/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* IDisposableAndDestructor: Part 6

\*

\* Core Topics:

\* 1. The Destructor method.

\* 2. SuppressFinalize method of the Garbage Collector.

\* 3. Describe the coordination between the Dispose method, the Destructor

\* method and the Garbage Collector.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

using System;

// This class simulates a simple customer record that might be read

// from a database. The Random class is used to generate the customer

// identification number and a history of the number of purchases

// the customer has made since signing up with the retailer.

//

// NOTE: Set a breakpoint in the destructor and in the public Dispose()

// method of Customer.

internal class Customer : IDisposable

{

// Used to generate the customer's ID number.

private static Random custIdRg;

// Used to generate the # of purchases the customer made.

private static Random purchasesRg;

// Stores the customer's identification number.

private int customerId;

// Stores the total # of purchases the customer has made.

private int numberOfPurchases;

// Flag indicating Dispose() was called.

private bool alreadyDisposed;

// This static constructor will be called once during the lifetime of

// the program. It initialized the two Random Number Generators using

// data from the current system date and time.

static Customer()

{

DateTime dt = DateTime.Now;

custIdRg = new Random (dt.DayOfYear);

purchasesRg = new Random (dt.DayOfYear + dt.Minute);

}

// This instance constructor is called when an object of type Customer

// is created. It will create the customer ID and the number of

// purchases. It also initialized the disposed flag to false.

internal Customer()

{

customerId = custIdRg.Next(10000, 99999);

numberOfPurchases = purchasesRg.Next (0, 99);

alreadyDisposed = false;

}

// Property that returns the Customer ID.

internal int CustomerNumber

{

get { return customerId; }

}

// Property that returns the number of purchases.

internal int PurchaseTotal

{

get { return numberOfPurchases; }

}

// Private cleanup method that is called from either the destructor

// or the Dispose method. The disposed flag is checked to make sure

// cleanup occurs only 1 time. The argument passed in is checked to

// see if any objects referenced by this one needs their Dispose()

// methods called.

private void Cleanup(bool disposeManagedResources)

{

// Check to see if the Dispose(bool) method was called already

// on the current object instance. If so, we don't want to perform

// cleanup operations again.

if (!alreadyDisposed)

{

// Check to see if this object holds references to any managed

// objects that implement a Dispose() method and call them.

if (disposeManagedResources)

{

// Here is where you put code that would perform the cleanup

// of managed objects. Typically, this would entail calling

// the Dispose() methods of these objects that this class has

// references to.

Console.WriteLine

("\n\tDISPOSE - Cleaning up managed resource. <<< \n");

}

// Here is where you put code that would perform the cleanup of

// unmanaged resources.

Console.WriteLine

("\n\tDISPOSE - Cleaning up unmanaged resources. <<< \n");

// Now set the disposed flag to true to indicate that the Dispose

// method was called and cleanup was done.

alreadyDisposed = true;

}

else

{

Console.WriteLine("\n \*\*\* Error! Attempt to dispose customer " +

"record {0} more than once. \*\*\* \n", CustomerNumber);

}

}

// Class destructor.

~Customer()

{

Cleanup(false);

}

// Class Dispose() method. Notice the GS.SuppressFinalize call.

// This method is declared in the IDisposable interface, and

// implemented here.

public void Dispose ()

{

Cleanup(true);

GC.SuppressFinalize(this);

}

}

// This class tests the Customer class and the Dispose design pattern.

class DisposeTest

{

// This method accepts ANY object that implements the IDisposable

// interface. Notice the use of polymorphism to demonstrate this work.

private static void CleanUpMemory (IDisposable id)

{

id.Dispose();

}

static void Main()

{

Customer c1;

Customer c2;

Customer c3;

Customer c4;

IDisposable iDispose;

// Now that customer implements the IDisposable interface,

// we can use the using keyword to make it easier to call

// the Dispose method. THIS IS THE BEST WAY TO FOLLOW THE

// DISPOSE DESIGN PATTERN! Here's what it looks like from

// the client's perspective.

using (c1 = new Customer ())

{

Console.WriteLine

("1) Read customer {0}. Total purchases: {1}.",

c1.CustomerNumber, c1.PurchaseTotal);

}

// In this example, the developer is calling the Dispose() method

// twice. Normally this would be caught because two consecutive

// calls to Dispose() looks odd. But in larger applications, the

// two calls to Dispose() might be in different locations of the

// application.

c2 = new Customer ();

Console.WriteLine ("2) Read customer {0}. Total purchases: {1}.",

c2.CustomerNumber, c2.PurchaseTotal);

iDispose = c2 as IDisposable;

if (c2 != null)

iDispose.Dispose();

if (c2 != null)

c2.Dispose();

c2 = null;

// Using polymorphism, we can pass a Customer object to a method

// that accepts an object which implements IDisposable.

c3 = new Customer();

Console.WriteLine("3) Read customer {0}. Total purchases: {1}.",

c3.CustomerNumber, c3.PurchaseTotal);

CleanUpMemory(c3);

// Here we show that the developer used a Customer object, but

// never called the Dispose() method. As a backup, the destructor

// will be called by the garbage collector.

c4 = new Customer ();

Console.WriteLine ("4) Read customer {0}. Total purchases: {1}.",

c4.CustomerNumber, c4.PurchaseTotal);

// Pause program. After these lines of code, we should see the GC

// calling the destructor in c4 to perform final cleanup.

Console.Write ("\n\nPress <ENTER> to end: ");

Console.ReadLine();

}

}