An article clarifying the various ways of comparing two values for equality in .NET

Introduction

The various ways of comparing two values for equality in .NET can be very confusing. In fact if we have two objects a and b in C# there are at least four ways to compare their identity, plus one operator that looks like an identity comparison to add to the confusion:

1. if (a.Equals(b)) {}
2. if (object.Equals(a, b)) {}
3. if (object.ReferenceEquals(a, b) {}
4. if (a == b) {}
5. if (a is b) {}

As if that isn't confusing enough, these methods and operators behave differently depending on:

* whether a and b are reference types or value types
* whether they are reference types which are made to behave like value types for these purposes (System.String is one of these)

This article is an attempt to clarify why we have all these versions of equality, and what they all mean.

What does it mean to be the same?

Firstly, we have to understand that there are actually two basic types of equality for objects:

1. Identity (reference equality): Two objects are identical if they actually are the same object in memory. That is, references to them point to the same memory address.
2. Equivalence (value equality): Two objects are equivalent if the value or values they contain are the same.

So if we have two integers, a and b, both set to value 3, they are equivalent (they have the same value) but not necessarily identical (a and b can refer to different memory addresses).

However if two objects are identical (the same object) then they must be equivalent (have the same underlying values).

What type of Equality do we expect?

Clearly these notions of identity and equivalence are related to the concept of reference types and value types.

**Value types** are intended as lightweight objects that have value semantics: two objects are the same if they have the same value, and then can be used interchangeably. So integers a and b are the same in the example above because their values are both 3, it doesn't matter if references a and b actually refer to the same underlying object in memory.

We don't in general expect **reference types** to behave this way. Suppose we have two separate objects of type Book (a class). Book has one member variable called 'title' (a string). Do we necessarily consider these the 'same' Book if they have the same title? We might do so, but it isn't clear.

To clarify the situation we might add an additional field 'BookId' which is unique for a given actual book. We could then say that two books are the same if they have the same BookId, even if they have different titles. But then we wouldn't normally expect to have two separate Books with the same BookId in memory at the same time: there's only one underlying book. So potentially we can just compare memory addresses to see if two Books are the same.

The point is that equality for reference types is trickier to define. Our default definition is going to be that two reference types are the same if they are identical.

Types of Equality

Now I'll go through each of the types of equality referred to in the first paragraph in turn and try to explain why they exist. I'll also explain how they are implemented for value and reference types, and when you should override or overload them.

1. a.Equals(b)
   * **Overview**

Equals() is a virtual method on System.Object. This means every single object can call this, and in your own type definitions you can override it to give the behaviour you want.

The base System.Object implementation of Equals() is to do an identity comparison. However, Equals() is intended to test for identity or equivalence as appropriate (see the discussion in the paragraph above).

* + **Value Types**

For value types this method is overridden to do a value (equivalence) comparison. In particular, System.ValueType itself, the root of all value types, contains an override that will compare two objects by reflecting over their internal fields to see if they are all equal. If you inherit this (by setting up a struct) your struct will get this override by default.

* + **Reference Types**

For reference types, as discussed above, the situation is trickier. In general we expect Equals() for reference types to do an identity comparison (to check whether the objects actually are the same in memory).

However, certain reference types aren't lightweight enough to work as value types, but nevertheless have value semantics. The canonical example is System.String. System.String is a reference type. However if we have a = "abc" and b = "abc" we expect a to be equal to b. So in the framework Equals() is overridden to do a value comparison.

* + **Override or not?**

As mentioned above, for **value types** there is a default override of a.Equals(b) in the base class System.ValueType which will work for any structs you set up. This method uses reflection to iterate over all of the fields of the two value types you are trying to compare, checking that their values are equal. In general this is what you want for value type comparison.

However, the overridden Equals() method uses reflection, which is slow, and involves a certain amount of boxing. For speed optimization it can be good to override this method. For a more detailed discussion of this see Jeffrey Richter's book 'Applied Microsoft .NET Framework Programming'.

In general it is considered good practice to leave Equals() doing its default identity comparison when defining new **reference types** (classes). The exception is when you know you want value semantics for your class (like System.String), or when you want Equals to work in a specific way. In particular, if your class is going to be used as a key in a Hashtable you need to override Equalsif that is to be in any way efficient.

Note that if you override a.Equals(b) you should also override GetHashCode() and should consider overriding IComparable.CompareTo().

1. object.Equals(a, b)
   * **Overview**

object.Equals(a, b) is a static method on the object class. Jeffery Richter describes it as 'a little helper method'. It's easiest to think of it as a method that does some checking for nulls and then calls a.Equals(b).

The reason it exists is that if a is null a call to a.Equals(b) will throw a NullReferenceException. If there's a possibility that a will be null it is easier to call object.Equals(a, b) than explicitly check for the null. If a can't be null there's no need for the additional check and a call to a.Equals(b) will be better.

* + **Detail**

In detail, this method does the following for a call to object.Equals(a, b):

* + 1. Check if a and b are identical (i.e. they refer to the same location in memory or are both null). If so return true.
    2. Check if either of a and b is null. We know they are not both null otherwise the routine would have returned in 1) above, so if either is null return false.
    3. Both a and b are not null: return the value of a.Equals(b).
  + **Value Types and Reference Types**

Since a and b can't be null for value types, object.Equals(a, b) is identical to a.Equals(b). In general you should call a.Equals(b) in preference to object.Equals(a, b) for value types.

For reference types, as discussed above, you should call this method if there's a chance that a will be null in a call to a.Equals(b).

* + **Override or not?**

object.Equals(a, b) is a static method on System.Object, and consequently can't be overridden. However, since it calls into a.Equals(b) any overrides of Equals will affect calls to this method as well.

1. object.ReferenceEquals(a, b)
   * **Overview**

Whilst the two incarnations of Equals() above check for identity or equivalence depending on the underlying type, ReferenceEquals is intended to always check for identity.

* + **Value Types and Reference Types**

For **reference types** object.ReferenceEquals(a, b) returns true if and only if a and b have the same underlying memory address.

In general we shouldn't care whether **value types** occupy the same underlying memory address. It isn't relevant for anything we'd want to normally use them for. But the definition above gives us a problem when we come to value types being compared with ReferenceEquals.

The difficulty comes from the fact that ReferenceEquals expects two System.Objects as parameters. This means that our value types will get boxed onto the heap as they are passed in to this routine. Normally, because of the way the boxing process works, they will get boxed separately to different memory addresses on the heap. This of course means the call to ReferenceEqualsreturns false.

So for example object.ReferenceEquals(10, 10) returns false, for these reasons.

You can see it's the boxing that causes the problem in the following code:

Hide   Copy Code

*// Set up value type in int variable - no boxing*

int value = 10;

object one = value; *// Cast to object so boxed*

object two = value; *// Cast again so boxed again separately*

*// one and two are now separate memory locations on the heap*

Console.WriteLine(object.ReferenceEquals(one, two)); *// false*

*// Set up value type in object variable which*

*// immediately boxes it onto the heap*

object value2 = 10; *// value is boxed already*

object three = value2; *// three points to the boxed value*

object four = value2; *// four also points to the same boxed value*

Console.WriteLine(object.ReferenceEquals(three, four)); *// true*

* + **Override or not?**

ReferenceEquals is a static method on object, and so once again cannot be overridden. It will always perform identity checks as outlined above.

1. a == b
   * **Overview**

== is an operator, clearly, and not a method. In my humble opinion it has been included in C# largely as a syntactic convenience and to make the language look like C/C++.

As with a.Equals(b), == is intended to test for identity or equivalence as appropriate (see the discussion in the paragraph "What type of Equality do we expect?" above. In fact, in almost all circumstances == should behave like a.Equals(b).

* + **Value Types**

For value types within the .NET Framework, == is implemented as you would expect, and will test for equivalence (value equality). However, for any custom value types you implement (structs) a default == will not be available unless you provide one.

* + **Reference Types**

For reference types a default == is available, and this will test for identity (reference equality). For most reference types in the .NET Framework == will again test for identity, but, as for a.Equals(b), there are certain classes where the operator has been overloaded to do a value comparison. System.String is once again the canonical example, for the reasons discussed in part 1 of this article.

* + **Override (overload?) or not?**

Since == is an operator we can't override it. However, we can overload it to provide a different functionality to the base functionality described above.

For **reference types** Microsoft recommends that you don't overload == unless you have reference types behaving as value types as discussed above. This means that even if you override a.Equals(b) to provide some custom functionality you should leave your == operator to provide an identity test. This is, I think, the only occasion where == should behave differently from a.Equals(b).

For **value types**, as mentioned above, a default overload of == will not be available and you will have to provide one if you need one. The easiest thing to do is simply to call a.Equals(b) from an operator overload in your struct: in general your implementation of == should not be different from a.Equals(b).

Note that if you overload == you should overload !=. You should also override a.Equals(b) to do the same thing, and as a result should overload GetHashCode. Finally you should consider overriding IComparable.CompareTo().

* + **Care with == and Reference Types**

One final thing to note is that operator overloads don't behave like overrides. If you use the ==operator with reference types without thinking, this can be a problem.

For example, suppose you have an untyped DataSet ds containing a DataTable dt. Suppose this has columns Id and Name. dt has two rows. Consider the following code:

Hide   Copy Code

*// Create DataSet*

DataSet ds= new DataSet("ds");

DataTable dt= ds.Tables.Add("dt");

dt.Columns.Add("Value", typeof(int));

*// Add two rows, both with Value column set to 1*

DataRow row1= dt.NewRow();row1["Value"] = 1;dt.Rows.Add(row1);

DataRow row2= dt.NewRow();row2["Value"] = 1;dt.Rows.Add(row2);

Console.WriteLine(row1["Value"] == row2["Value"]);

*// Compare with == returns false.*

Console.WriteLine(row1["Value"].Equals(row2["Value"]));

*// Compare with .Equals returns true.*

When we compare with == in the example above we get false, even though the column in both rows contains the integer 1. The reason is that both row1[Value] and row2[Value] return objects, not integers. So == will use the == in System.Object, not any overloaded version in integer. The == in System.Object does an identity comparison (reference equality test). The underlying values have been separately boxed onto the heap, so aren't in the same memory address, and the test fails.

When we compare with .Equals we get true. This is because .Equals is overridden in System.Int32 to do a value comparison, so the comparison uses the overridden version to correctly compare the values of the two integers.

1. a is b
   * **Overview**

a is b isn't actually a test for object equality at all, although it looks like one. b here has to be a type name (so b would need to be a class name, for example). The operator tests whether object a is either of type b or can be cast to it without an exception being thrown. This is equivalent to TypeOf a Is b in VB.NET, which is a little clearer.

* + **Value Types/Reference Types**

The operator works in the same way for both value types and reference types.

* + **Override (overload?) or not?**

The operator cannot be overloaded (or overridden clearly).

The Final Twist: String Interning

On the basis of the above what should this do?

Hide   Copy Code

object a = "Hello World";

object b = "Hello World";

Console.WriteLine(a.Equals(b));

Console.WriteLine(a == b);

At first glance you might say that:

1. a and b are reference types containing strings (you would be right).
2. .Equals is overridden in the string class to do an equivalence (value) comparison, and the values are equal. So a.Equals(b) is true (you would still be right).
3. However, a == b is an overload and on the object type it does an identity comparison, not a value comparison (you would still be right).
4. a and b are separate objects in memory so a == b is false (you would be wrong)

4. is actually wrong, but only because of an optimization in the CLR. The CLR keeps a list of all strings currently being used in an application in something called the intern pool. When a new string is set up in code the CLR checks the intern pool to see if the string is already in use. If so, it will not allocate memory to the string again, but will re-use the existing memory. Hence a == b is true above.

You can prevent strings being interned by using a StringBuilder as below. In this case a.Equals(b) will be true, and a== b will be false, which is what you would expect:

Hide   Copy Code

object a = "Hello World";

object b = new StringBuilder().Append("Hello").Append(" World").ToString();

Console.WriteLine(a.Equals(b));

Console.WriteLine(a == b);

VB.NET

This article has talked mainly about C#. However, the situation is similarly confusing in VB.NET. Because they are methods on System.Object, VB.NET has methods a.Equals(b), object.Equals(a, b) and object.ReferenceEquals(a, b) which are the same as the methods described above.

VB.NET has no == operator, or any operator equivalent to it.

VB.NET additionally has the Is operator. This operator's use in TypeOf a Is b statements was discussed under a is b: Overview above.

VB.NET: a Is b

The Is operator can also be used for identity (reference equality) comparisons on two reference types in VB.NET. However, unlike a.ReferenceEquals(b), which does the same thing for reference types, the Is operator cannot be used at all with value types. The Visual Basic compiler will not compile code where either of a or b in the statement a Is b are value types.