The following code demonstrates the use of .NET asynchronous programming using a simple class that factorizes some numbers.

C#

```
using System;
using System.Threading;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Messaging;

// Create an asynchronous delegate.
public delegate bool FactorizingAsyncDelegate (
        int factorizableNum,
        ref int primefactor1,
        ref int primefactor2);

// Create a class that factorizers the number.
public class PrimeFactorizer
{
  public bool Factorize(
              int factorizableNum,
              ref int primefactor1,
              ref int primefactor2)
  {
    primefactor1 = 1;
    primefactor2 = factorizableNum;

    // Factorize using a low-tech approach.
    for (int i=2;i<factorizableNum;i++)
    {
      if (0 == (factorizableNum % i))
      {
        primefactor1 = i;
        primefactor2 = factorizableNum / i;
        break;
      }
    }
```

```
      if (1 == primefactor1 )
        return false;
      else
        return true   ;
    }
}

// Class that receives a callback when the results are available.
public class ProcessFactorizedNumber
{
    private int _ulNumber;

    public ProcessFactorizedNumber(int number)
    {
      _ulNumber = number;
    }

    // Note that the qualifier is one-way.
    [OneWayAttribute()]
    public void FactorizedResults(IAsyncResult ar)
    {
      int factor1=0, factor2=0;

      // Extract the delegate from the AsyncResult.
      FactorizingAsyncDelegate fd = (FactorizingAsyncDelegate)
((AsyncResult)ar).AsyncDelegate;

      // Obtain the result.
      fd.EndInvoke(ref factor1, ref factor2, ar);

      // Output the results.
      Console.WriteLine("On CallBack: Factors of {0} : {1} {2}",
                  _ulNumber, factor1, factor2);
    }
```

```
}

// Class that shows variations of using Asynchronous
public class Simple
{
  // The following demonstrates the Asynchronous Pattern using a callback.
  public void FactorizeNumber1()
  {
    // The following is the client code.
    PrimeFactorizer pf = new PrimeFactorizer();
    FactorizingAsyncDelegate fd = new FactorizingAsyncDelegate (pf.Factorize);

    int factorizableNum = 1000589023, temp=0;

    // Create an instance of the class that is going
    // to be called when the call completes.
    ProcessFactorizedNumber fc = new
ProcessFactorizedNumber(factorizableNum);

    // Define the AsyncCallback delegate.
    AsyncCallback cb = new AsyncCallback(fc.FactorizedResults);

    // You can use any object as the state object.
    Object state = new Object();

    // Asynchronously invoke the Factorize method on pf.
    IAsyncResult ar = fd.BeginInvoke(
                    factorizableNum,
                    ref temp,
                    ref temp,
                    cb,
                    state);

    //
    // Do some other useful work.
```

```
    //. . .
  }

  // The following demonstrates the Asynchronous Pattern using a BeginInvoke,
followed by waiting with a time-out.
  public void FactorizeNumber2()
  {
    // The following is the client code.
    PrimeFactorizer pf = new PrimeFactorizer();
    FactorizingAsyncDelegate fd = new FactorizingAsyncDelegate (pf.Factorize);

    int factorizableNum = 1000589023, temp=0;

    // Create an instance of the class that is going
    // to be called when the call completes.
    ProcessFactorizedNumber fc = new
ProcessFactorizedNumber(factorizableNum);

    // Define the AsyncCallback delegate.
    AsyncCallback cb =
    new AsyncCallback(fc.FactorizedResults);

    // You can use any object as the state object.
    Object state = new Object();

    // Asynchronously invoke the Factorize method on pf.
    IAsyncResult ar = fd.BeginInvoke(
                    factorizableNum,
                    ref temp,
                    ref temp,
                    null,
                    null);

    ar.AsyncWaitHandle.WaitOne(10000, false);
```

```
      if (ar.IsCompleted)
      {
        int factor1=0, factor2=0;

        // Obtain the result.
        fd.EndInvoke(ref factor1, ref factor2, ar);

        // Output the results.

        Console.WriteLine("Sequential : Factors of {0} : {1} {2}",
                    factorizableNum, factor1, factor2);

      }
    }


    public static void Main(String[] args)
    {
      Simple simple = new Simple();
      simple.FactorizeNumber1();
      simple.FactorizeNumber2();
    }
}
```