Story of Equality in .Net – Part 1

# Introduction

The purpose of this post is to outline and explore some of the issues that makes performing the equality much more complex than you might expect. Amongst other things, we will examine the difference between the value and reference equality and why equality and inheritance don’t work well together. So let’s get started.

We will start from a simple example that will compare two numbers. For instance let’s say that 3 is less than 4, conceptually it is trivial and the code for that is also very easy and simple:

if(3 < 4)

        {

        }

If you look at the [**System.Object**](https://msdn.microsoft.com/en-us/library/system.object(v=vs.110).aspx) class from which all other types inherit, you will find the following 4 methods which are for equality checking:

[**static Equals()**](https://msdn.microsoft.com/en-us/library/w4hkze5k(v=vs.110).aspx)

[**virtual Equals()**](https://msdn.microsoft.com/en-us/library/bsc2ak47(v=vs.110).aspx)

[**static ReferenceEquals()**](https://msdn.microsoft.com/en-us/library/system.object.referenceequals(v=vs.110).aspx)

[**virtual GetHashCode()**](https://msdn.microsoft.com/en-us/library/system.object.gethashcode(v=vs.110).aspx)

In addition to that Microsoft has provided 9 different interfaces for performing equality or comparison of types:

[**IEquatable<T>**](https://msdn.microsoft.com/en-us/library/ms131187(v=vs.110).aspx)

[**IComparable**](https://msdn.microsoft.com/en-us/library/system.icomparable(v=vs.110).aspx)

[**IComparable<T>**](https://msdn.microsoft.com/en-us/library/4d7sx9hd(v=vs.110).aspx)

[**IComparer**](https://msdn.microsoft.com/en-us/library/system.collections.icomparer(v=vs.110).aspx)

[**IComparer<T>**](https://msdn.microsoft.com/en-us/library/system.collections.icomparer(v=vs.110).aspx)

[**IEqualityComparer**](https://msdn.microsoft.com/en-us/library/ms132151(v=vs.110).aspx)

[**IEqualityComparer<T>**](https://msdn.microsoft.com/en-us/library/ms132151(v=vs.110).aspx)

[**IStructuralEquatable**](https://msdn.microsoft.com/en-us/library/system.collections.istructuralequatable(v=vs.110).aspx)

[**IStructuralComparable**](https://msdn.microsoft.com/en-us/library/system.collections.istructuralcomparable(v=vs.110).aspx)

Most of these method and interfaces come with a risk that if you override their implementation incorrectly, it will cause bugs in your code and can will also break the existing collections provided by the framework which depend on them

We will see what’s the purpose of these method and interfaces is and how to use these them correctly. We will also focus on How to provide custom implementation for equality and comparisons in the right way, which will perform efficiently and follows best practices and most importantly does not break other types implementation.

# Equality is Complex/Difficult

There are 4 reasons that make equality complex than you might expect:

1. Reference v/s Value Equality
2. Multiple ways to Compare Values
3. Accuracy
4. Conflict with OOP

# Reference V/S Value Equality

There is an issue of reference versus value equality, it’s possible to treat equality either way and unfortunately c# is not being designed in a way so that it can distinguish between two of these and that can cause unexpected behavior sometimes, if you don’t understand how these various operators and method work.

As you know, in C# reference types do not contain actual value, they contains a pointer to a location in memory that actually holds those values, which means that for reference types there are two possible ways to measure the equality.

You can say that do both variables refer to same location in memory which is called reference equality and known as Identity or you can say that do the location to which both variables are pointing contains the same value, even if they are different locations which is called Value Equality.

We can illustrate the above points using the following example:

class Program

{

    static void Main(String[] args)

    {

        Person p1 = new Person();

        p1.Name = "Ehsan Sajjad";

        Person p2 = new Person();

        p2.Name = "Ehsan Sajjad";

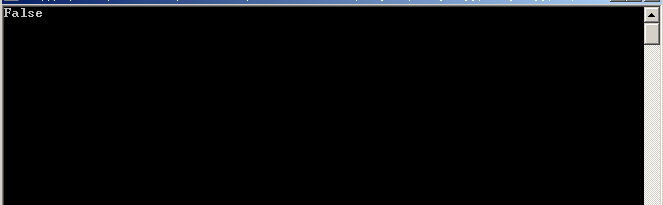
        Console.WriteLine(p1 == p2);

        Console.ReadKey();

    }

}

As you can see in the above example, we have instantiated two objects of Person class and both contains the same value for Name property, clearly the above two instances of Person class are identical as they contain the same values, are they really equal? When we check their equality of both instances using c# equality operator and runt the example code, it prints out on the console **False** as output, which means that they are not equal.



It is because for Person class both C# and the .Net framework considers the equality to be the Reference Equality. In other words, the Equality operator checks whether these two variable refer to the same location in memory, so in this example, they are not equal because though both instances of Person class are identical but they are separate instances, the variables **p1** and **p2** both refer to different locations in memory.

Reference Equality is very quick to perform, because you only need to check for one thing whether the two variable holds the same memory address, while comparing values can be a lot slower.

For Example, if Person class holds a lot of fields and properties instead of just one, and if you wanted to check if the two instances of Person class have same values, you will have to check every field/property, there is no operator in C# which would check the value equality of two Person class instances which is reasonable though, because comparing two instances of Person class containing exactly the same values is not the sort of thing you would normally want to do, obviously if for some reason you would want to do that you will need to write your own code to do that.

Now take this code as example:

class Program

{

    static void Main(String[] args)

    {

        string s1 = "Ehsan Sajjad";

        string s2 = string.Copy(s1);

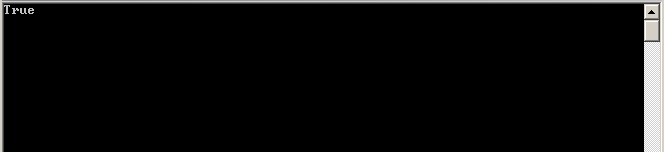
        Console.WriteLine(s1 == s2);

        Console.ReadKey();

     }

 }

The above code is quite similar to previous example code, but in this case we are applying equality operator on to identical strings, we instantiated a string and stored it’s reference in a variable named **s1**, then we created copy of its value and hold that in another variable **s2**, now if we run this code, we will see that according to output we can say that both strings are equal.



If the equality operator had been checking for reference equality, we would had seen **false** printed on the console for this program, but for strings == operator evaluates equality of values of the operands.

Microsoft has implemented it like that, because checking whether one string contains another string is something a programmer would very often need to do.

# Reference and Value Types

The reference and value issue only exists for Reference Types by the way. For unboxed value types for such as integer, float etc the variable directly contains the value, there are no references which means that equality only means to compare values.

The following code which compares two integers will evaluate that both are equal, as the equality operator will compare the values that are hold by the variables.

class Program

{

    static void Main(String[] args)

    {

        int num1 = 2;

        int num2 = 2;

        Console.WriteLine(num1 == num2);

        Console.ReadKey();

    }

}

So in the above code the equality operator is comparing the value stored in variable **num1** with the value stored in **num2**.

However if we modify this code and cast both variables to object, as we did in the following lines of code:

int num1 = 2;

int num2 = 2;

Console.WriteLine((object)num1 == (object)num2);

Now if we run the code, you will see that the result in contradictory with the result we got from the first version of code, which is the second version of code comparison returns **false**, that happened because the object is a reference type, so when we cast integer to object, it ends up boxed in to object as reference, which means the second code is comparing references not values and it returns false because both integers are boxed in to different reference instances.

This is something that a lot of developers don’t expect, normally we don’t cast value types to object, but there is another common scenario that we often see is that if we need to cast value type in to an interface.

 Console.WriteLine((IComparable<int>)num1 == (IComparable<int>)num2);

For illustrating what we said above, let’s modify the example code to cast the integer variables to **ICompareable<int>**. This is an interface provided by .Net framework which integer type inherits or implements, we will talk about it in some other post about it.

In .Net interfaces are always reference types, so the above line of code involves boxing too, and if we run this code, we will see that this equality check also returns false, and it’s because this is again checking reference equality.

So, you need to be careful when casting values types to interfaces, it will always result in reference equality if you do equality check.

# == operator

All this code would probably not had been a problem, if C# had different operators for value-types and reference types equality, but it does not, which some developers think is a problem. C# has just one equality operator and there is no obvious way to tell upfront what the operator is actually going to do for a given type.

For instance consider this line of code:

Console.WriteLine(var1 == var2)

We cannot tell what the equality operator will do in the above, because you just have to know what equality operator does for a type, there is no way around, that’s how C# is designed.

In this post we will go through, what the equality operator does and how it works under the hood in detail, so after reading the complete post, I hope you will have a much better understanding than other developers that what actually happens when you write an equality check condition and you will be better able to tell how equality between two objects is evaluated and will be able to answer correctly whenever you came across the code where two objects are being compared for equality.

# Different Ways to Compare Values

Another issue that exists in that complexity of equality is, there are often more than one ways to compare values of a given type. String type is the best example for this. Suppose we have two string variable the contain the same value in them:

string s1 = "Equality";

        string s2 = " Equality";

Now if compare both s1 and s2, should we expect that the result would be true for equality check? Means should we consider these two variables to be equal?

I am sure you are looking as both string variables contains exactly same values, then it makes sense to consider them equal, and indeed that is what c# does, but what if I change the case of one of them to make them different like:

string s1 = "EQUALITY";

        string s2 = "equality";

Now should these two strings to be considered equal? In C# the equality operator will evaluate to false saying that the two strings are not equal, but if we are not asking about C# equality operator, but about in Principle we should consider those two strings as equal then we cannot really answer, as it completely depends on the context whether we should consider or ignore the case, Let’s say I have a database of food items, and we are querying a food item to be searched from database, then the changes are we want to ignore the case and treat the both string equal, but if the user is typing in password for logging in to an application, and you have to check if the password entered by user is correct, then you should not certainly consider the lower case and title case strings to be equal.

The equality operator for strings in C# is always case sensitive, so you can’t use it for comparison and ignore the case. If you want to ignore the case, you can do but you will have to call the special methods which are defined in the String type. For Example:

string s1 = "EQUALITY";

        string s2 = "equality";

if(s1.Equals(s2,StringComparison.OrdinalIgnoreCase))

The above example will evaluate if statement to true as we are telling to ignore the case when doing comparison for equality between s1 and s2.

Now I am sure that none of that will surprise you. Case sensitivity is an issue that almost everyone encounters very early on when they do programming. From the above example we can illustrate a wider point for equality in general that Equality is not absolute in programming, it is often context-sensitive (e.g. case-sensitivity of string)

One example of this is that user is searching for an item on a shopping cart web application and user types an item name with extra whitespace in it, but when we are comparing that with items in our database, so should we consider the item in our database equal to the item entered by user with whitespace, normally we consider them equal and display that result to user as a result of searching, which again illustrates that equality is context sensitive.

Let’s take one more example, consider the following two database records:

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Name** | **Price** | **LastUpdated** |
| 3211 | Cold Coffee | 2$ | 1 Jan 2015 |

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Name** | **Price** | **LastUpdated** |
| 3211 | Cold Coffee | 2.5$ | 2 Jan 2016 |

Are the equal? In one sense Yes. Obviously these are the same records, they refer to the same drink item and they have the same primary key, but couple of columns value are different, it is clear that the second records item is the data after the records was updated and the first one is before updating, so this illustrates another conceptual issue with equality which comes in to play when you are updating data. Do you care about the precise values of the record or do you care whether it is the same record and clearly there is no one right answer to that. So once again it depends on the context what you are trying to do!

# Equality and Comparison

The way .Net deals with multiple meanings of equality is quite neat. .Net allows each type to specify its own single natural way of measuring equality for that type. So, for example, String type defines it’s natural equality to be if two strings contains exactly same sequence of characters, that’s why comparing two strings with different case returns false as they contains different character. This is because “**eqaulity”** is not equal to **“EQUALITY”** as lower case and uppercase are different characters.

It is very common that the types expose their natural way of determining equality by means of a generic interface called [**IEquatable<T>**](https://msdn.microsoft.com/en-us/library/ms131187(v=vs.110).aspx). String also implements this interface for equality. But separately .Net also provides a mechanism for you to plug in a different implementation of equality if you don’t like the Type’s own definition or if that does not fulfill your needs.

This mechanism is based on what is known as Equality Comparers. An Equality Comparer is an object whose purpose is to test whether instances of a type are equal using the definition provided by the comparer for checking equality.

Equality Comparers implement an interface called [**IEqualityComparer<T>**](https://msdn.microsoft.com/en-us/library/ms132151(v=vs.110).aspx). So for example, if you want to compare string ignoring the extra whitespaces, you could write an equity comparer that knows how to do that and then use that equality comparer instead of the equality operator as required.

Things work basically the same way for doing ordering comparisons. The main difference is that you would use different interfaces. .Net also provides an interface to provide mechanism for a type to do less than or greater then comparison for a type which is known as[**ICompareable<T>**](https://msdn.microsoft.com/en-us/library/4d7sx9hd(v=vs.110).aspx), and separately you can write what are known as comparers which is [**IComparer<T>**](https://msdn.microsoft.com/en-us/library/system.collections.icomparer(v=vs.110).aspx), this can be used to define an alternative implementation for comparison done for ordering, we will see how to implement these interfaces in some other post.

# Equality for Floating Points

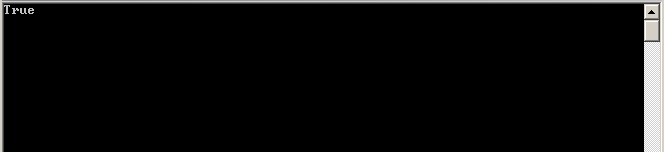
Some data types are inherently approximate. In .Net you will encounter this problem with floating point types like **float**, **double** or **decimal** or any type that contains a floating point type as a member field. Let’s have a look on an example.

float num1 = 2.000000f;

float num2 = 2.000001f;

Console.WriteLine(num1 == num2);

We have two floating point numbers that are nearly equal. So are they equal? It looks pretty obvious that they are not equal as they differ in the final digit and we are printing the equality result on console, so when we run the code, the program displays true



This program has come out saying that they both are equal which is completely contradictory to what we have evaluated by looking at the numbers and you can probably guess what the problem is. Computers can only the numbers to a certain level of accuracy and the float type just cannot store the enough significant digits to distinguish these two particular numbers and it can work other way around two, see this example:

float num1 = 1.05f;

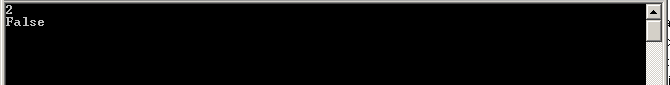
            float num2 = 0.95f;

            var sum = num1 + num2;

            Console.WriteLine(sum);

            Console.WriteLine(sum == 6.0f);

This is a simple calculation where we are adding **1.05** to **0.95**. It looks very obvious that when you add those two numbers you will get the answer **2.0**, so we have written a small program for this which adds those two numbers and then we check that the sum of two numbers is equal to **2.0**, if we run the program, the output contradicts what we had thought, which says the sum is not equal to **2.0,** the reason is that rounding errors happened in the floating point arithmetic resulting in the answer storing a number that is very close to **2**, so close that string representation on **Console.WriteLine** even displayed it as **2** but it’s still not quite equal to **2**.



Those rounding errors in floating point arithmetic has resulted in the program to give the opposite answer to what any common sense reasoning would tell you. Now this is an inherent difficulty with the floating point numbers. Rounding error means that testing for equality often give you the wrong result and .Net has no solution for this. The recommendation is, you don’t try to compare floating point numbers for equality because the results might not be what you predict. This only applies to equality, this problem does not normally affect the less than an greater than comparisons, in most cases there are no problems with comparing the floating points number to see whether one is greater than or less than another , it’s equality that gives the problem.

# Equality Conflict with Object Oriented Principles

This one often comes to as a surprise to experienced developers as well, there is in fact a fundamental conflict between equality comparisons, type safety and good object oriented practices. These 3 things do not sit well together, this often makes very hard to make equality right and bug free even once you resolved the other issues.

We will not talk much about this in details as it will be easy for you to understand once we start seriously coding which I will demonstrate in a separate post and you will be able to then how the problem naturally arises in the code you right.

Now let’s just try and give you a rough idea of the conflict for now. Let’s say we have base class **Animal** which represents different animals and will have a derived class for example **Dog** which adds information specific to the Dog.

    public class Animal

    {

    }

    public class Dog : Animal

    {

    }

If we wanted the Animal class to declare that Animal instances know how to check whether they are equal to other Animal instances, you might attempt to have it implement **IEquatable<Animal>**. This requires it to implement an **Equals()** method which takes an **Animal** instance as a parameter .

public class Animal : IEquatable<Animal>

    {

        public virtual bool Equals(Animal other)

        {

            throw new NotImplementedException();

        }

    }

If we want Dog class to also declare that Dog instances know how to check wether they are equal to other Dog instances, we probably have implement **IEquatable<Dog>** that means it will also implement similar Equals() method which take Dog instance as parameter.

public class Dog : Animal, IEquatable<Dog>

    {

        public virtual bool Equals(Dog other)

        {

            throw new NotImplementedException();

        }

    }

And this is where the problem comes in. You can probably guess that in a well-designed OOP code, you would expect the **Dog** class to override the **Equals()** method of **Animal** class, but the trouble is Dog equals method has a different argument parameter than Animal Equals method which means it won’t override it and if you are not very careful that can cause sort of subtle bugs where you end up calling the wrong equals method and so returning the wrong result.

Often the only work around to this lose type-safety and that’s what you exactly see in the **Object** type Equals method which is the most basic way most types implement equality.

 class Object

{

  public virtual bool Equals(object obj)

{

}

}

This method takes an instance of **object** type as parameter which means it is not type-safe, but it will work correct with inheritance. This is a problem that is not well-known, there were a few blogs around that gave incorrect advice on how to implement equality because they don’t take account of this issue, but it is a problem there. We should be very careful how we design our code to avoid it.

# Summary

* C# does not syntactically distinguish between value and reference equality which means it can sometimes be difficult to predict what the equality operator will do in particular situations.
* There are often multiple different ways of legitimately comparing values. .Net addresses this by allowing types to specify their preferred natural way to compare for equality, also providing a mechanism to write equality comparers that allow you to place a default equality for each type
* It is not recommended to test floating point values for equality because rounding errors can make this unreliable
* There is an inherent conflict between implementing equality, type-safety and good Object Oriented practices.