Item 6: Understand the Relationships Among the Many Different Concepts of Equality

When you create your own types (either classes or structs), you define what equality means for that type. C# provides four different functions that determine whether two different objects are “equal”:

```
public static bool ReferenceEquals
    (object left, object right);
public static bool Equals
    (object left, object right);
public virtual bool Equals(object right);
public static bool operator ==(MyClass left, MyClass right);
```

The language enables you to create your own versions of all four of these methods. But just because you can doesn’t mean that you should. You should never redefine the first two static functions. You’ll often create your own instance Equals() method to define the semantics of your type, and you’ll occasionally override operator==(), typically for performance reasons in value types. Furthermore, there are relationships among these four functions, so when you change one, you can affect the behavior of the others. Yes, needing four functions to test equality is complicated. But don’t worry—you can simplify it.

Of course, those four methods are not the only options for equality. Types that override Equals() should implement IEquatable<T>. Types that implement value semantics should implement the IStructuralEquality interface. That means six different ways to express equality.

Like so many of the complicated elements in C#, this one follows from the fact that C# enables you to create both value types and reference types. Two variables of a reference type are equal if they refer to the same object, referred to as object identity. Two variables of a value type are equal if they are the same type and they contain the same contents. That’s why equality tests need so many different methods.

Let’s start with the two functions you should never change. Object.ReferenceEquals() returns true if two variables refer to the same object—that is, the two variables have the same object identity. Whether the types being compared are reference types or value types, this method always tests object identity, not object contents. Yes, that means that ReferenceEquals() always returns false when you use it to test equality for value types. Even when you compare a value type to itself, ReferenceEquals() returns false. This is due to boxing, which is covered in Item 45.

```
int i = 5;
int j = 5;
if (Object.ReferenceEquals(i, j))
    Console.WriteLine("Never happens.");
else
    Console.WriteLine("Always happens.");

if (Object.ReferenceEquals(i, i))
```

```
    Console.WriteLine("Never happens.");
else
    Console.WriteLine("Always happens.");
```

You'll never redefine Object.ReferenceEquals() because it does exactly what it is supposed to do: tests the object identity of two different variables.

The second function you'll never redefine is static Object.Equals(). This method tests whether two variables are equal when you don't know the runtime type of the two arguments. Remember that System.Object is the ultimate base class for everything in C#. Anytime you compare two variables, they are instances of System.Object. Value types and reference types are instances of System.Object. So how does this method test the equality of two variables, without knowing their type, when equality changes its meaning depending on the type? The answer is simple: This method delegates that responsibility to one of the types in question. The static Object.Equals() method is implemented something like this:

```
public static new bool Equals(object left, object right)
{
    // Check object identity
    if (Object.ReferenceEquals(left, right) )
        return true;
    // both null references handled above
    if (Object.ReferenceEquals(left, null) ||
        Object.ReferenceEquals(right, null))
        return false;
    return left.Equals(right);
}
```

This example code introduces a method I have not discussed yet: namely, the instance Equals() method. I'll explain that in detail, but I'm not ready to end my discussion of the static Equals() just yet. For right now, I want you to understand that static Equals() uses the instance Equals() method of the left argument to determine whether two objects are equal.

As with ReferenceEquals(), you'll never overload or redefine your own version of the static Object.Equals() method because it already does exactly what it needs to do: determines whether two objects are the same when you don't know the runtime type. Because the static Equals() method delegates to the left argument's instance Equals(), it uses the rules for that type.

Now you understand why you never need to redefine the static ReferenceEquals() and static Equals() methods. It's time to discuss the methods you will override. But first, let's briefly discuss the mathematical properties of an equality relation. You need to make sure that your definition and implementation are consistent with other programmers' expectations. This means that you need to keep in mind the mathematical properties of equality: Equality is reflexive, symmetric, and transitive. The reflexive property means that any object is equal to itself. No matter what type is involved, a == a is always true. The symmetric property means that order does not matter: If a == b is true, b == a is also true. If a == b is false, b == a is also false. The last property is that if a == b and b == c are both true, then a == c must also be true. That's the transitive property.

Now it's time to discuss the instance Object.Equals() function, including when and how you override it. You create your own instance version of Equals() when the default behavior is inconsistent with your type. The Object.Equals() method uses object identity to determine whether two variables are equal. The default Object.Equals() function behaves exactly the same as Object.ReferenceEquals(). But wait—value types are different. System.ValueType does override Object.Equals(). Remember that ValueType is the base class for all value types that you create (using the `struct` keyword). Two variables of a value type are equal if they are the same type and they have the same contents. ValueType.Equals() implements that behavior. Unfortunately, ValueType.Equals() does not have an efficient implementation. ValueType.Equals() is the base class for all value types. To provide the correct behavior, it must compare all the member variables in any derived type, without knowing the runtime type of the object. In C#, that means using reflection. As you'll see in Item 43, there are many disadvantages to reflection, especially when performance is a goal. Equality is one of those fundamental constructs that gets called frequently in programs, so performance is a worthy goal. Under almost all circumstances, you can write a much faster override of Equals() for any value type. The recommendation for value types is simple: Always create an override of ValueType.Equals() whenever you create a value type.

You should override the instance Equals() function only when you want to change the defined semantics for a reference type. A number of classes in the .NET Framework Class Library use value semantics instead of reference semantics for equality. Two string objects are equal if they contain the same contents. Two DataRowView objects are equal if they refer to the same DataRow. The point is that if your type should follow value semantics (comparing contents) instead of reference semantics (comparing object identity), you should write your own override of instance Object.Equals().

Now that you know when to write your own override of Object.Equals(), you must understand how you should implement it. The equality relationship for value types has many implications for boxing and is discussed in Item 45. For reference types, your instance method needs to follow predefined behavior to avoid strange surprises for users of your class. Whenever you override Equals(), you'll want to implement IEquatable<T> for that type. I'll explain why a little further into this item. Here is the standard pattern for overriding System.Object.Equals. The highlight shows the changes to implement IEquatable<T>.

```
public class Foo : IEquatable<Foo>
{
    public override bool Equals(object right)
    {
        // check null:
        // this pointer is never null in C# methods.
        if (object.ReferenceEquals(right, null))
            return false;

        if (object.ReferenceEquals(this, right))
            return true;
        // Discussed below.

        if (this.GetType() != right.GetType())
            return false;

        // Compare this type's contents here:
        return this.Equals(right as Foo);
    }

    #region IEquatable<Foo> Members
    public bool Equals(Foo other)
```

```
        {
            // elided.
            return true;
        }
        #endregion
    }
```

First, Equals() should never throw exceptions—it doesn't make much sense. Two variables are or are not equal; there's not much room for other failures. Just return false for all failure conditions, such as null references or the wrong argument types. Now, let's go through this method in detail so you understand why each check is there and why some checks can be left out. The first check determines whether the right-side object is null. There is no check on this reference. In C#, this is never null. The CLR throws an exception before calling any instance method through a null reference. The next check determines whether the two object references are the same, testing object identity. It's a very efficient test, and equal object identity guarantees equal contents.

The next check determines whether the two objects being compared are the same type. The exact form is important. First, notice that it does not assume that this is of type Foo; it calls this.GetType(). The actual type might be a class derived from Foo. Second, the code checks the exact type of objects being compared. It is not enough to ensure that you can convert the right-side parameter to the current type. That test can cause two subtle bugs. Consider the following example involving a small inheritance hierarchy:

```
public class B : IEquatable<B>
{
    public override bool Equals(object right)
    {
        // check null:
        if (object.ReferenceEquals(right, null))
            return false;

        // Check reference equality:
        if (object.ReferenceEquals(this, right))
            return true;

        // Problems here, discussed below.
        B rightAsB = right as B;
        if (rightAsB == null)
            return false;

        return this.Equals(rightAsB);
    }

    #region IEquatable<B> Members

    public bool Equals(B other)
    {
        // elided
        return true;
    }

    #endregion
}
```

```
public class D : B, IEquatable<D>
{
    // etc.
    public override bool Equals(object right)
    {
        // check null:
        if (object.ReferenceEquals(right, null))
            return false;

        if (object.ReferenceEquals(this, right))
            return true;

        // Problems here.
        D rightAsD = right as D;
        if (rightAsD == null)
            return false;

        if (base.Equals(rightAsD) == false)
            return false;

        return this.Equals(rightAsD);
    }

    #region IEquatable<D> Members
    public bool Equals(D other)
    {
        // elided.
        return true; // or false, based on test
    }
    #endregion
}

//Test:
B baseObject = new B();
D derivedObject = new D();

// Comparison 1.
if (baseObject.Equals(derivedObject))
    Console.WriteLine("Equals");
else
    Console.WriteLine("Not Equal");

// Comparison 2.
if (derivedObject.Equals(baseObject))
    Console.WriteLine("Equals");
else
    Console.WriteLine("Not Equal");
```

Under any possible circumstances, you would expect to see either Equals or Not Equal printed twice. Because of some errors, this is not the case with the previous code. The second comparison will never return true. The base object, of type B, can never be converted into a D. However, the first comparison might evaluate to true. The derived object, of type D, can be implicitly converted to a type B. If the B members of the right-side argument match the B members of the left-side argument, B.Equals() considers the objects equal. Even though the two objects are different types, your method has considered them equal. You've broken the symmetric property of Equals. This construct broke because of the automatic conversions that take place up and down the inheritance hierarchy.

When you write this, the D object is explicitly converted to a B:

```
baseObject.Equals(derived)
```

If baseObject.Equals() determines that the fields defined in its type match, the two objects are equal. On the other hand, when you write this, the B object cannot be converted to a D object:

```
derivedObject.Equals(base)
```

The derivedObject.Equals() method always returns false. If you don't check the object types exactly, you can easily get into this situation, in which the order of the comparison matters.

All of the examples above also showed another important practice when you override Equals(). Overriding Equals() means that your type should implement IEquatable<T>. IEquatable<T> contains one method: Equals(T other). Implemented IEquatable<T> means that your type also supports a type-safe equality comparison. If you consider that the Equals() should return true only in the case where the right-hand side of the equation is of the same type as the left side, IEquatable<T> simply lets the compiler catch numerous occasions where the two objects would be not equal.

There is another practice to follow when you override Equals(). You should call the base class only if the base version is not provided by System.Object or System.ValueType. The previous code provides an example. Class D calls the Equals() method defined in its base class, Class B. However, Class B does not call baseObject.Equals(). It calls the version defined in System.Object, which returns true only when the two arguments refer to the same object. That's not what you want, or you wouldn't have written your own method in the first place.

The rule is to override Equals() whenever you create a value type, and to override Equals() on reference types when you do not want your reference type to obey reference semantics, as defined by System.Object. When you write your own Equals(), follow the implementation just outlined. Overriding Equals() means that you should write an override for GetHashCode(). See Item 7 for details.

We're almost done. operator==() is simple. Anytime you create a value type, redefine operator==(). The reason is exactly the same as with the instance Equals() function. The default version uses reflection to compare the contents of two value types. That's far less efficient than any implementation that you would write, so write your own. Follow the recommendations in Item 46 to avoid boxing when you compare value types.

Notice that I didn't say that you should write operator==() whenever you override instance Equals(). I said to write operator==() when you create value types. You should rarely override operator==() when you create reference types. The .NET Framework classes expect operator==() to follow reference semantics for all reference types.

Finally, you come to IStructuralEquality, which is implemented on System.Array and the Tuple<> generic classes. It enables those types to implement value semantics without enforcing value semantics for every comparison. It is doubtful that you'll ever create types that implement IStructuralEquality. It is needed only for those lightweight types. Implementing IStructuralEquality declares that a type can be composed into a larger object that implements value-based semantics.

C# gives you numerous ways to test equality, but you need to consider providing your own definitions for only two of them, along with supporting the analogous interfaces. You never override the static Object.ReferenceEquals() and static Object.Equals() because they provide the correct tests, regardless of the runtime type. You always override instance Equals() and operator==() for value types to provide better performance. You override instance Equals() for reference types when you want equality to mean something other than object identity. Anytime you override Equals() you implement IEquatable<T>. Simple, right?