The ICloneable Interface

As you might recall from System.Object defines a method named MemberwiseClone(). This

method is used to obtain a *shallow copy* of the current object.

Object users do not call this method directly, as it is **protected**.

However, a given object may call this method itself during the *cloning* process.

To illustrate, create a new Console Application named CloneablePoint that defines a class named Point:

// A class named Point.

public class Point

{

public int X {get; set;}

public int Y {get; set;}

public Point(int xPos, int yPos) { X = xPos; Y = yPos;}

public Point(){}

// Override Object.ToString().

public override string ToString()

{ return string.Format("X = {0}; Y = {1}", X, Y ); }

}

Given what you already know about reference types and value types you are aware

that if you assign one reference variable to another, you have two references pointing to the same object in memory.

Thus, the following assignment operation results in two references to the same Point object on the heap; modifications using either reference affect the **same object** on the heap:

static void Main(string[] args)

{

Console.WriteLine("\*\*\*\*\* Fun with Object Cloning \*\*\*\*\*\n");

// Two references to same object!

Point p1 = new Point(50, 50);

Point p2 = p1;

p2.X = 0;

Console.WriteLine(p1);

Console.WriteLine(p2);

Console.ReadLine();

}

When you want to give your custom type the ability to return an identical copy of itself to the caller, you may implement the standard **ICloneable interface**. As this type defines a single method named Clone():

public interface ICloneable

{

object Clone();

}

Obviously, the implementation of the Clone() method varies among your classes. However, the

basic functionality tends to be the same: copy the values of your member variables into a new object instance of the same type, and return it to the user. To illustrate, ponder the following update to the Point class:

// The Point now supports "clone-ability."

public class Point : ICloneable

{

public int X { get; set; }

public int Y { get; set; }

public Point(int xPos, int yPos) { X = xPos; Y = yPos; }

public Point() { }

// Override Object.ToString().

public override string ToString()

{ return string.Format("X = {0}; Y = {1}", X, Y); }

// Return a copy of the current object.

public object Clone()

{

return **new** Point(this.X, this.Y); }

}

In this way, you can create exact stand-alone copies of the Point type, as illustrated by the following code:

static void Main(string[] args)

{

Console.WriteLine("\*\*\*\*\* Fun with Object Cloning \*\*\*\*\*\n");

// Notice Clone() returns a plain object type.

// You must perform an explicit cast to obtain the derived type.

Point p3 = new Point(100, 100);

Point p4 = (Point)p3.Clone();

// Change p4.X (which will not change p3.X).

p4.X = 0;

// Print each object.

Console.WriteLine(p3);

Console.WriteLine(p4);

Console.ReadLine();

}

While the current implementation of Point fits the bill, you can streamline things just a bit. Because the Point type does not contain any internal reference type variables, you could simplify the implementation of the Clone() method as follows:

public object Clone()

{

// Copy each field of the Point member by member.

return this.MemberwiseClone();

}

Be aware, however, that if the Point did contain any reference type member variables,

MemberwiseClone() will copy the references to those objects (i.e., a *shallow copy*).

If you want to support a true *deep copy*, you will need to create a new instance of any reference type variables during the cloning process. Let’s see an example next.

**A More Elaborate Cloning Example**

Now assume the Point class contains a reference type member variable of type PointDescription. This class maintains a point’s friendly name

Here is the implementation:

// This class describes a point.

public class PointDescription

{

public string PetName {get; set;}

public PointDescription()

{

PetName = "No-name";

}

}

The initial updates to the Point class itself included modifying ToString() to account for these new bits of state data, as well as defining and creating the PointDescription reference type. To allow the outside world to establish a pet name for the Point, you also update the arguments passed into the overloaded constructor:

public class Point : ICloneable

{

public int X { get; set; }

public int Y { get; set; }

public PointDescription desc = new PointDescription();

public Point(int xPos, int yPos, string petName)

{

X = xPos; Y = yPos;

desc.PetName = petName;

}

public Point(int xPos, int yPos)

{

X = xPos; Y = yPos;

}

public Point() { }

// Override Object.ToString().

public override string ToString()

{

return string.Format("X = {0}; Y = {1}; Name = {2};\nID = {3}\n",

X, Y, desc.PetName, desc.PointID);

}

// Return a copy of the current object.

public object Clone()

{

**return this.MemberwiseClone();** }

}

Notice that you did not yet update your Clone() method. Therefore, when the object user asks for a clone using the current implementation, a shallow (member-by-member) copy is achieved. To illustrate,

assume you have updated Main() as follows:

static void Main(string[] args)

{

Console.WriteLine("\*\*\*\*\* Fun with Object Cloning \*\*\*\*\*\n");

Console.WriteLine("Cloned p3 and stored new Point in p4");

Point p3 = new Point(100, 100, "Jane");

Point p4 = (Point)p3.Clone();

Console.WriteLine("Before modification:");

Console.WriteLine("p3: {0}", p3);

Console.WriteLine("p4: {0}", p4);

p4.desc.PetName = "My new Point";

p4.X = 9;

Console.WriteLine("\nChanged p4.desc.petName and p4.X");

Console.WriteLine("After modification:");

Console.WriteLine("p3: {0}", p3);

Console.WriteLine("p4: {0}", p4);

Console.ReadLine();

}

Notice in the following output that while the value types have indeed been changed, the internal

reference types maintain the same values, as they are “pointing” to the same objects in memory

(specifically, note that the pet name for both objects is now “My new Point”).

\*\*\*\*\* Fun with Object Cloning \*\*\*\*\*

Cloned p3 and stored new Point in p4

Before modification:

p3: X = 100; Y = 100; Name = Jane;

p4: X = 100; Y = 100; Name = Jane;

Changed p4.desc.petName and p4.X

After modification:

p3: X = 100; Y = 100; Name = My new Point;

p4: X = 9; Y = 100; Name = My new Point;

To have your Clone() method make a complete deep copy of the internal reference types, you need

to configure the object returned by MemberwiseClone()

Here is one possible implementation:

// Now we need to adjust for the PointDescription member.

public object Clone()

{

// First get a shallow copy.

**Point newPoint = (Point)this.MemberwiseClone();**

// Then fill in the gaps.

**PointDescription currentDesc = new PointDescription();**

**currentDesc.PetName = this.desc.PetName;**

**newPoint.desc = currentDesc;**

**return newPoint;**

}

If you rerun the application once again and view the output (see below), you see that the Point

returned from Clone() does copy its internal reference type member variables (note the pet name is now

unique for both p3 and p4).

\*\*\*\*\* Fun with Object Cloning \*\*\*\*\*

Cloned p3 and stored new Point in p4

Before modification:

p3: X = 100; Y = 100; Name = Jane;

p4: X = 100; Y = 100; Name = Jane;

Changed p4.desc.petName and p4.X

After modification:

p3: X = 100; Y = 100; Name = Jane;

p4: X = 9; Y = 100; Name = My new Point;

To summarize the cloning process, if you have a class or structure that contains nothing but value types, implement your Clone() method using MemberwiseClone(). However, if you have a custom type that maintains other reference types, you might want to create a new object that takes into account each reference type member variable, in order to get a “deep copy.”