;login:

THE MAGAZINE OF USENIX & SAGE
April 2004 • volume 29 • number 2

inside:

PROGRAMMING

McCluskey: Using C# Abstract Classes

USENIX
The Advanced Computing Systems Association

using C# abstract classes

by Glen McCluskey

Glen McCluskey is a consultant with 20 years of experience and has focused on programming languages since 1988. He specializes in Java and C++ performance, testing, and technical documentation areas.

glenm@glenmccl.com

In our last column we discussed C# interfaces, a mechanism for specifying a contract, a particular set of methods, that an implementing class must define.

In this column we'll consider another somewhat similar feature known as abstract classes. Such classes are a basic design and structuring tool for C# applications and allow you to provide partial class implementations that can be customized.

An Example

Imagine that you're doing some work with benchmarking and performance analysis, and you'd like to develop some C# utility classes to aid in this effort. You need one utility that executes a particular routine or task repeatedly and keeps track of the elapsed time.

Here's some C# code that captures this idea:

```
using System;
using System.Threading;

abstract public class PerfUtils {
    abstract public void DoRun();

    public long TimeRun(int repcount) {
        long currtick = DateTime.Now.Ticks;
        for (int i = 0; i < repcount; i++)
            DoRun();
        return DateTime.Now.Ticks - currtick;
    }
}

public class BenchMark1 : PerfUtils {
    public override void DoRun() {
        Thread.Sleep(500);
    }
}

public class PerfUtilsDemo {
    public static void Main() {
        PerfUtils pu = new BenchMark1();
        long elapsed = pu.TimeRun(10);
        Console.WriteLine("elapsed time in milliseconds = " +
            elapsed / 10 / 1000);
    }
}
```

PerfUtils is an abstract class, meaning that it declares but does not define all its methods. An abstract class, like an interface, specifies a contract that must be fulfilled or implemented by another class (BenchMark1). In the case at hand, the TimeRun method is implemented, but the DoRun method is not – it's an application-specific method that a subclass must supply.

Since an abstract class does not have definitions for all its methods, it's not possible to create instances of such classes, and the following code will evoke a compiler error:

```
abstract public class A {
    abstract public void f();
}

public class AbstractNew {
    public static void Main() {
        A aref = new A();

        aref.f();    }

}
```

If such code was legal, then at runtime there might be calls to unimplemented methods.

We can say that an abstract class must be derived from or extended in order to be of any value; that is, there must be a further class that uses the abstract class. By contrast, the other extreme is a sealed class that cannot be derived from at all. For example, this code is illegal:

```
sealed public class A {
    void f() {}
}

public class B : A {}
```

Sealed classes are useful in a case where a derived class would alter the semantics of the class in some way, causing it to break.

Factoring Common Functionality

One of the key differences between an interface and an abstract class is that an abstract class can provide partial implementations of some methods, as a base, and thus factor out common functionality. By contrast, interface methods cannot be defined, and so the following code is invalid:

```
public interface IA {
    void f() {}
}

public class B : IA {
```

```
    public void f() {}

    public static void Main() {}
}
```

In the earlier example, the common functionality is TimeRun, a routine that's useful across a range of applications. It works with an application-specific method DoRun, supplied in a derived class.

Another example of factoring is the code below which illustrates a bit of a framework for putting together some collection classes (lists, hashtables, etc.):

```
public interface ICollection {
    int Size();

    bool IsEmpty();

    // ... other methods ...
}

abstract public class Collection : ICollection {
    abstract public int Size();

    public bool IsEmpty() {
        return Size() == 0;
    }

    // ... other methods ...
}

public class ListCollection : Collection {
    public override int Size() {
        // ... logic for computing size of ListCollection ...

        return 0; // dummy return
    }
}

public class Test {
    public static void Main() {
        ICollection ic = new ListCollection();
    }
}
```

ICollection is an interface that declares some common methods all collections will have. These methods include both Size, used to obtain the number of elements currently in the collection, and IsEmpty, which determines whether a collection is empty. Since IsEmpty can be implemented in terms of Size, it makes sense to provide a definition in the abstract class. By contrast, the appropriate logic to compute the size of a collection will vary, depending, for example, on whether the collection is a list or a hashtable.

Using interfaces and abstract classes together in this way is a very powerful technique. The abstract class serves as a means of factoring common functionality and providing an implementation for it. But because an interface is also defined, it's possible to sidestep the abstract class and start over with your own custom implementation that implements the interface. If you program in terms of interfaces, as we described in the last column, then it's easier to substitute your own implementation for that provided to you in a standard library.

Other Differences Between Interfaces and Abstract Classes

An abstract class can provide a partial implementation, as we mentioned above, whereas interfaces are used to specify but not implement a contract.

Another difference involves multiple inheritance. It's possible to implement more than one interface at a time, like this:

```
public interface IA {
    void f();
}

public interface IB {
    void g();
}

public class C : IA, IB {
    public void f() {}
    public void g() {}

    public static void Main() {}
}
```

whereas the corresponding code with abstract classes is not permitted:

```
abstract public class A {
    abstract public void f();
}

abstract public class B {
    abstract public void g();
}

public class C : A, B {
    public override void f() {}
    public override void g() {}

    public static void Main() {}
}
```

Implementing multiple unrelated interfaces can be quite useful, and sometimes the term "mixin" is used to describe this technique. For example, in the previous column we declared an interface:

```
public interface IDistance {
    double GetDistance(object obj);
}
```

A class implements this interface to provide functionality to compute the distance between two objects, for example the Euclidean distance between X,Y points or the number of days between two calendar dates. The class could implement several of these interfaces, each one adding a bit of functionality.

Polymorphic Programming

Our final example illustrates another aspect of programming with abstract classes. Modern object-oriented languages make use of what is called polymorphic programming, with virtual functions as another term for the same idea. This idea centers on programming with a common interface across a hierarchy of classes and their associated objects, and runtime binding for method calls.

We can tie down this concept by considering an example:

```
using System;

abstract public class A {
    abstract public void f1();

    public virtual void f2() {
        Console.WriteLine("A.f2");
    }

    public virtual void f3() {
        Console.WriteLine("A.f3");
    }
}

public class B : A {
    public override void f1() {
        Console.WriteLine("B.f1");
    }

    public new void f2() {
        Console.WriteLine("B.f2");
    }

    public override void f3() {
        Console.WriteLine("B.f3");
    }
}

public class Polymorphic {
    public static void Main() {
        A aref = new B();

        aref.f1();
        aref.f2();
        aref.f3();
    }
}
```

In this code, we create a new B object and assign it to a base class reference (A is the base of B). We then call methods f1, f2, and f3 through the base reference.

What happens when the methods are called? For f1, B.f1 is called, because the object pointed at by the A reference is really a B, and we specified that B.f1 overrides A.f1, and A.f1 is abstract anyway.

The same consideration applies to B.f3. The method is virtual (bound at runtime), and we're operating on a B object.

What about f2? It's marked as virtual in A, but B declares f2 to be "new," that is, the virtual dispatch hierarchy is broken. So A.f2 is called.

Virtual method dispatch is extremely powerful. For example, suppose that you have an abstract class Graphics that represents a graphics object and declares a Draw method. Then you have a variety of classes that extend the abstract class and that represent graphical objects like Circle and Line and Point and Rectangle. Instances of these classes can be assigned to a Graphics reference (pointer), and then the Draw method can be called on each instance, without worrying about the exact type of the object being referenced.

Abstract classes are fundamental building blocks that you can use to structure your C# programs. They are a good choice if you'd like to provide a partial class implementation that can be extended and customized.