C# Attributes and Reflection

Peter Sestoft
KVL and IT University of Copenhagen

C#/.Net project cluster

Tuesday 2 May 2006

- Attributes on types, fields, methods, parameters
- Declaring custom attribute classes
- C# reflection mechanisms and accessing attribute values

Attributes (C# Precisely section 28)

Attributes associate metadata with parts of a program, like Java 5.0 annotations.

A C# attribute is associated with a *target*, which may be an assembly, class, constructor, delegate type, enum type, event, field, interface, method, module, parameter, property, return value, or struct type.

Local variable declarations and individual statements cannot be targets. In Java local variable declarations can.

Example: The `Serializable` attribute indicates that a class can be serialized: written to an XML-document or byte stream (like the Java `Serializable` marker interface):

```
[Serializable()]
class SC { public int ci; }
```

Example: The `NonSerialized` attribute indicates that a field of a serializable class should not be serialized (like the Java `transient` field modifier):

```
[Serializable()]
class SO {
  public int i;
  public SC c;
  [NonSerialized()] public String s;
  ...
}
```


Some standard .Net attribute classes

Attribute Name	Targets	Meaning
Flags	Enum type	Print enum value combinations symbolically
Serializable	Class	Instances can be serialized and deserialized
NonSerialized	Field	Field is omitted when class is serialized
AttributeUsage	Class	Permissible targets for this attribute class
Diagnostics.Conditional	Method	Determine when (diagnostic) method should be called
Obsolete	All	Inform users that target is deprecated and should not be used

Declaring custom attribute classes

Example: An attribute `Author` to indicate authorship of a class or method:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
                AllowMultiple = true)]
class AuthorAttribute : Attribute {
  public readonly String name;
  public readonly Month mm;
  public AuthorAttribute(String name, Month mm) {
    this.name = name; this.mm = mm;
    Console.WriteLine("Creating AuthorAttribute: {0}", this);
  }
  public override String ToString() { return ...; }
}
public enum Month { Jan=1, Feb, Mar, Apr, May, Jun, ... }
```

Example use of the `Author` attribute, including multiple attributes on same target:

```
class TestAttributes {
  [Author("Donald", Month.May)]
  public void MyMethod1() { }
  [Author("Andrzej", Month.Jul)]
  [Author("Andreas", Month.Mar)]
  public void MyMethod2() { }
  ...
}
```


Attribute rules

An attribute must be a subclass of `System.Attribute`.

The name the class must have the form `XAttribute`.

An attribute occurrence is a call to a constructor in the class; only the `X` part of the name is used.

Attribute constructor arguments can be simple types or Object or String or System.Type or array of these.

The same attribute can be used multiple times on the same target, unlike in Java.

The attribute instance is created at run-time, every time the `GetCustomAttributes` method is called, unlike in Java.

It does not seem possible to specify the retention of an attribute, unlike in Java.

Accessing attribute values

Attributes can be accessed using the reflection mechanism.

Example: Print the `Author` attributes from methods called `MyMethodX` in class `TestAttributes`:

```
Type ty = typeof(TestAttributes);
foreach (MemberInfo mif in ty.GetMembers()) {
  if (mif.Name.StartsWith("MyMethod")) {
    Console.WriteLine("\nGetting attributes of {0}:", mif.Name);
    Object[] attrs = mif.GetCustomAttributes(false);
    Console.WriteLine("\nThe attributes of {0} are:", mif.Name);
    foreach (Attribute attr in attrs)
      Console.Write("{0} ", attr);
    Console.WriteLine();
    Console.WriteLine("\nGetting attributes of {0} again:", mif.Name);
    mif.GetCustomAttributes(false);
  }
}
```

The attribute instance is created at run-time, every time the `GetCustomAttributes` method is called.

The argument `false` to `GetCustomAttributes` means that inherited attributes are ignored.

Comparison with Java's reflection mechanism, part 1

The reflection mechanism in C# is very similar to that of Java.

	Java	C#
Class decription	`java.lang.Class`	`System.Type`
Class object for class `C`	`C.class`	`typeof(C)`
One method in class `co`	`co.getMethod(...)`	`co.GetMethod(...)`
Public methods in `co`	`co.getMethods()`	`co.GetMethods()`
One field in class `co`	`co.getField(...)`	`co.GetField(...)`
Public fields in `co`	`co.getFields()`	`co.GetFields(...)`
One constructor in `co`	`co.getConstructor(...)`	`co.GetConstructor(...)`
Public constructors in `co`	`co.getConstructors()`	`co.GetConstructors()`

Above, `co` represents a class.

Comparison with Java's reflection mechanism, part 2

	Java	C#
Reflection API	`java.lang.reflect`	`System.Reflection`
Method description	`Method`	`MethodInfo`
Field description	`Field`	`FieldInfo`
Constructor description	`Constructor`	`ConstructorInfo`
Call method	`mo.invoke(...)`	`mo.Invoke(...)`
Get field's value	`fo.get(...)`	`fo.GetValue(...)`
Set field's value	`fo.set(...)`	`fo.SetValue(...)`
Call constructor	`cco.newInstance(...)`	`cco.Invoke(...)`

Above, `mo` represents a method, `fo` represents a field, and `cco` represents a constructor.

But C# has more concepts: an *assembly* contains modules, a *module* contains types, a *type* contains methods.

Example: Get the run-time type objects corresponding to some types

The expression `typeof(t)` of type Type is the type object representing type `t`.

Unlike in Java, such a type object exists for all types, including generic type instances:

```
class Reflect0G {
  public static void Main(String [] args) {
    Type ty1 = typeof(int);
    Console.WriteLine(ty1);
    Console.WriteLine(typeof(long));
    Console.WriteLine(typeof(Reflect0G));
    Console.WriteLine(typeof(String[][]));
    Console.WriteLine(typeof(C<int>));
    Console.WriteLine(typeof(C<C<int>>));
  }
}
class C<T> { }
```

Also, the type arguments of a generic type can be found using `ty.GetGenericArguments()`.

Example: Call a method using reflection

```
using System;                                // For String
using System.Reflection;                     // For MethodInfo

class Reflect1 {
  public static void Main(String [] args) {
    Type co = typeof(Reflect1);              // Get the Reflect1 class (type)
    MethodInfo mo = co.GetMethod("Foo");     // Get the Foo method
    mo.Invoke(null, new Object[] { });       // Call it
  }

  public static void Foo() {
    Console.WriteLine("Foo was called");
  }
}
```


Example: List public static methods; call those whose names begin with `Test`

```
class Reflect2 {
  public static void Main(String [] args) {
    Type co = typeof(Reflect2);                          // Get Reflect2 class
    MethodInfo[] mos = co.GetMethods();                  // Get all methods
    Console.WriteLine("These public static methods are available:");
    for (int i=0; i<mos.Length; i++)
      if (mos[i].IsStatic)
        Console.WriteLine(mos[i].Name);
    Console.WriteLine();
    Console.WriteLine("Calling public static methods whose names start with Test:");
    for (int i=0; i<mos.Length; i++)
      if (mos[i].IsStatic && mos[i].Name.IndexOf("Test") == 0)
        mos[i].Invoke(null, new Object[] {});
  }

  public static void Foo() { Console.WriteLine("Foo"); }
  public void NonStaticFoo() { Console.WriteLine("NonStaticFoo"); }
  static void NonPublicFoo() { Console.WriteLine("NonPublicFoo"); }
  public static void TestGoo() { Console.WriteLine("TestGoo"); }
  public static void TestFoo() { Console.WriteLine("TestFoo"); }
}
```