Equality in .Net

Gregory Adam

07/12/2008

This article describes how equality works in .net

Introduction

How is equality implemented in .Net ? This is a summary of how it works.

Object.Equals()

Object.Equals() tells you in the most general way if two objects are equal. There are two meanings of equivalence: one for value types and one for reference types.

(1) Value types

The default implementation for **value type** classes iterates over the internal fields comparing them for value equality.

Let's define a struct and test the speed.

The following executes in 7.56 sec

You may wonder what the `ExecuteTimed()` is. It measures the execution time of a method. The source is included in Appendix A.

```
using GregoryAdam.Base.ExtensionMethods;
using System;

public struct RectangleStruct
{
	double Width;
	double Height;
}
class test_struct
{

	const int times = 100000000;

	static void Main(string[] args)
	{

		((delegateVoid)TestEqualStruct).ExecuteTimed(true);


	}

	static void TestEqualStruct()
	{
		RectangleStruct x = new RectangleStruct();
		RectangleStruct y = new RectangleStruct();
		bool b;
		for (int i = times; --i != 0; )
			b = x.Equals(y);
		;

	}
}
```

You may wonder whether the speed can be improved. As it happens, yes – we can – by implementing an override implementation of Equals()

Before we do that, there are some rules to adhere to. Let's examine them

- x.Equals(x) must be true
- x.Equals(y) must yield the same result as y.Equals(x)
- if x.Equals(y) and y.Equals(z) is true, then x.Equals(z) must be true
- x.Equals(null) is false for all x that are not null
- Equals() must not throw any exceptions

Let's try. Overriding the Equals() results in 4.98 secs – which is a third faster.

```
using GregoryAdam.Base.ExtensionMethods;
using System;

public struct RectangleStruct
{
	double Width;
	double Height;

	// IEquatable
	public override bool Equals(object obj)
	{
		RectangleStruct other;

		if (obj != null && obj is RectangleStruct)
		{
			other = (RectangleStruct)obj;
			return (Width == other.Width) && (Height == 
other.Height);
		}
		return false;
	}


} // the class Test is the same
```

You may have noticed that the value type has to be boxed to an object, i.e. the argument of Equals is boxed – to cast it to an object – prior to each call to Equals(). Can we do any faster ? Yes...

The object.Equals() implements the IEquatable interface. In addition to that, we can implement the IEquatable<T> interface. This sounds like a mouthful, but in essence, it's just adding an overloaded method of Equals() which accepts another `RectangleStruct`. If we do, the compiler will call that overloaded method which, as you may have guessed, does not need boxing.

The following code executes in 2.44 seconds.

`Equals(RectangleStruct other) is called instead of Equals(object obj)`

So, if the argument is a `RectangleStruct`, the execution time is reduced to slightly over 30% of the original.

`IEquatable< RectangleStruct> on the` first line indicates that the struct implements the IEquatable<T> interface.

```
using GregoryAdam.Base.ExtensionMethods;
using System;

public struct RectangleStruct : IEquatable<RectangleStruct>
{
	double Width;
	double Height;

	// IEquatable
	public override bool Equals(object obj)
	{
		RectangleStruct other;

		if (obj != null && obj is RectangleStruct)
		{
			other = (RectangleStruct)obj;
			return (Width == other.Width) && (Height ==
other.Height);
		}
		return false;
	}



	// IEquatable<T>
	public bool Equals(RectangleStruct other)
	{
		return (Width == other.Width) && (Height == other.Height);
	}

	public override int GetHashCode()
	{
		return Width.GetHashCode() ^ Height.GetHashCode();
	}

} // the class Test is the same
```

One final note

If you override Equals() then it is best to override GetHasCode() too. GetHasCode () is called when objects are stored in a hash table (Dictionary, HashSet, ...). The value returned by GetHashCode() will be used as the key. The 'rules' for GetHashCode() are as follows:

- If x.Equals(y), then x.GetHashCode() and y.GetHashCode() must return the same result
- HashCodes do not need to be unique.
- Never throw an exception from within GetHashCode()

So, for completeness' sake, I have added GetHashCode() in the code above.

(2) Reference types

Let's examine the reference types. The default behaviour is identity equivalence. It just means that their references must be equal, i.e. var1 and var2 must point to the same object.

Sometimes, it feels more natural to return a value type result. Suppose we had implemented the Rectangle as a reference type. Would it not make more sense for the Equals() method to return whether their values are all equal ? If you do override Equals() you can still test whether two variables point to the same object by using the static method ReferenceEquals() of the object class.

Since the default of object.Equals() is a reference comparison and we want value type behaviour, let's move to our own implementation which executes in 2.96 sec.

```
using GregoryAdam.Base.ExtensionMethods;
using System;

public class RectangleReference
{
	double Width;
	double Height;

	// constructors
	public RectangleReference()
	{
		Width = Height = 0.0;
	}

	public RectangleReference(double width, double height)
	{
		Width = width;
		Height = height;
	}

	// IEquatable
	public override bool Equals(object obj)
	{
		// this cannot be null
		if( obj == null )
			return false;

		// same class ?
		RectangleReference other;

		if( (Object) (other = obj as RectangleReference) != null )
			return (Width == other.Width) && (Height ==
other.Height);

		return false; // obj is another type
	}
}

class test_class
{

	const int times = 100000000;

	static void Main(string[] args)
	{
		((delegateVoid)TestEqualReference).ExecuteTimed(true);


	}
	static void TestEqualReference()
	{
		RectangleReference x = new RectangleReference();
		RectangleReference y = new RectangleReference();


		bool b;
		for (int i = times; --i != 0; )
```

```
                  b = x.Equals(y);
            ;

      }
}
```

Now, like in the value type, let's add the IEquatable<T>

The code below executes in 2.65 sec (compared to 3.10 sec) which is only a bit faster

```
using GregoryAdam.Base.ExtensionMethods;
using System;

public class RectangleReference : IEquatable<RectangleReference>
{
      double Width;
      double Height;

      // constructors
      public RectangleReference()
      {
            Width = Height = 0.0;
      }
      public RectangleReference(double width, double height)
      {
            Width = width;
            Height = height;
      }
      // IEquatable
      public override bool Equals(object obj)
      {
            // this cannot be null
            if( obj == null )
                  return false;

            // same class ?
            RectangleReference other;

            if( (Object) (other = obj as RectangleReference) != null )
                  return (Width == other.Width) && (Height ==
other.Height);

            return false; // obj is another type
      }

      // IEquatable<T>
      public bool Equals(RectangleReference other)
      {
            // this cannot be null

            if ((Object)other == null)
                  return false;

            return (Width == other.Width) && (Height == other.Height);
      }
      public override int GetHashCode()
      {
            return Width.GetHashCode() ^ Height.GetHashCode();
      }
}
// the class Test is the same
```

Implementing the == and != operators

Adding the == and != operators is just a small step. We can use object.Equals()

(1) Value type

```
using GregoryAdam.Base.ExtensionMethods;
using System;

public struct RectangleStruct : IEquatable<RectangleStruct>
{
	double Width;
	double Height;

	// IEquatable
	public override bool Equals(object obj)
	{
		RectangleStruct other;

		if (obj != null && obj is RectangleStruct)
		{
			other = (RectangleStruct)obj;
			return (Width == other.Width) && (Height == 
other.Height);
		}
		return false;
	}



	// IEquatable<T>
	public bool Equals(RectangleStruct other)
	{
		return (Width == other.Width) && (Height == other.Height);
	}

	public override int GetHashCode()
	{
		return Width.GetHashCode() ^ Height.GetHashCode();
	}

	static public bool operator ==(RectangleStruct c1, RectangleStruct 
c2)
	{
		return c1.Equals(c2);
	}
	static public bool operator !=(RectangleStruct c1, RectangleStruct 
c2)
	{
		return !(c1 == c2);
	}
```

(2) Reference type

```
using GregoryAdam.Base.ExtensionMethods;
using System;

public class RectangleReference : IEquatable<RectangleReference>
{
      double Width;
      double Height;

      // constructors
      public RectangleReference()
      {
            Width = Height = 0.0;
      }

      public RectangleReference(double width, double height)
      {
            Width = width;
            Height = height;
      }

      // IEquatable
      public override bool Equals(object obj)
      {
            // this cannot be null
            if( obj == null )
                  return false;

            // same class ?
            RectangleReference other;

            if( (Object) (other = obj as RectangleReference) != null )
                  return (Width == other.Width) && (Height ==
other.Height);

            return false; // obj is another type
      }

      // IEquatable<T>
      public bool Equals(RectangleReference other)
      {
            // this cannot be null

            if ((Object)other == null)
                  return false;

            return (Width == other.Width) && (Height == other.Height);
      }
      public override int GetHashCode()
      {
            return Width.GetHashCode() ^ Height.GetHashCode();
      }
      static public bool operator ==(RectangleReference c1,
RectangleReference c2)
      {
            // c1 can be null here
            if ((Object)c1 == null)
                  return (Object)c2 == null;

            return c1.Equals(c2);
      }
      static public bool operator !=(RectangleReference c1,
RectangleReference c2)
      {
            return !(c1 == c2);
      }
}
```

Summary

- Object.Equals() does
  - A value comparison on value types, i.e. each field must be equal
  - A reference comparison for reference types, i.e. the referenced objects must be the same.
    - Comparing with null
      - Two null values are considered to be equal
      - A null value compared to a non null value is not equal
      - In short: null is only equal to null
- Overriding Object.Equals()
  - For value types
    - There is a significant performance gain
    - Overloading the Equals() method in case the other object is of the same type even increases performance gain
      - Add : IEquatable<ClassName> to the definition of the class
  - For reference types
    - If we want to alter the default reference comparison and implement a value type comparison, this is the method we want to override
    - Overloading the Equals() method in case the other object is of the same type gives only a small performance gain – but I would still do it
      - Add : IEquatable<ClassName> to the definition of the class
  - If you override object.Equals() then override GetHashCode() also
- Use object.Equals() if you implement the operators == and !=

Appendix A

```
using System;
using System.Collections.Generic;

// delegates
namespace GregoryAdam.Base.ExtensionMethods
{
	public delegate void delegateVoid();

	public static partial class ExtensionMethods
	{

		#region ExecuteTimed


		private static Stack<DateTime> Times = new Stack<DateTime>();
		private static Stack<string> Subject = new Stack<string>();
		//-------------------------------------------------------------
		public static void ExecuteTimed(this delegateVoid function)
		{
			ExecuteTimed(function, false);
		}

		public static void ExecuteTimed(this delegateVoid function, 
bool pause)
		{
			Start(function.GetInvocationList()[0].Method.Name + 
"()");
			function();
			End(pause);
		}

		//-------------------------------------------------------------
		private static void Start(string subject)
		{
			lock (Times)
			{
				Subject.Push(subject);
				Times.Push(DateTime.Now);
			}
		}
		//-------------------------------------------------------------
		private static void End()
		{
			End(false);
		}
		//-------------------------------------------------------------
		private static void End(bool pause)
		{
			TimeSpan elapsed;
			string subject;

			lock (Times)
			{
				if (Times.Count == 0)
				{
					System.Diagnostics.Debug.Assert(false, "Time 
Stack empty");
```

```
                    return;
                }

                elapsed = DateTime.Now - Times.Pop();
                subject = Subject.Pop();
            }

            Console.WriteLine("{0} : {1:00}:{2:00}:{3:00}.{4:00}",
                        subject,
                        elapsed.Hours,
                        elapsed.Minutes,
                        elapsed.Seconds,
                        elapsed.Milliseconds / 10
                        );

            if (pause)
            {
                Console.WriteLine("Enter to continue");
                Console.ReadLine();
            }
        }
        //--------------------------------------------------------------
        #endregion

    }
}
```

Appendix B

References

(1) Accelerated C# 2008 – Apress – by Trey Nash
(2) Msdn online