Review of C#2 Features – Part III

Nico Ludwig (@ersatzteilchen)

TOC

- Review of C#2 Features – Part III
  - Iterators
  - Nullable Types

- Sources:
  - Jon Skeet, CSharp in Depth

Iterator Support in C#1

- "Iterator" is one of the simplest (behavioral) Design Patterns.
  - The aim of iterator is to separate the Collection from the item access.
  - Not all items of a Collection are fetched, but only the current item.
- In .NET "iterator" is represented by the interface *IEnumerator*/<*T*>.
  - (This matches to COM's *IEnum*XXX interfaces directly.)
  - Iterator allows to program concise forward loops with an unknown end condition.
  - In C# usage, one encounters them when using Collections on foreach loops.
  - (During iteration the Collection and the current item are generally mutation guarded!)
- But it is relatively hard to write new iterators in C#1, e.g. to support foreach.
  - It is required to implement *IEnumerable*/*IEnumerator* in userdefined types.
  - It is recommended to implement them in two classes to separate responsibilities.
  - A few methods must be written and the state (the current item) must be maintained.

- In C# most known programming patterns are only supported indirectly, i.e. they can be implemented in C# easily. Few patterns are supported by dedicated language features, iterator is one of these patterns.
- We're going to discuss the iterator pattern in .NET.
- An object implementing *IEnumerator*/<*T*> is called iterator, or cursor or enumerator object.
- Extra benefit: Iterators are good to avoid off-by-one errors: there is no index.

Guts of the foreach Loop unveiled

See accompanying project <ForeachExample>
Expresses a foreach loop with a while loop to show the usage of iterators.

- C#'s iterator terms:
  - An object implementing *IEnumerable*/<*T*> is called sequence or enumerable object.
  - An object implementing *IEnumerator*/<*T*> is called iterator or enumerator object. Iterators are returned by a sequence's method *GetEnumerator*(). I.e. multiple iterators can correspond to a sequence. In database terms: *IEnumerable*/<*T*> is a table and *IEnumerator*/<*T*> is a cursor.

The C#2 Way to provide Iterators

- Wrap up: The target is to support foreach in user defined types somehow.
- In C#2 the new contextual keyword yield allows easy iterator implementation.
  - Indeed, this idea is not new, it is almost literally taken from Ruby/Python.
- How this simple iterator is used:
  - The iterator block is a method that returns *IEnumerator*/<*T*> or *IEnumerable*/<*T*> and uses the keyword yield (generally) in a loop to create the resulting sequence.
  - Iterator blocks enable deferred execution: no line of code in the body of the iterator block runs until the first call to *MoveNext*() is performed.
  - yield return puts the current item (of the loop) into the sequence (*IEnumerable*/<*T*>).
  - yield break terminates the iteration.
- A C#2 compiler creates classes implementing *IEnumerator*/<*T*> and *IEnumerable*/<*T*>.

5

- Iterator blocks are similar to so-called coroutines. Coroutines are a way to pass the execution control to another block of code and then return to the original execution point. It can be simulated by multiple threads or by other ways to store the original stackframe. In generated C# code: the status of the coroutine is stored on the stack as status and a switch statement acts on that status on each call of *MoveNext*().
- In C# an object being returned from an iterator block is called sequence or enumerable object if it is an *IEnumerable*/<*T*>. If the returned object is an *IEnumerator*/<*T*> it is called iterator or enumerator object in C#.
- (Btw.: Do not implement both *IEnumerator*/*IEnumerable* on the same type!)
- Within an iterator block a simple return is not allowed, but it can have multiple yield returns/breaks.
- It is also possible to implement a recursive iterator block or w/o any loop!

C#2 Iterators in Action

See accompanying project <IteratorExamples>
Presents examples of generator functions to produce sequences.

- A method returning an *IEnumerable*/<*T*> or *IEnumerable*/<*T*> is called iterator block when it uses one or many yield statements.
- Iterator blocks implement a state machine:
  - In opposite to an ordinary return statement the yield return statement seams only to temporarily step out and pause the method.
  - They are a form of continuation or coroutine or fibre (last in the programming language D, where the stackpointers will be really switched).
  - They enable deferred execution: no line of code in the body of the iterator block runs until the first call to *MoveNext*().
- The code of the finally blocks in iterator blocks will be executed:
  - when the iterator block leaves execution, or
  - when a yield break statement is executed, or
  - when the iterator will be disposed of, which is done by foreach automatically.

Some Words on C#2 Iterators

- This easy way to implement and think about iterators is very powerful:
  - Iterators are not limited to "collection-iteration", one can work w/o collections.
  - (E.g. you can implement iterator blocks as properties! – But that's not wise.)
  - Possibility to implement infinite mathematical sets and sequences directly from iteration.
  - Possibility to defer the calculation of data until they're really needed.
- Gotcha: Using generator *IEnumerable* to replace returning of Collections.
  - The *IEnumerable*<*T*> is generated every time the generator function is called/iterated!
  - Framework guideline: use methods returning iterators and properties returning collections.
- Gotcha: The mutation guard during foreach/iteration may not be checked.
  - It must be implemented manually if required.
- C#2 iterators are an important basis for "LINQ to Objects" in C#3.

- Anonymous methods can not be iterator blocks. It was counted to be too much effort for the C# compiler team.
  - Only top level methods can be iterator blocks.
  - Anonymous iterator blocks are supported in VB11!
- For similar reasons methods with out or ref parameters can not be iterator blocks.

Lack of Nullity for Value Types

- Value types cannot have no value.
  - A value type can at least have a default value (this is an incarnation of 0 or false).
  - On the other hand reference types are allowed to have no value, via null.

- Sometimes it is desirable to have nullable value types.
  - E.g. as seen on databases, where values of primitive type can be *NULL*.
  - So on doing database ORM with C# similar requirements arise as well.

- To simulate "no value" for value types there exist some patterns in C#1:
  - Sacrifice a sentinel value to express "no value", e.g. -1, *DateTime.MinValue* ...
  - Box the value type into a reference type, so null expresses "no value" again.
  - Add a boolean flag for the value type to indicate “no value”.

- In C#2 the last pattern has been promoted to a language feature.

Introducing the Type Nullable<T>

- C#2 provides the type *Nullable<T>* and syntactic sugar to provide nullity.

See accompanying project <NullableExample>
Presents examples of the usage of type Nullable<T>.

.NET 2 Support for Nullity

- Having a *Nullable<T>* instance, null is a value, not a null-reference any longer!
- Defined as struct *Nullable<T>*, where *T* must be a value type.
  - *Nullable<T>* is itself a value type! (I.e. locally created on stack, boxable etc.)
  - *T* is called the underlying type.
  - The property *HasValue* checks, whether the value type has a value (is not null).
  - The property *Value* allows to access the wrapped value type's value, though.
  - The method *GetHashCode*() returns 0, if the value has no value.
  - The method *Equals*() returns false, if one operand is null or unequal the other operand.
- The static class *Nullable* provides further tool methods:

```
public static class Nullable { // (members hidden)
    int Compare<T>(Nullable<T>, Nullable<T>);
    bool Equals<T>(Nullable<T>, Nullable<T>);
    Type GetUnderlyingType(Type nullableType);
}
```


- *Equals*/*Compare<T>*() use *EqualityComparer*/*Comparer<T>.Default*, which returns *IEqualityComparer<T>*. If T implements *IEquateable<T>* (which is most efficient in most cases, it calls its generic *Equals<T>*()), or it returns the default implementation that makes use of *Object*'s virtual *Equals*() (causing boxing) and *GetHashCode*().

C#2 Support for Nullity

- In addition to the syntax *Nullable<T>* C#2 allows the more concise syntax *T*?.

  Nullable<int> <=> int?

- *T*? allows the usage of operators by lifting them from the underlying type!
  - Lifting: For most operators of *T*, operators with *T*?-operands/results are generated.
  - E.g. for arithmetic and bitwise operators of *T*?: If any operand is null the result is null.

- *T*'s operators need to obey following restrictions for lifting:
  - Only operands with non-nullable types get lifted, return types must be non-nullable.
  - The return type of equality and relational operators must be bool.
  - (The operators true and false get never lifted.)
  - (Yes, this looks complicated, but everything works as expected.)

- Additionally C#2 provides extra logical operators for bool? (i.e. "treebool").
  - The operators | and & provide logic here, not ||/&& (they aren't allowed on bool?).

- Lifting of operators when *Nullable*s are used, results in more robust code, as null-values don't harm anymore. But lifting can also be confusing!
- More features of *Nullable*s in the attached example code:
  - The boxing behavior of nullables is special. When a *Nullable<T>* is boxed the wrapped type *T* will be boxed, not the "whole" *Nullable<T>*.
  - The as-operator can be used for reference types and for boxed value types as *Nullable*s.
- Following the .NET Framework Design Guidelines nullable value type should also be used instead of otherwise special types like *System.DBNull*.

The Null Coalescing Operator

- C#2 provides the null coalescing operator ?? to compact syntax.
  - It can be understood as conditional operator (?:) for nullity in C#.
  - This operator is binary, but right associative and cannot be overloaded.
  - It can be used for *Nullable<T>* and any reference type.
  - Good to write defaulting or fall-back patterns.

Not discussed Features new in C#2

- Primarily for code generation:
  - Partial types
  - Namespace aliases
  - #pragma checksum
- #pragma warning (disable|restore)
- fixed buffers in unsafe code
- Friend assemblies
- CLR 2:
  - Support for 64 bit processors (x86 and IA64).
  - CLR hosting within SQL Server 2005.

13

- The new feature "partial types" (for classes, structs and interfaces) is intended to part a type into multiple separate files to better handle generated code. Some files (typically one) contains the IDE-generated code, the other files can be edited by the programmer. It is e.g. used in the Forms API.
- The #pragma directives are instructions to the compiler. The Microsoft compiler understands the #pragmas warning and checksum. In future versions of the compiler other #pragmas, or compiler-vendor specific #pragmas could be provided. Unknown #pragmas will be ignored by the compiler.

Thank you!