## Item 32: Avoid ICloneable

ICloneable sounds like a good idea: You implement the ICloneable interface for types that support copies. If you don't want to support copies, don't implement it. But your type does not live in a vacuum. Your decision to support ICloneable affects derived types as well. Once a type supports ICloneable, all its derived types must do the same. All its member types must also support ICloneable or have some other mechanism to create a copy. Finally, supporting deep copies is very problematic when you create designs that contain webs of objects. ICloneable finesses this problem in its official definition: It supports either a deep or a shallow copy. A shallow copy creates a new object that contains copies of all member variables. If those member variables are reference types, the new object refers to the same object that the original does. A deep copy creates a new object that copies all member variables as well. All reference types are cloned recursively in the copy. In built-in types, such as integers, the deep and shallow copies produce the same results. Which one does a type support? That depends on the type. But mixing shallow and deep copies in the same object causes quite a few inconsistencies. When you go wading into the ICloneable waters, it can be hard to escape. Most often, avoiding ICloneable altogether makes a simpler class. It's easier to use, and it's easier to implement.

Any value type that contains only built-in types as members does not need to support ICloneable; a simple assignment copies all the values of the `struct` more efficiently than Clone(). Clone() must box its return so that it can be coerced into a System.Object reference. The caller must perform another cast to extract the value from the box. You've got enough to do. Don't write a Clone() function that replicates an assignment.

What about value types that contain reference types? The most obvious case is a value type that contains a `string`:

```
public struct ErrorMessage
{
    private int errCode;
    private int details;
    private string msg;

    // details elided
}
```

`string` is a special case because this class is immutable. If you assign an error message object, both error message objects refer to the same `string`. This does not cause any of the problems that might happen with a general reference type. If you change the msg variable through either reference, you create a new `string` object (see Item 16).

The general case of creating a `struct` that contains arbitrary reference variables is more complicated. It's also far rarer. The built-in assignment for the `struct` creates a shallow copy, with both `structs` referring to the same object. To create a deep copy, you need to clone the contained reference type, and you need to know that the reference type supported a deep copy with its Clone()

method. Even then, that will only work if the contained reference type also supports ICloneable, and its Clone() method creates a deep copy.

Now let's move on to reference types. Reference types could support the ICloneable interface to indicate that they support either shallow or deep copying. You could add support for ICloneable judiciously because doing so mandates that all classes derived from your type must also support ICloneable. Consider this small hierarchy:

```
class BaseType : ICloneable
{
    private string label = "class name";
    private int[] values = new int[10];

    public object Clone()
    {
        BaseType rVal = new BaseType();
        rVal.label = label;
        for (int i = 0; i < values.Length; i++)
            rVal.values[i] = values[i];
        return rVal;
    }
}

class Derived : BaseType
{
    private double[] dValues = new double[10];

    static void Main(string[] args)
    {
        Derived d = new Derived();
        Derived d2 = d.Clone() as Derived;

        if (d2 == null)
            Console.WriteLine("null");
    }
}
```

If you run this program, you will find that the value of d2 is null. The Derived class does inherit ICloneable.Clone() from BaseType, but that implementation is not correct for the Derived type: It only clones the base type. BaseType.Clone() creates a BaseType object, not a Derived object. That is why d2 is null in the test program—it's not a Derived object. However, even if you could overcome this problem, BaseType.Clone() could not properly copy the dValues array that was defined in Derived. When you implement ICloneable, you force all derived classes to implement it as well. In fact, you should provide a hook function to let all derived classes use your implementation (see Item 23). To support cloning, derived classes can add only member variables that are value types or reference types that implement ICloneable. That is a very stringent limitation on all derived classes. Adding ICloneable support to base classes usually creates such a burden on derived types that you should avoid implementing ICloneable in nonsealed classes.

When an entire hierarchy must implement ICloneable, you can create an abstract Clone() method and force all derived classes to implement it. In those cases, you need to define a way for the derived classes to create copies of the base members. That's done by defining a protected copy constructor:

```
class BaseType
{
    private string label;
    private int[] values;

    protected BaseType()
    {
        label = "class name";
        values = new int[10];
    }

    // Used by devived values to clone
    protected BaseType(BaseType right)
    {
        label = right.label;
        values = right.values.Clone() as int[];
    }
}

sealed class Derived : BaseType, ICloneable
{
    private double[] dValues = new double[10];

    public Derived()
    {
        dValues = new double[10];
    }
    // Construct a copy
    // using the base class copy ctor
    private Derived(Derived right) :
        base(right)
    {
        dValues = right.dValues.Clone()
          as double[];
    }

    public object Clone()
    {
        Derived rVal = new Derived(this);
        return rVal;
    }
}
```

Base classes do not implement ICloneable; they provide a protected copy constructor that enables derived classes to copy the base class parts. Leaf classes, which should all be sealed, implement ICloneable when necessary. The `base` class does not force all derived classes to implement ICloneable, but it provides the necessary methods for any derived classes that want ICloneable support.

ICloneable does have its use, but it is the exception rather than rule. It's significant that the .NET Framework did not add an ICloneable<T> when it was updated with generic support. You should never add support for ICloneable to value types; use the assignment operation instead. You should add support for ICloneable to leaf classes when a copy operation is truly necessary for the type. Base

classes that are likely to be used where ICloneable will be supported should create a protected copy constructor. In all other cases, avoid ICloneable.