New C#4 Features – Part I

Nico Ludwig (@ersatzteilchen)

TOC

- New C#4 Features – Part I
  - Focus of C#4 Features
  - Simplifications in Syntax
  - Simplifications for Working with COM
  - Improvements in Code Generation

- Sources:
  - Jon Skeet, CSharp in Depth

Focus of C#4

- Interoperability (basically not new CLR 4 features):
  - Enhancements for COM interoperability and PIA.
  - Dynamic typing and the Dynamic Language Runtime (DLR).
- VB and C# co/evolution (already in the CLR 1 and 2, now enabled in C#4):
  - Optional parameters, named arguments.
  - Configurable generic co- and contravariance for parameter and return types.
- Performance (new features in the CLR 4):
  - New threading model (esp. addressing thread pools and the gc).
  - Parallel extensions (Task Parallel Library (TPL), the types *Task* and *Parallel*, PLINQ).
- Code Quality:
  - Code Contracts

- Dynamic typing enables easy usage of COM types (e.g. Microsoft Office automation) and dynamic/scripting languages (IronRuby, IronPython).
- The CLR didn't need to change in order to enable the DLR.
- PLINQ is implemented as a set of extension methods on *ParallelQuery*<*T*>.

Named Arguments

```
// Let's assume that we'd like to call following method:
public void AMethod(int foo, int bar) { /* pass */ }
```

```
// Positional argument passing in C#1/C#2/C#3:
anInstance.AMethod(42, 21);
```

→

```
// New in C#4: named argument passing:
anInstance.AMethod(foo: 42, bar: 21);
/* or: */
anInstance.AMethod(bar: 21, foo: 42);
```

- Positional and named arguments can be used as alternatives or in combination.
- It can enhance readability.
- (Good if certain parameters often change their position in declarations.)
- The names of formal parameters are now part of a type's public interface in C#.
  - .NET/CLR supported named arguments from the start.

- Named arguments are present in VB for a long time. Smalltalk and Objective-C use a combination of positional and named argument passing for all method calls. In CLOS there exist so-called "keywords" that work similar to named argument passing.
- Being part of the public interface means:
  - Changing the name of a parameter could break calling code at compile time. We can circumvent this by avoiding the usage of named parameters or by avoiding to change the names of the parameters on public, or protected methods.

Named Arguments and optional Parameters in Action

See accompanying project <NamedArgumentsAndOptionalParameters>
Presents named arguments and optional parameters.

- These features will be shown in one example, because we will encounter them together in practice very often.
- Parameter: The variable being declared in the declaration of a function or method.
- Argument: The expression passed to a function or method.

Optional Parameters

```
// C#4's new syntax to declare a method having an optional parameter "bar":
public void DoSomething(int foo, int bar = 0) { /* pass */ }
```

```
// This method can be called this way:
anInstance.DoSomething(42, 56);
/* or this way: */
anInstance.DoSomething(42);
```

- Make parameters optional by specification of a default value constant.
  - Can be used with almost all kinds of methods (incl. interfaces' methods).
  - Reduces the need for overloads.
- The CLR supported them from the start (via the *OptionalAttribute*).
  - Optional parameters are CLS compliant, but...
- The default values are compile time bound (like the resolution of overloads).
  - Can cause version problems similar to public constants. – Callers must recompile!
  - Prevent version problems: named arguments, overloads or value reinterpretation.

6

- Optional parameters were present in VB before it was a .NET language (often used as hack to simulate non-existing overloads in earlier versions of VB).
- Exactly as for constants the optional parameters' values are stored in the metadata of the assembly, thus we can only use primitive types.
- The default values need to be compile time constants: literals, incl. null (i.e. default(<ReferenceType>)) or any parameterless ctor of a value type (i.e. default(<ValueType>)) or const members or enum values; an implicit conversion to the parameter type must be existent (not a user defined conversion). For empty strings we have to use "" rather than *string.Empty*, because later is not a compile time constant but a static property. The restrictions are similar to custom attribute values.
- If we run the code analysis (CA) against code using optional parameters we'll get the warning CA1026 "Default parameters should not be used". Overloads are considered the better feature, because languages w/o optional parameter support need to pass all parameters. – The CA says: better avoid it (after the Framework Design Guidelines 5.1.1).
  - The usage of optional parameters in internal and private methods is ok.
- To be frank there is one important advantage: optional parameters are itself a good documentation.
- Optional parameters seem to be similar to default arguments in C++. But they're not. In C++ default arguments don't need to be compile time constants. But in C++ we also have to recompile all callers of a function/method, if a default argument is a compile time constant and has been modified. Mind that in C++ we _have to recompile_ to do this, because the modification has then been done in an h-file (due to separated compilation); and whenever a h-file has changed _each_ includer must recompile. – Such a dependency isn't existing in C#, but recompilation against modified APIs comes into play when any public compile time constant has been changed.
- Optional parameters carry the "defaultvalue" attribute in COM.

Another practical Example: Value Interpretation

See accompanying project <NamedArgumentsAndOptionalParameters>
Solving versioning problems with value interpretation.

CLS Site of Named Arguments and Optional Parameters

- Default values of methods with optional parameters are stored on the caller site.
  - We can inspect the caller site (e.g. with Reflector) to see the default values (in IL code).
  - If the default value was modified, all callers need to recompile to get the new value.
  - If the callers don't recompile they will continue to use the previous default parameter.
  - Adding new optional parameters to public/protected methods will break at run time, if callers don't recompile!
  - => In effect methods can go silently incompatible! This is the versioning problem.
  - A ctor having only optional parameters is not accepted as parameterless ctor (dctor)!
  - (Method overloads are more robust than optional parameters.)

- The names of named arguments are stored on the caller site.
  - If a parameter's name was modified, formerly compiled callers will continue to work.
  - If a caller then tries to recompile, it needs to rename the named arguments respectively.
  - => Renamed parameters are only a breaking change on compile time afterwards.

- Method calls to methods with optional arguments can't be done in expression trees, if no arguments are passed.
- The break at run time does only happen, if a method of a referenced assembly was modified. Modifications in the same assembly can go really silently incompatible by using wrong or "shifted" default values.
- The versioning problem mentioned before is not existing when using overloads instead of optional parameters. Optional parameters couple the caller tighter than overloads.
- In C++ a ctor only having parameters with default arguments is automatically also a dctor.
- To make the versioning problem visible we can force developers to recompile their code, e.g. if we make the interface deliberately incompatible.

Summary: Named Arguments and Optional Parameters

- Enable features in C# that were already present in the CLR.
- Its combination is useful.
  - Esp. to reduce the noisiness of APIs like COM automation (more to come later).
- Can be used to with almost any kind of method (also ctors and indexers).
  - Can not be used to implement/call operator methods.
- Influence resolving of method overloads.
  - Named arguments may reduce the count of candidates on resolving.
  - Optional parameters may increase the count of candidates on resolving.
- These features were introduced into C# to improve the support of COM.

- Usefulness of the combination of named arguments and optional parameters:
  - Named arguments and optional parameters can save 15 overloads of a four-parameter method!
  - There exist MS Office APIs with up to 16 parameters! – We don't need to pass ten *Type.Missing* arguments and to fill the other six arguments with meaningful arguments, instead we just exploit the combination of named argument passing and optional parameters.
- C#'s indexers aren't operators. They are properties. They aren't static methods and they can be defined with another arity.
- (In C++ the subscript operator can only be binary. However the function call operator can have any arity and can, as the only operator in C++, have default arguments.)
- Indexers can have default arguments, but it makes only sense having indexers with more than one parameter, because we can't directly call an indexer w/o specifying at least one argument in C#.
- Named arguments reduce the count of candidates, because only overloads having parameters with exactly the written names will be considered.
- Optional arguments increase the count of candidates, because some methods could have more parameters than the number of passed arguments.
- Named arguments and optional parameters are similar to positional and named parameters in .NET custom attributes.

COM Interop Improvements in Action

See accompanying project <EvolutionOfComInteropImprovements>
This example presents some Microsoft specific simplifications for COM interoperability.

"COM is not dead, it is done!" Don Box, Nov. 2003

Summary: Simplifications for Working with COM

- I.e. simplifications for generated code of interop assemblies.
- COM does not support method overloads. Therefor, many parameters (and arguments) are required.
  - Optional parameters and named arguments come in handy...
  - ... in fact they have been introduced into C# only to support better COM interop.
- Almost all parameters in COM interfaces are ref parameters.
  - Values (e.g. literals) can be directly passed as arguments to ref parameters.
- One more simplification: Named indexers can be called easily.
- No-PIA deployment: PIA code can now be linked in instead of only referenced.
  - Only the used types will be linked/embedded into our resulting assembly.
  - Self contained assemblies have a smaller "footprint" and reduce version problems.

11

- Mind that the simplifications shown in this section are highly Microsoft dependent.
  - E.g. there exist MS Office COM APIs with up to 16 parameters!
- The usage of ref parameters in COM is due to performance gains (often seen with *VARIANT*s that must be passed as plain objects). Typically the arguments aren't modified within these COM methods.
- Named indexers can only be called in C#, but we can't define new ones. (These so called "indexed properties" are known in the CLR and applicable in VB since .NET 1. The unnamed indexed property (i.e. the "indexer") present in C# is called "default property/member".)
- Named indexers in generated interop code, that only have optional arguments, can also be called from C#.
- No-PIA deployment bases on .NET 4's new feature "type equivalence". It allows the run time to consider certain types as being interchangeable even if they are defined in different assemblies, so it enhances interoperability. The idea of PIA was that only one assembly contains all the types, this was relaxed with type equivalence. Every linked-in copy of the type gets a *TypeIdentifier* attribute that declares equivalent types with a common identifier (not for types with behavior, only interfaces, delegates, enums and PODs).
- No-PIA deployed types will enable dynamic coding that simplifies working with COM (esp. with *IDispatch* and *VARIANT*s) even further. This will be discussed in a later lecture...

Improvements in Code Generation

- Some less prominent changes for thread-safety made their way into C#4.
- The lock statement's implementation got more robust.
  - The generated code uses monitors in a thread-safe and exception-safe fashion.
- Field-like events' implementation got more robust.
  - The this reference or the type of the surrounding type won't be locked any longer.
    - Instead on add/remove an atomic *Interlocked.CompareExchange<T>()*-call is used.
  - The +=/-= operators will always be performed on the event field.
    - Also within the declaring class. Before C#3 it was performed on the backing field directly!
    - This was done to exploit the gained thread-safety as explained above (read: atomicity).
    - (On other operations (assignment/call), the backing field will still be used directly.)

Thank you!