Review of C#2 Features – Part II

Nico Ludwig (@ersatzteilchen)

TOC

- Review of C#2 Features – Part II
  - Delegates, Methods and Method Groups
  - Anonymous Methods
  - Variance of Delegates

- Sources:
  - Jeffrey Richter, Applied Microsoft .NET Framework Programming
  - Jon Skeet, CSharp in Depth

Delegates in C#1

See accompanying project <DelegateExamples>
Presents how we work with Delegates in C#1.

Delegates, Methods and Method Groups

- Delegates are types: A Delegate describes a type of a method.
  - Any Delegate eventually derives from *Delegate* (*MulticastDelegate*).
- A method matching the Delegate can be called by an instance of that Delegate.
  - Similar to function pointers in C/C++.
  - (Multicasting (i.e. a delegate as a collection of methods (an invocation list)) is an extra feature of Delegates.)
- All overloads of a method visible in the current context are called method group.
  - The method group is a kind of symbolic name of a method.
- Most important uses of Delegates:
  - Working with events (e.g. in UIs *Delegate* instances are used as callbacks).
  - Passing around code to be executed in a different thread (*Control.Invoke*(*Delegate*), new *Thread*(*ThreadStart*), *ThreadPool.QueueUserWorkItem*(*WaitCallback*) etc.).

- A method group is the set of all methods found by a member lookup of a certain method name in a certain context. E.g. all overloads of a method being effectively present due to inheritance from multiple types are part of that method's method group.

Simplified Delegate Instantiation in Action

See accompanying project <DelegateExamples>
Presents simplified Delegate Instantiation in C#2.

Simplified Delegate Instantiation

```
// Creating a Delegate instance in C#1:
aDelegateInstance = new AcceptString(Foo);
```

```
// New in C#2: create a Delegate instance from a method group:
aDelegateInstance = Foo;
```

- In C#2 it's allowed to instantiate a *Delegate* from a method group item directly.
  - I.e. the immediate step to create a *Delegate* instance can be skipped.

```
delegate void EventHandler(object sender, EventArgs e); // The Delegate.
private void MyForm_Load(object sender, EventArgs e) { /* pass */ } // An event handler.
```

```
// Advise an event handler in C#1:
Load += new EventHandler(MyForm_Load);
```

```
// New in C#2: advise event handler from a method group:
Load += MyForm_Load;
```

- (Perspective: an Event/delegate object is just a collection of methods in .NET)
- Event handlers can also be advised from a method group in C#2.
  - I.e. method groups can be chained to combine them to an invocation list directly.

- The definition of a Delegate type looks a little bit like a typedef in C/C++.

First Example with Anonymous Methods

See accompanying project <DelegateExamples>
First example with anonymous methods.

From Simplified Delegate Instantiation to Anonymous Methods

```
// Initialization of a Delegate instance the conventional way:
AcceptString printToConsole = new AcceptString(PrintSomethingToConsole);
```

```
// New in C#2: Initialization of a Delegate instance with an anonymous method:
AcceptString printToConsole = delegate(string s)
{
    Debug.WriteLine(s);
};
```

- Anonymous methods: *Delegate* instance code can be inlined completely.
  - I.e. a method can be defined as an anonymous block of code.
  - Similar concepts: Closure (Groovy, Python, ECMAScript), Block (Smalltalk, Ruby, Objective-C)
  - An anonymous method can access outer local contexts/variables (i.e. closures)!
  - The signature of the anonymous method is checked against the Delegate.
  - This is an outstanding feature in C#2!

- The syntax is similar to ECMAScript's/JavaScript's function literals (var f = function(x) { return x * x; }). – In other words, JavaScript has this feature for ages.

Anonymous Methods underneath

See accompanying project <DelegateExamples>
Anonymous methods underneath.

More on anonymous Methods

- For each anonymous method the compiler will generate an ordinary method.
- For anonymous methods the compiler may (MS specific) generate a class.
  - Classes are needed to put closures into effect with locals.
    - Such a class has a method containing the anonymous method's code.
    - Such a class has fields to hold/capture references of the local context respectively.
    - If instance members are accessed in the anonymous method, a field to hold this is added.
  - No class needs to be generated, if no outer locals are being referenced.
    - Then the anonymous method's code is held by an ordinary class or instance method.
- Attention: The lifetime of captured locals is extended! Captured references can't be garbage collected (gc'd), as long as the *Delegate* instance wasn't gc'd!
  - An anon. method (*Delegate* instance) has a lexical binding of the enclosing scope's locals and the this reference! -> You can access these bindings in the anon. method.
  - If you use these bindings in an anon. method, the bound objects can only be gc'd, if the *Delegate* instance expressed by the anon. method is gc'd.

- From within anonymous methods you can refer the enclosing scope's locals and also the this reference in instance methods of reference types. So in anonymous methods locals and the this reference are bound lexically.
- MS specific: The local variables will be captured as instance variables of a compiler generated "capture-class" that also holds the anonymous method as instance method. An instance of this "capture-class" is created locally where the anonymous method was written and provides the mentioned instance method as Delegate instance. If this is captured, a field to hold it will be created as well, and this will also be the Target of the Delegate instance (in other cases it is null). When the instance of the "capture-class" is gc'd, the captured variables (as instance variables) can be gc'd as well.
- Attention, there is a source of memory leaks: Captured references can't be gc'd, as long as the Delegate instance (or the "capture-class") wasn't gc'd! And the Delegate instance's target object can not be gc'd as long as the object, to which the Delegate instance was advised as event handler wasn't gc'd!
- Another problem: during run time "names" of anonymous methods are obfuscated to unspeakable compiler-generated symbols. These symbols are then visible in stack-traces and in reverse-engineered code.
  - A different way to understand this: anonymous methods a esp. suitable for methods that are not reusable

Everyday use of anonymous Methods

See accompanying project <DelegateExamples>
Everyday use of anonymous Methods.

Transporting Code with Delegates

- Another perception of Delegate instances: a way to pass code to methods.
- Therefor .NET 2 provides two generic multipurpose Delegate types:

```
delegate void Action<T>(T item);          // A Delegate describing methods acting on item.

delegate bool Predicate<T>(T item);       // A Delegate describing methods that check a
                                          // predicate on item.
```

- On the opposite some types (e.g. *List<T>*) provide methods that make use of those new *Delegate* types e.g.:

```
public void List<T>.ForEach(Action<T> action);          // Do action for each item.

public bool List<T>.RemoveAll(Predicate<T> match); // Remove all items that
                                                    // match the passed Predicate.
```

- These new tools provide ways to get rid of explicitly writing loops.
  - But this is just the beginning of the "transporting code" story, in C#3 this concept has been developed to the next level to work w/ any enumerable object: with LINQ.

- In .NET 1.x most developers used Delegates (types and instances) for two purposes:
  - Handling events (e.g. in UIs).
  - Passing around code to be executed in a different thread (e.g. by *Control.Invoke*, or by *Thread*(*ThreadStart*), or by *ThreadPool.QueueUserWorkItem*(*WaitCallback*)).

Delegate Variance

See accompanying project <DelegateExamples>
These examples show Delegate co- and contravariance.

Delegates are variant in C#2

- Variance expresses compatibility of *Delegate* instances to a *Delegate* type, if:
  - the return types of *Delegate* instance and type are related: as covariance; and if
  - the parameter types of *Delegate* instance and type are related: as contravariance.
- Covariant return type:
  - The instance's return type can be more derived than the *Delegate* type's.
  - I.e. a *Delegate* instance may "deliver more" than the *Delegate* type.
- Contravariant parameter type:
  - An instance's parameter types can be less derived than the *Delegate* type's.
  - I.e. a *Delegate* instance may "require less" than the *Delegate* type.
  - Fine for event handlers: you get more derived *EventArgs* and deal with the base type.
- (This has nothing to do with generic variance available in C#4/.NET 4!)

- In the development phase of the CLR 2, this feature was called "relaxed Delegates".

Thank you!