. 앞뒤의 문맥을 따져서 정확한 위치에 Moniter.Exit()를 호출해서 반드시 동기화의 종료지점을 정해 주어야 한다.



**11.4.2 Mutex 클래스**  
  
스레드가 공유자원을 사용하도록 하는 동기화 메커니즘으로 Mutex 클래스를 사용할 수 있다. Mutex를 생성한 후 생성된 Mutex의 사용 권한을 하나의 스레드에게만 부여하는 동기화 방식이다.  
  
**□ Mutex 클래스**  
◇ 여러 개의 스레드에서 Mutex에 대한 사용을 요청할 때 사용권한을 하나의 스레드에게만 부여하는 방식  
  
프로세스는 공유자원을 사용할 경우에 Mutex에게 소유권을 요청한다. 이 때 공유자원을 사용하는 스레드가 있다면 뮤텍스 객체는 공유자원이 사용 중이므로 기다리라는 신호를 보내고, 공유자원의 사용이 끝나면 Mutex는 사용을 요청했던 대기 중인 스레드에게 공유자원을 사용할 수 있다는 신호를 세팅하는 방식으로 동기화가 구현된다.  
  
**▣ 1단계 Mutex 객체 생성하기**  
  
뮤텍스를 사용하기 위해서는 우선 임의의 Mutex 객체를 하나 생성해야 한다.   
  
**▒ Mutex 객체 생성하기**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1 | Mutex mtx = new Mutex(false,"mutexsample"); |

첫번째 매개변수는 Mutex가 생성되자마자 신호를 받은 상태인지 아닌지를 결정한다. 그리고 두번째 매개변수는 해당 뮤텍스의 이름을 의미한다.  
  
**▣ 2단계 Mutex 요청하기**  
  
만들어진 뮤텍스를 사용할 권한을 요청하게 된다. 만약 다른 스레드가 mtx를 사용하고 있다면 대기상태에 있으면서 사용권한 요청을 반복적으로 하게 된다. 이 때 사용권한 요청은 WaitOne() 함수를 이용한다.  
  
**▒ 뮤텍스의 사용권한 요청**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1 | mtx.WaitOne(); |

**▣ 3단계 Mutex 해제하기**  
  
만약 작업을 끝마쳤다면 mtx의 사용권한을 해제해주어야 한다. WaitOne()한 수만큼 ReleaseMutex()를 호출해야 한다.  
  
**▒ 뮤텍스의 사용권한 해제**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1 | mtx.ReleaseMutex(); |

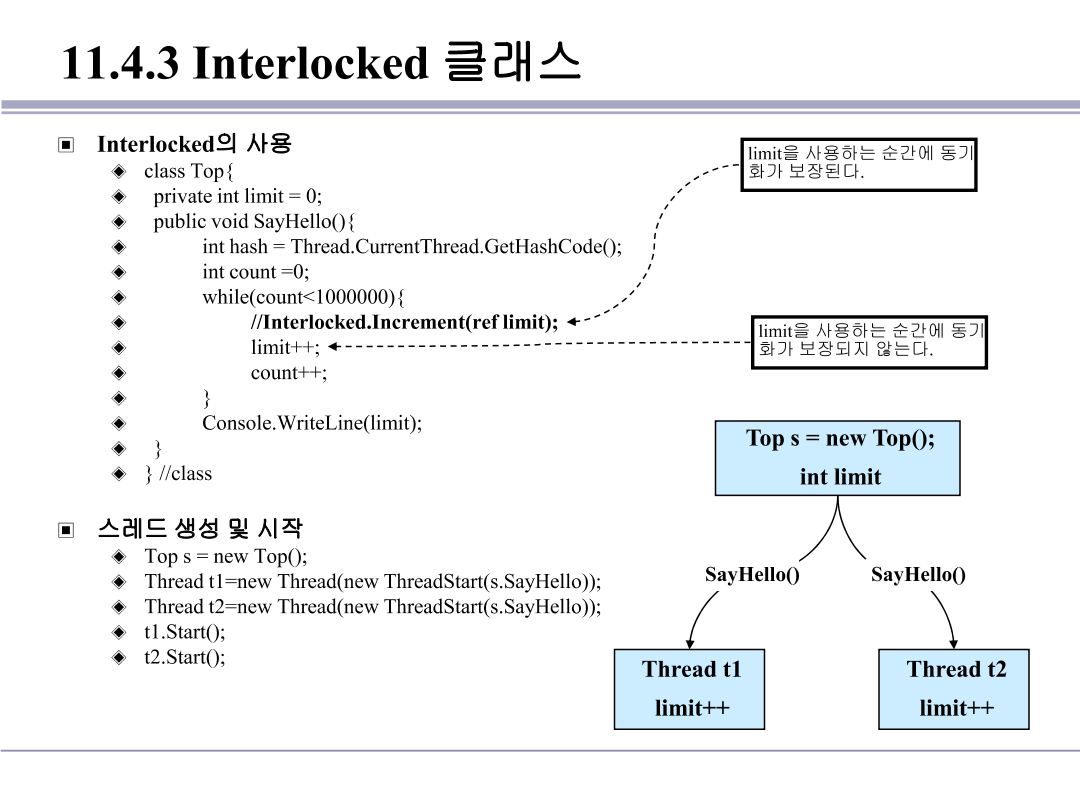
실제로 예제를 통해서 Mutex를 사용하는 방법에 대해서 알아보자. 다음 예제를 보자.

**§ chap11\ThreadMutexTest.cs**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | /\*\*  Mutex를 이용한 스레드의 동기화를 테스트하는 예제  \*\*/  using System;  using System.Threading;  public class ThreadMutexTest{      public static void Main(){          Top t1 = new Top();          Top t2 = new Top();          ThreadStart ts1 = new ThreadStart(t1.SayHello);          ThreadStart ts2 = new ThreadStart(t2.SayHello);          Thread thread1= new Thread(ts1);          Thread thread2= new Thread(ts2);          thread1.Start();          thread2.Start();          Console.Write("\nThread " + Thread.CurrentThread.GetHashCode() + "메인 종료\n" );      }//main  }//class    public class Top{      private static Mutex mtx = new Mutex(false, "mutexsample");      public void SayHello(){          int hash = Thread.CurrentThread.GetHashCode();          int count =0;          mtx.WaitOne();          while(count<10){              Console.WriteLine("Thread " + hash + ":" + count++);              Thread.Sleep(10);          }          mtx.ReleaseMutex();      }  }//class  /\*\*\*  Thread 3메인 종료  Thread 1:0  Thread 1:1  Thread 1:2  Thread 1:3  Thread 1:4  Thread 1:5  Thread 1:6  Thread 1:7  Thread 1:8  Thread 1:9  Thread 2:0  Thread 2:1  Thread 2:2  Thread 2:3  Thread 2:4  Thread 2:5  Thread 2:6  Thread 2:7  Thread 2:8  Thread 2:9  \*\*\*/ |

위의 예제에서 mtx의 WaitOne()으로 누가 먼저 권한을 얻느냐에 따라서 동기화 순서가 결정된다. 하지만 위에서는 공유자원은 존재하지 않는다. 잘 생각해보면 Mutex 자신을 공유자원처럼 사용하고 있다. 공유자원이 존재하지 않아도 Mutex 자체만으로도 동기화를 구현할 수 있다.  
  
공유자원이 없어도 WaitOne() 영역부터 ReleaseMutex() 영역까지 동기화가 보장된다. 하나의 스레드가 Mutex를 차지했다면 다른 스레드들은 Mutex의 권한 요청을 얻을 때까지 대기해야 한다. 이 때 권한 요청은 WaitOne()을 이용하며, Mutex의 권한을 얻었을 때 작업할 수 있다. 그리고 권한을 해제하기 위해서는 ReleaseMutex()를 이용한다.



**11.4.3 Interlocked 클래스**  
  
Interlocked 클래스는 여러 개의 스레드가 하나의 정수 데이터를 공유할 때 정수 데이터의 값이 증가하고 감소하는 작업 자체를 동기화시킨다. 이 클래스의 주요 함수를 살펴보면 다음과 같다.  
  
**□ Interlocked 클래스의 함수 - Increment()**  
◇ 동기화를 보장하면서 지정된 변수의 값을 증가시키고 저장  
◇ 변수의 참조를 나타내는 ref를 사용하므로 지정된 변수의 값이 바뀜  
  
**□ Interlocked 클래스의 함수 - Decrement()**  
◇ 동기화를 보장하면서 지정된 변수의 값을 감소시키고 저장  
  
정수형의 데이터를 증가 또는 감소시키면서 자동으로 동기화를 지원하는 Interlocked 기법이다. 다음 예제는 Interlocked 클래스를 이용하는 방법을 보여주고 있다.

**§ chap11\InterlockedTest.cs**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

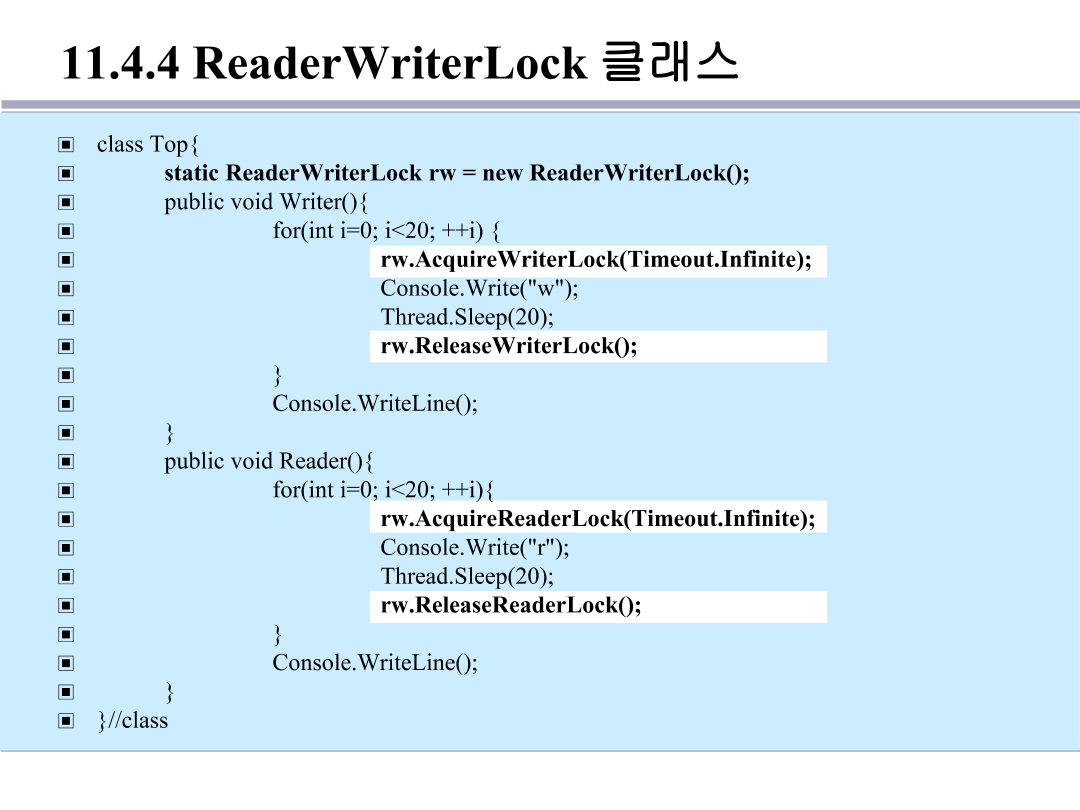
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | /\*\*  Interlocked를 이용한 동기화 보장  \*\*/  using System;  using System.Threading;  class InterlockedTest{      public static void Main(){          Top s = new Top();          Thread t1 = new Thread(new ThreadStart(s.SayHello));          Thread t2 = new Thread(new ThreadStart(s.SayHello));          t1.Start();          t2.Start();      } //main  } //class    class Top{      private int limit = 0;      public void SayHello(){          int hash = Thread.CurrentThread.GetHashCode();          int count =0;          while(count<1000000){                  Interlocked.Increment(ref limit);//limit을 사용하는 순간에 동기화가 보장된다.                  //limit++; //limit을 사용하는 순간에 동기화가 보장되지 않는다.                  count++;          }          Console.WriteLine(limit);      }  } //class  /\*\*\*  1. Interlocked.Increment(ref limit)으로 실행할 때  1987429  2000000  [참고] 마지막 값은 무조건 2000000이 된다. 첫번째 스레드가 완료되었을 때는 임의의 값이 나오지만 두번째 스레드가 완료되는 2000000이 된다. 1000000을 두번 실행했기 때문에 동기화가 보장이 되었다면 2000000이 되는 것이다.    2. limit++으로 실행할 때  1182104  1799100  [참고] 동기화가 보장되지 않았기 때문에 임의의 값이 나온다.  \*\*\*/ |

여기서는 Interlocked 클래스의 Increment 함수를 사용해서 동기화를 구현해주었다.   
  
**▒ Interlocked를 이용해서 동기화를 보장하면서 값 증가시키기**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1 | Interlocked.Increment(ref limit); |

반대의 경우에는 Decrement() 함수를 사용하면 된다.



**11.4.4 ReaderWriterLock 클래스**  
  
ReaderWriterLock은 다른 동기화 방법들과 약간 다른 면을 보이고 있다. Mutex를 예를 들면 하나의 스레드가 뮤텍스의 사용권한을 얻었다면 무조건적으로 다른 스레드는 사용권한을 얻기 위해서 대기해야 한다. 하지만 ReaderWriterLock은 Reader 권한을 얻었다면 Writer를 할 수 없다. 즉 Reader 권한을 해제할 때까지 Writer의 권한을 얻을 수 없기 때문에 대기상태에 있게 된다. Writer는 그 반대의 경우가 된다.   
  
**□ Reader 권한을 얻었다면 Writer를 할 수 없다.**  
◇ Reader 사용권한 얻기: AcquireReaderLock()  
◇ Reader 사용권한 해제: ReleaseReaderLock()  
  
**□ Writer 권한을 얻었다면 Reader를 할 수 없다.**  
◇ Writer 사용권한 얻기: AcquireWriterLock()  
◇ Writer 사용권한 해제: ReleaseWriterLock()  
  
**□ ReaderWriterLock의 특징**  
◇ Reader와 Writer는 둘 중 하나만 사용할 수 있도록 동기화시켜 준다.  
◇ Reader 권한을 두개 얻어서 실행시키면 2개의 Reader 권한끼리는 동기화가 보장되지 않는다.  
  
Reader 권한을 얻기 위해서 ReaderWriterLock의 AcquireReaderLock() 함수를 사용한다. 그리고 Writer 권한을 얻기 위해서는 AcquireWriterLock() 함수를 사용한다. 작업을 다 한 후에는 ReleaseWriterLock()과 ReleaseReaderLock() 함수를 호출해서 권한을 해제해주어야 한다. 다음의 예는 ReaderWriterLock의 사용 예를 보여주고 있다.

**§ chap11\ReaderWriterLockTest.cs**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

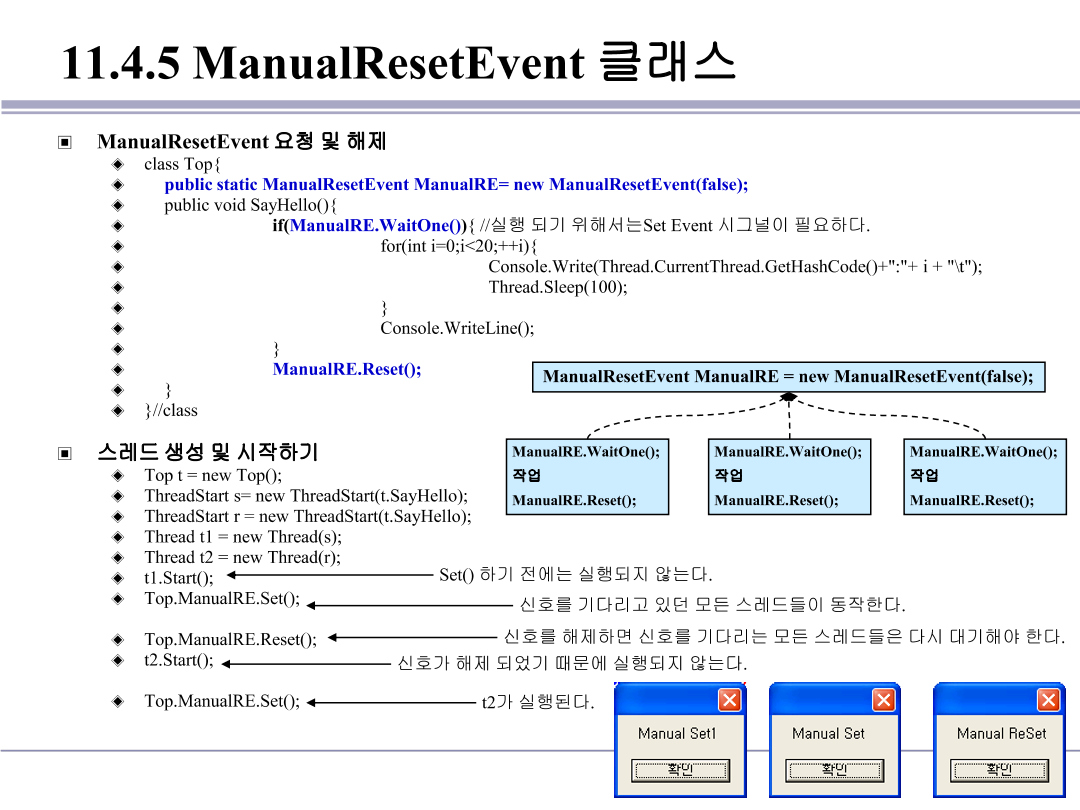
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | /\*\*  ReaderWriterLock 예제  \*\*/  using System;  using System.Threading;  public class ReaderWriterLockTest{      public static void Main(){          Console.Write("Reading-r, Writing-w : ");          Top t = new Top();          ThreadStart s= new ThreadStart(t.Writer);          ThreadStart r = new ThreadStart(t.Reader);          Thread t1 = new Thread(s);          Thread t2 = new Thread(r);          t1.Start();          t2.Start();      }//main  }//class    class Top{      static ReaderWriterLock rw = new ReaderWriterLock();      public void Writer(){          for(int i=0; i<20; ++i){              rw.AcquireWriterLock(Timeout.Infinite);              Console.Write("w");              Thread.Sleep(20);              rw.ReleaseWriterLock();          }          Console.WriteLine();      }      public void Reader(){          for(int i=0; i<20; ++i){              rw.AcquireReaderLock(Timeout.Infinite);              Console.Write("r");              Thread.Sleep(20);              rw.ReleaseReaderLock();          }          Console.WriteLine();      }  }//class  /\*\*\*  Reading-r, Writing-w : wrwrwrwrwrwrwrwrwrwrwrwrwrwrwrwrwrwrwrwr  \*\*\*/ |

Mutex와 비슷하지만 Mutex는 사용권한이 하나인데 비해서 ReaderWriterLock은 Reader와 Writer의 권한으로 양분되어 있어서 어느 한쪽을 사용하면 다른 쪽은 사용하지 못하는 방식의 동기화를 지원하고 있다. 다음은 락(Lock)을 걸고 해제하는 코드이다.  
  
**▒ ReaderWriterLock을 이용한 사용권한 요청과 해제**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3  4 | rw.AcquireWriterLock(Timeout.Infinite); //사용권한 요청  rw.ReleaseWriterLock(); //권한 해제  rw.AcquireReaderLock(Timeout.Infinite); //사용권한 요청  rw.ReleaseReaderLock(); //권한 해제 |

AcquireWriterLock() 함수와 ReleaseWriterLock() 함수는 쌍으로 이용된다. 만약 AcquireWriterLock()으로 사용권한을 얻은 후 ReleaseWriterLock() 함수를 호출하지 않는다면 AcquireReaderLock()을 이용해서 사용권한을 얻을 수 없는 상태에 빠지게 된다.



**11.4.5 ManualResetEvent 클래스**  
  
어떠한 시그널(Signal)를 받을 때까지 스레드를 대기상태로 두려면 ManualResetEvent 클래스를 사용한다. ManualResetEvent에서 스레드에게 시그널을 보내기 위해서는 ManualResetEvent의 Set() 함수를 사용한다. 그리고 스레드에서 이 시그널을 대기하기 위해서는 WaitOne() 함수를 사용한다.  
  
**□ ManualResetEvent 클래스**  
◇ 스레드들이 ManualResetEvent에 실행요청을 했을 때 시그널에 따라 실행여부가 결정된다.  
◇ 시그널을 설정하는 것은 Set()이며 시그널을 해제하는 것은 Reset()이다.   
◇ 이 시그널은 수동으로 설정된다.  
◇ 시그널이 Set으로 설정되면 모든 스레드는 자동으로 실행된다.  
  
WaitOne()은 지속적으로 시그널을 대기하게 되며 시그널을 받을 때까지 해당 스레드를 대기상태로 만든다. Set()으로 시그널을 세팅하면 Reset()을 호출하기 전에는 계속해서 시그널이 Set된 상태가 된다.  
  
다음의 예는 두개의 스레드에서 시그널을 대기하는 ManualResetEvent의 WaitOne()을 호출하고 있다. 그리고 프로그램에서 메시지 박스 표시 후 Set()을 이용해서 시그널을 보냈을 때 모든 스레드들은 시그널을 받은 상태가 되어 일제히 스레드가 동작하게 된다.

**§ chap11\ManualResetEventTest.cs**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | /\*\*  ManualResetEvent 예제  \*\*/  using System;  using System.Threading;  using System.Windows.Forms;  public class ManualResetEventTest{      public static void Main(){          Top t = new Top();          ThreadStart s= new ThreadStart(t.SayHello);          ThreadStart r = new ThreadStart(t.SayHello);          Thread t1 = new Thread(s);          Thread t2 = new Thread(r);          t1.Start();          t2.Start();          MessageBox.Show("Manual Set1");          Top.ManualRE.Set();      }  }    class Top{      public static ManualResetEvent ManualRE= new ManualResetEvent(false);      public void SayHello(){          if(ManualRE.WaitOne()){  //실행 되기 위해서는Set Event 시그널이 필요하다.              for(int i=0; i<20; ++i){                  Console.Write(Thread.CurrentThread.GetHashCode() + ":" + i + "\t");                  Thread.Sleep(100);              }              Console.WriteLine();          }      }  }//class  /\*\*\*  그림1  3:0     4:0     4:1     3:1     4:2     3:2     4:3     3:3     4:4     3:4     4:5     3:5     3:6     4:6     3:7     4:7     3:8     4:8     3:9     4:9     3:10    4:10    4:11    3:11    4:12    3:12    4:13    3:13    4:14    3:14     4:15    3:15    4:16    3:16    4:17    3:17    3:18    4:18    3:19    4:19  \*\*\*/ |

그림1 

스레드에서 시그널을 대기하는 WaitOne()이 호출되었다. 만약 시그널이 Set된다면 WaitOne()은 true를 반환한 후 스레드의 작업을 진행할 것이다. 주의할 것은 위의 예에서는 두개의 스레드가 모두 동작하고 있다는 사실이다. 시그널을 받은 모든 스레드가 동작하는 것이다. 이것은 한번 시그널이 세팅되면 Reset을 해주기 전까지는 계속적으로 시그널이 세팅되어 있다는 증거이다.  
  
이러한 경우에 다시 시그널을 해제해주어야 한다. ManualResetEvent라는 이름의 특성상 Set()을 했다면 수동으로 Reset()을 해주지 않으면 시그널이 계속해서 세팅되어 있게 된다. 이것이 바로 ResetEvent 앞에 Manual이 붙어있는 이유이다.  
  
다음의 경우는 Reset()을 이용해서 시그널을 해제하고 다시 Set()을 하는 예를 보여주고 있다.

**§ chap11\ManualResetEventTest2.cs**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | /\*\*  ManualResetEvent 예제2  \*\*/  using System;  using System.Threading;  using System.Windows.Forms;  public class ManualResetEventTest2{      public static void Main(){          Top t = new Top();          ThreadStart s = new ThreadStart(t.SayHello);          ThreadStart r = new ThreadStart(t.SayHello);          Thread t1 = new Thread(s);          Thread t2 = new Thread(r);          t1.Start();          MessageBox.Show("Manual Set");          Top.ManualRE.Set();          MessageBox.Show("Manual ReSet");          Top.ManualRE.Reset();          t2.Start();          MessageBox.Show("Manual Set");          Top.ManualRE.Set();      }  }    class Top{      public static ManualResetEvent ManualRE = new ManualResetEvent(false);      public void SayHello(){          if(ManualRE.WaitOne()){  //실행 되기 위해서는Set Event 시그널이 필요하다.              for(int i=0; i<20; ++i){                  Console.Write(Thread.CurrentThread.GetHashCode() + ":" + i + "\t");                  Thread.Sleep(100);              }              Console.WriteLine();          }          ManualRE.Reset();      }  }//class  /\*\*\*  그림1  3:0     3:1     3:2     3:3     3:4     3:5     3:6     3:7     3:8     3:9     3:10    3:11    3:12    3:13    3:14    3:15    3:16    3:17    3:18    3:19  그림2  그림1  4:0     4:1     4:2     4:3     4:4     4:5     4:6     4:7     4:8     4:9     4:10    4:11    4:12    4:13    4:14    4:15    4:16    4:17    4:18    4:19  \*\*\*/ |

그림1 

그림2 

그림3 

위의 예에서 스레드는 ManualResetEvent의 WaitOne()을 호출해서 대기상태에 있게 된다. 만약 시그널을 받았다면 WaitOne()은 true를 반환한 후 작업을 진행하게 된다.  
  
**▒ ManualReset의 시그널 대기**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1 | if(ManualRE.WaitOne()){ |

하나의 스레드가 Set()에 의해서 작동하고 있다.   
  
**▒ ManualReset의 시그널 설정하기**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

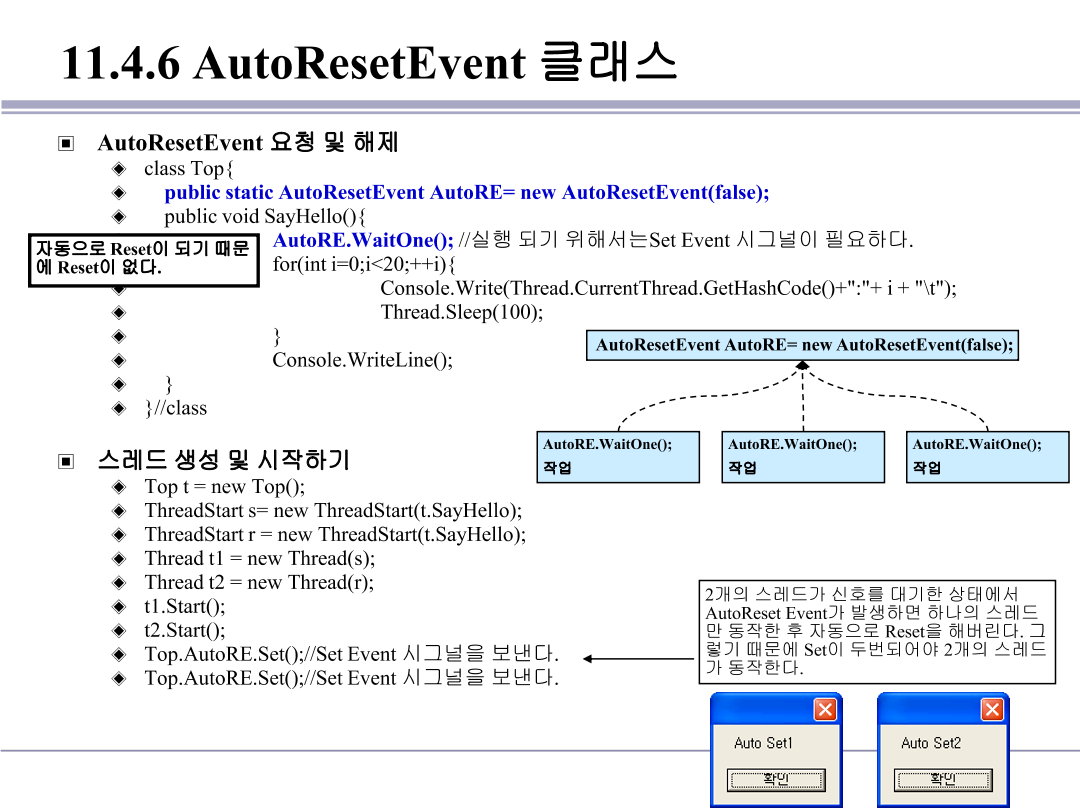
|  |  |
| --- | --- |
| 1  2  3 | t1.Start();  MessageBox.Show("Manual Set");  Top.ManualRE.Set(); |

만약 이 상태에서 다른 스레드를 시작한다면 바로 실행이 될 것이다. 그것은 한번 Set()을 했다면 지속적으로 세팅되어 있기 때문이다. 그래서 바로 Reset()을 호출해서 세팅된 것을 해제해주고 있다.  
  
다음은 Reset()을 호출했기 때문에 t2 스레드는 시그널이 세팅될 때까지 대기상태가 되는 것을 보여주고 있다.  
  
**▒ ManualReset의 시그널 리셋하기**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3 | MessageBox.Show("Manual ReSet");  Top.ManualRE.Reset();  t2.Start(); |

대기상태에 있는 스레드 t2는 Set()에 의해서 다시 시그널을 받음으로써 대기상태에서 벗어나 작업을 계속할 수 있는 것을 볼 수 있다.



**11.4.6 AutoResetEvent 클래스**  
  
ManualResetEvent의 경우에는 수동으로 Reset()을 해주지 않으면 계속적으로 시그널이 보내진 상태가 된다. 그래서 Manual이라는 단어가 앞에 붙어있는 것이다.   
  
이것의 반대의 경우는 Set()을 하더라도 자동으로 Reset()이 되는 경우이다. 즉 Set()이 호출되었다 하더라도 여러 개의 스레드가 동작하는 것이 아니라 하나의 스레드가 이 시그널을 받고 자동(Auto)으로 Reset()을 하는 것이다. 이것을 지원하는 클래스가 AutoResetEvent이다.  
  
**□ AutoResetEvent 클래스**  
◇ Set()을 한 후 스레드가 실행되고 그리고 작업을 완료하면 자동으로 Reset()이 된다.  
  
다음의 경우는 두개의 스레드를 실행시킨 후 Set()을 2번 실행해서 각각의 스레드를 실행하는 예이다.

**§ chap11\AutoResetEventTest.cs**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | /\*\*  AutoResetEvent 예제  \*\*/  using System;  using System.Threading;  using System.Windows.Forms;  public class AutoResetEventTest{      public static void Main(){          Top t = new Top();          ThreadStart s = new ThreadStart(t.SayHello);          ThreadStart r = new ThreadStart(t.SayHello);          Thread t1 = new Thread(s);          Thread t2 = new Thread(r);          t1.Start();          t2.Start();          MessageBox.Show("Auto Set1");          Top.AutoRE.Set();//Set Event 신호를 보낸다.          MessageBox.Show("Auto Set2");          Top.AutoRE.Set();//Set Event 신호를 보낸다.      }  }//class    class Top{      public static AutoResetEvent AutoRE= new AutoResetEvent(false);      public void SayHello(){          AutoRE.WaitOne();  //실행 되기 위해서는Set Event 신호가 필요하다.          for(int i=0; i<20; ++i){              Console.Write(Thread.CurrentThread.GetHashCode() + ":" + i + "\t");              Thread.Sleep(100);          }          Console.WriteLine();      }  }//class  /\*\*\*  그림1  3:0     3:1     3:2     3:3     3:4     3:5     3:6     3:7     3:8     3:9    3:10    3:11    3:12    3:13    3:14    3:15    3:16    3:17    3:18    3:19  그림2  4:0     4:1     4:2     4:3     4:4     4:5     4:6     4:7     4:8     4:9     4:10    4:11    4:12    4:13    4:14    4:15    4:16    4:17    4:18    4:19  \*\*\*/ |

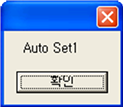
그림1 

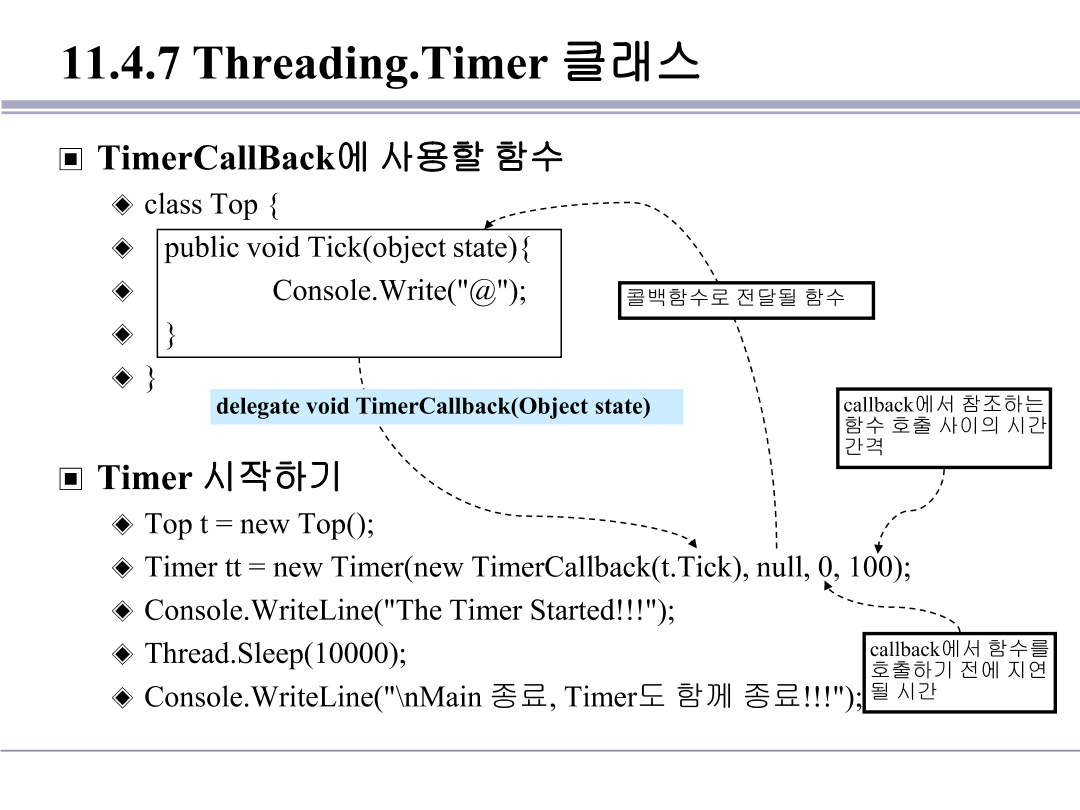
그림2 

위의 예에서도 ManualResetEvent와 마찬가지로 시그널을 받기 위해서 스레드 내에서 WaitOne()을 호출해서 시그널을 지속적으로 대기하게 한다. 그리고 시그널을 보낼 때마다 하나씩 실행되는 것을 확인할 수 있다.   
  
**▒ AutoResetEvent의 시그널 설정하기**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3  4 | MessageBox.Show("Auto Set1");  Top.AutoRE.Set(); //Set Event 시그널을 보낸다.  MessageBox.Show("Auto Set2");  Top.AutoRE.Set(); //Set Event 시그널을 보낸다. |

시그널을 두번 보내는 이유는 하나의 스레드가 시그널을 받았다면 자동으로 Reset을 해버리기 때문에 다른 스레드가 동작하기 위해서는 다시 Set 시그널을 보내 주어야 한다. 즉 이것은 Set 시그널을 보냈다 하더라도 자동으로 Reset이 되는 것을 보여주는 것이다.



**11.4.7 Threading.Timer 클래스**  
  
Timer 클래스는 지정된 간격으로 함수를 실행하는 메커니즘을 제공한다. Timer 클래스의 종류에는 쓰이는 위치에 따라서 다음과 같은 두 종류의 Timer를 만들 수 있다.  
  
**□ System.Threading.Timer**  
◇ 스레드 생성시 사용하는 타이머 클래스  
  
**□ System.Windows.Forms.Timer**  
◇ 윈폼에서 사용하는 타이머 클래스  
  
사용자가 정의한 간격마다 이벤트를 발생시키는 것은 Theading.Timer나 Forms.Timer는 동일하다. 일반적인 응용 프로그램에서는 Threading.Timer를 사용하며, Windows Forms 응용 프로그램에서는 Forms.Timer를 사용한다.  
  
이 절에서는 Threading.Timer 클래스에 대해서 알아보자. Timer 클래스를 사용하려면 먼저 TimerCallback 델리게이트를 선언하고 사용하려는 함수를 등록한 후, 그 델리게이트를 매개변수로 하는 Timer 클래스를 생성하면 된다. 이것을 순서대로 나타내보면 다음과 같다.  
  
**▒ Timer로 사용할 객체 생성**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2 | Top t = new Top()  //타이머에 사용할 함수는 Tick(object state) 함수라고 가정 |

**▒ TimerCallback 델리게이트 생성**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1 | TimerCallback tc = new TimerCallback(t.Tick); |

**▒ Timer 객체 생성**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1 | Timer tt = new Timer(tc, null, 0, 100); |

Timer는 하나의 스레드처럼 주기적으로 실행되며, Timer 델리게이트는 타이머가 생성될 때 지정할 수 있으며 변경할 수는 없다. 그럼 타이머를 사용한 예제를 살펴보자.

**§ chap11\ThreadingTimerTest.cs**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | /\*\*  System.Threading.Timer를 사용한 예제  \*\*/  using System;  using System.Threading;  public class ThreadingTimerTest{      public static void Main(){          Top t = new Top();          Timer tt = new Timer(new TimerCallback(t.Tick), null, 0, 100);          Console.WriteLine("The Timer Started!!!");          Thread.Sleep(10000);          Console.WriteLine("\nMain 종료, Timer도 함께 종료!!!");      }  }    class Top{      public void Tick(object state){          Console.Write("@");      }  }  /\*\*\*  The Timer Started!!!  @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  Main 종료, Timer도 함께 종료!!!  \*\*\*/ |

생성자에 포함된 타이머를 살펴보면, 타이머가 생성된 0초 후에 타이머의 함수를 호출하고 이어서 100ms 간격으로 타이머를 호출하고 있다.   
  
**▒ Timer 객체 생성**

[?](http://www.jabook.com/jabook2/bs/bsTreeLoad.do?ba_no=73)

|  |  |
| --- | --- |
| 1 | Timer tt = new Timer(new TimerCallback(this.Tick), null, 0, 100); |

**□ public Timer(TimerCallback callback, object state, int dueTime, int period);**  
◇ callback : TimerCallback 대리자  
◇ state : callback 매개변수에서 호출한 함수 또는 null 참조와 관련된 응용 프로그램 관련 정보  
◇ dueTime : callback에서 함수를 호출하기 전에 지연될 시간(단위: 1/1000초), 타이머를 즉시 시작하려면: 0   
◇ period : callback에서 참조하는 함수 호출 사이의 시간 간격(단위: 1/1000초)  
  
호출하는 Tick 함수에서는 단지 @를 출력하게 되어 있다. 따라서 100ms에 한번씩 @를 찍어주다가 Main() 함수가 종료하는 10초 후 즉 10000ms 후에 타이머도 함께 종료하게 된다.