New C#3 Features (LINQ) – Part I

Nico Ludwig (@ersatzteilchen)

TOC

- New C#3 Features – Part I
  - Focus of C#3 Features
  - Automatically implemented Properties
  - Object and Collection Initializers: new Ways to initialize Objects and Collections

- Sources:
  - Jon Skeet, CSharp in Depth

Changes in .NET 3/3.5 – Overview

- .NET 3/3.5 runtime
  - All the .NET 3 and .NET 3.5 features are based on .NET 2 features.
  - The CLR wasn't even changed, .NET 3.5 still uses a CLR 2 underneath.
- .NET 3/3.5 libraries
  - Some new libraries have been added:
    - Windows Presentation Foundation (WPF)
    - Windows Communication Foundation (WCF)
    - Windows Workflow Foundation (WF)
    - Windows CardSpace
  - Some present APIs have been extended.
- C#3 and VB9 got syntactical enhancements to cope with LINQ.
  - The compilers have been updated, but no changes to the CIL have been done.

- Windows CardSpace is a system to manage different identities against other environments, e.g. web sites. CardSpace is like your wallet with different cards representing different identities.

Which Aims does C#3 pursue?

- Focus of C#3 features:
  - Enhance readability and reduce syntactic fluff.
  - Increase productivity: "less code that monkeys could write".
  - More expressiveness to support a functional programming style.
  - LINQ
- C#3 requires the .NET 3.5 being installed at minimum.
  - As well as Visual Studio 2008 or newer for development.
- In this presentation:
  - The new boilerplate features of C#3 will be examined.

Automatically implemented Properties

```
// Public property Age in C#1/C#2:
private int _age;
public int Age
{
    get { return _age; }
    set { _age = value; }
}
```

→

```
// New in C#3: Age as automatically implemented property:
public int Age { get; set; }
```

- C#3 allows the definition of properties with a simplified syntax.
  - OK, it can only be used to define simple get/set properties with bare backing fields.
  - A real no-brainer, it; reduces the amount of required code.
  - Idea taken from C++/CLI (trivial properties) and Objective-C (synthesized properties).
  - Gotcha: Automatically implemented properties can break serialization.
  - (Not available for indexers. Default values can't be specified on them explicitly.)
- In principle field-like events are automatically implemented by default since C#1
  - The methods add and remove and the backing field can be implemented optionally.

5

- .NET's properties support the so called Unified Access Principle (UAP). UAP means that all the data of an object can be accessed and manipulated with the same syntax. Automatically implemented properties takes the UAP to the next level, because implementing properties is so easy now. Later on automatically implemented properties can be "promoted" to full properties in order to add logic to setters and getters.
- The gotcha: Serialization can depend on the names of the fields of a type getting serialized (this is esp. true for the *BinaryFormatter*, which is heavily used for .NET Remoting). And for automatically implemented properties the name of the field may change when the type is modified. So an instance's data being serialized in past may no longer deserialize successfully later on, if the type has been modified in the meantime. Put simple: don't use automatically implemented properties in serializable types!
- Automatically implemented properties are not suitable for immutable types as the backing field is always writeable.
- VB 10 supports auto-implemented properties with default values.
- Also present in Ruby with the *attr_accessor* generator method.

Automatic Properties and Object Initializers in Action

See accompanying project <InitializerExamples>
Presents automatically implemented properties and object initializers.

Object Initializers

```
// Our toy: class Person with some properties and ctors:
public class Person
{
    public int Age { get; set; }
    public string Name { get; set; }
    public Person() {}
    public Person(string name) { Name = name; }
}
```

```
// Create a Person object and
// set property Age in C#1/C#2:
Person p = new Person();
p.Age = 42;
```

```
// New equivalent in C#3: Create (dctor) a Person object and
// set property Age with one expression as object initializer:
Person p = new Person { Age = 42 };
```

```
// Create a 2nd Person object, set property Age with an
// object initializer and set Name with the ctor:
Person p2 = new Person { Name = "Joe", Age = 32 };
/* or: */ Person p3 = new Person("Joe") { Age = 32 };
```

- Object initializers: simplified initialization of properties during object creation.
  - Create a new object and initialize properties (and fields) with one expression.
  - Reduces the need for a bunch of overloaded ctors and conversion operators.
  - This single expression style supports functional paradigms required for LINQ.

7

- The individually assigned properties/fields are assigned in exactly the order you write them in the object initializer. After assigning the last item in the object initializer, you are allowed to leave a dangling comma.
- You are not allowed to advise event handlers in an object initializer.
- If an exception is thrown on the initialization of a property/field the ctor is called, but the assigned-to symbol is only default initialized afterwards (as if the exception happened in the ctor).
- Resources being initialized with object or collection initializers within a using-context are not exception save (*Dispose*() won' be called).
- C++11 introduced uniform initialization, which addresses the same idea.

More to come: Embedded Object Initializers

See accompanying project <InitializerExamples>
Presents embedded object initializers.

Embedded Object Initializers

```
// Let's introduce a new type 'Location':
public class Location
{
    // Just these two properties:
    public string Country { get; set; }
    public string Town { get; set; }
}
```

```
// Here is class Person with a property 'Home' of type 'Location':
public class Person
{
    // Field _home is created by all Person's ctors:
    private Location _home = new Location();
    public Location Home { get { return _home; } }
    // Other properties (Age, Name) and ctors...
}
```

```
// Create a Person object and apply an object initializer on property Home;
// initialize Home with an embedded object initializer for type Location:
Person p = new Person
{
    // Apply object initializer in an embedded manner:
    Home = { Country = "US", Town = "Boston" }
};
```

- Object initializers can be embedded as well.
  - But embedded objects can't be created, only initialized. I.e. *Home* must be created by *Person* (the 'new *Location*()' expression is executed in all ctors of *Person*)!

- The language specification refers to this as "setting the properties of an embedded object".
- Also readonly fields can be initialized like this.
- The field *_home* could be created (new *Location()*) by any dedicated ctor as well to be functional with embedded object initializers.

Collection Initializers

See accompanying project <InitializerExamples>
Presents Collection Initializers.

Hands on Collection Initializers

```
// C#2's way to create and fill a List<string>:
IList<string> names = new List<string>();
names.Add("Rose");
names.Add("Poppy");
names.Add("Daisy");
```



```
// New in C#3: create (dctor) and initialize a
// List<string> with a Collection initializer:
IList<string> names = new List<string>
{
    // Calls the method Add(string) for three times.
    "Rose", "Poppy", "Daisy"
};
```

- Remarks on Collection initializers:
  - They look like array initializer expressions (braced and comma separated value lists).
  - Can be used in concert with any ctor of the Collection to be initialized.
  - Can be used locally or for initialization of members.
  - Render the construction and initialization of a Collection into one expression.

- A Collection must fulfill some requirements to be used with Collection initializers:
  - The Collection must implement *IEnumerable* or *IEnumerable<T>*.
  - The Collection must provide any public overload (any signature) of method *Add*().

- After assigning the last item of the anonymous type, you are allowed to leave a dangling comma (you can also leave it on arrays).
- In Objective-C there are so called container literals, which solve the same problem like C#'s Collection initializers.

More Collection Initializers and implicitly typed Arrays

See accompanying project <InitializerExamples>
Presents more Collection initializers and implicitly typed arrays.

More on Collection Initializers and implicitly typed Arrays

```
// Apply a Collection initializer on a Dictionary<string, int> in C#3:
IDictionary<string, int> nameToAge = new Dictionary<string, int>
{
    // Calls the method Add(string, int) for three times.
    {"Rose", 31}, {"Poppy", 32}, {"Daisy", 24}
};
```

- Any public overload of method *Add*() can be used to initialize Collections.
  - Additionally the Collection initializers can be used as embedded object initializers!

```
// Assume we have a method with a string-array parameter:
private void AwaitsAStringArray(string[] strings);
```

```
// Pass a newly created array with an array initializer in C#1/2:
AwaitsAStringArray(new string[] { "Monica", "Rebecca", "Lydia" });
```

```
// New in C#3: leave the type name away on the new keyword:
AwaitsAStringArray(new[] { "Monica", "Rebecca", "Lydia" });
```

- This is a syntactic simplification, which is handy on passing arrays to methods.
  - In C#3 there exist situations, in which static types are unknown or anonymous.
  - Only allowed, if the initial array items can be implicitly converted to the same type.

13

- Under certain circumstances implicitly typed arrays can be initialized with initializer lists having mixed static types of their items:
  - There must be one type into which all the initializer items can be converted to implicitly.
  - Only the static types of initializer items' expressions are considered for this conversion.
  - If these bullets do not meet, or if all the initializer items are typeless expressions (null literals or anonymous methods with no casts) the array initializer will fail to compile.

What is the Sense of these Features?

- Removal of syntactical fluff.
  - There are many developers that longed for these simplifications.
  - Programmatic "creation" of test data is much simpler now.
  - Once again: functional, "one-expression"- programming style is possible.

- Why is the functional style preferred?
  - Syntactically concise.
  - Contributes to LINQ's expressiveness.

- Simplified initialization is the basis of the so-called "anonymous types" in C#3.
  - More to come in next lectures...

Thank you!