New C#4 Features – Part III

Nico Ludwig (@ersatzteilchen)

TOC

- New C#4 Features – Part III
  - Why dynamic Programming
  - Dynamic Programming in VB
  - Dynamic Programming in C#4

- Sources:
  - Jon Skeet, C# in Depth
  - Chaur Wu, Pro DLR in .NET 4
  - Programming Microsoft® LINQ in Microsoft .NET Framework 4
  - MSDN Magazine 05/10, 06/10, 0710, 02/11
  - Software Engineering Radio; Episode 28: Type Systems, <http://www.se-radio.net/2006/09/episode-28-type-systems/>
  - Software Engineering Radio; Episode 49: Dynamic Languages for Static Minds, <http://www.se-radio.net/2007/03/episode-49-dynamic-languages-for-static-minds/>

- "Programming Microsoft® LINQ in Microsoft .NET Framework 4" provides a good overview of programming with Expression Trees.

Dynamic Typing as major new Feature in .NET 4/C#4

- A big topic in .NET 4 is interoperability.
- Interoperability with other technologies is often a pain.
  - COM interop.
  - Explorative APIs that sit upon a dynamic object model (e.g. DOM).
  - Remoting technologies with loosely coupling, like SOA and cloud computing do rise.
    - Services (read: operations or methods) may be available or not.
  - Domain Specific Languages (DSLs).
  - New scripting languages came into play, (some even base on the CLR).
    - They're especially employing dynamic typing.
  - => As developers we encounter more and more things having a dynamic interface.
- Besides interoperability, dynamic typing enables other features.
  - Scripting .NET and with .NET. opens your application's object model to be scripted.
  - Mighty data-driven patterns (expando objects and dynamic Expression Trees).

3

- Interop often meant to use *Object* and casting all over.
- It makes sense to base scripting languages on the CLR, because it already offers some features: type loader, resolver, memory manager, garbage collector etc.
- In Java we found this development some years ago (JSR 292 and JSR 223). – Some new scripting languages have been built on top of the JRE (Groovy, Clojure and Scala) and others have been migrated (JRuby and Jython).

Leering at VB: Dynamic Programming

See accompanying project <DynamicProgrammingVB>
Shows dynamic typing, late binding, duck typing and COM automation with VB.

Key aspects of dynamic programming:
1. Dynamic Typing
2. Late Binding
3. Duck Typing
(4. Dynamic Objects)

- The feature, which we discuss here as dynamic typing, is also known as loosely typing. – The main idea is that a dynamically typed variable is just a named placeholder for a value.

Static Typing vs. dynamic Typing – Part I

- Programming languages can be divided into statically and dynamically typed ones, whereby basically:
  - Statically typed languages make type checks at compile time.
    - Expressions are of a known type at compile time, variables are fixed to one type at compile time.
  - Dynamically typed languages make type identification and checks at run time.

- Dynamically typed languages claim "convention over configuration/static check".
  - The believe is that discipline is more important than technical guidance.
  - The conventions work, because duck typing allows consistency at run time.
  - Conventions must be checked with unit-tests, a compiler can just check the syntax.
  - (Unit tests are required: they make dynamic code safe, because the compiler can't.)

- In dynamic typing the objects' types you're dealing with could be unknown.
  - The types are checked at run time and may even change at run time!
  - In static typing the objects' types are known and checked at compile time.

- Esp. dynamically typed languages do not require to declare a type at compile time.
- Example conventions: (esp. naming conventions) Ruby on Rails' scaffolding pattern and also ASP.NET MVC.
- Dynamic typing and unknown types are typically found in JavaScript (also called loosely typing in JavaScript) programming.
- The type-changes at run time are not expressed by polymorphism in dynamic typing! – The needed consistency is solely expressed by duck typing!

Static Typing vs. dynamic Typing – Part II

- Incompatible interfaces (e.g. missing methods) are handled differently.
  - Static typing uses methods that must be bound at compile time successfully.
  - Dynamic typing rather uses messages being dispatched at run time.
    - In static typing we have typing before run time, but
    - in dynamic typing an object decides, whether it accepts a message.
  - In dynamic typing messages that are not "understood" can be handled at run time.
    - Methods and fields could be added or removed during run time.
    - Possible perception: Not: "Are you of this type?", rather: "Can you respond to this message?"
- Static type-checks, rely on explicit conversions; dynamic type-checks are deferred to run time and explicit conversions are required.
  - This is called late binding.
  - Late binding resolves textual symbols at run time (e.g. property names).
  - Early binding means to resolve all textual symbols early at compile time.
  - Static polymorphism is class-based polymorphism with static interfaces and dynamic dispatch.
  - Dynamic polymorphism means that objects don't need a common static interface like classes as we need no dynamic dispatch.

- Missing methods:
  - Mind the problem of unavailable Services in a SOA.
  - In Ruby and Python methods can be removed during run time.
  - If a method wasn't found in Obj-C, the method invocation can be forwarded and handled at run time. If the case of unknown methods was not handled by the programmer *doesNotRecognizeSelector*: will be called, which will eventually throw *NSInvalidArgumentException* ("unrecognized selector sent to instance <address>").
  - In Smalltalk *doesNotUnderstand*: will be called, which, if not handled by the programmer, will be handled by the runtime (or the debugger).

So eventually: Dynamic Programming in .NET 4 and C#4

- The .NET framework and C# are based on static typing, nothing changed here.
  - I.e. types and their interfaces are resolved/bound at compile time.
- C#4 allows dynamic typing in a statically typed language.
  - C# stays a statically typed language primarily, but it has new dynamic features now.
- Dynamic type analysis can be done via reflection, but it's difficult to read.
  - Dynamic typing makes the syntax much more appealing, even natural.
- The .NET 4 framework introduces the Dynamic Language Runtime (DLR).
  - The DLR is a library that enables dynamic dispatching within a CLR language.
- The new C#4 keyword dynamic is the syntactical entrance into the DLR.
  - If you apply dynamic you tell the compiler to turn off type-checks.

- In Obj-C you can use the static type id to enable dynamic typing on objects.
- The COM type *IDispatch* is indeed also an enabler for dynamic dispatching/late binding, when scripting and COM automation come into play.
- For a long time VB was the only language, in which you can switch between early bound (static) and lately bound (dynamic) typing (Option Explicit On/Off, Option Strict On/Off). In past versions of VB, e.g. in VB 6 this ability was based on the VB- and COM-type *VARIANT*.
  - But *VARIANT* is best explained to be an example of weak typing rather than of dynamic typing.

Syntactical Basics of dynamic Coding in C#4

See accompanying project <BasicSyntax, Dispatching>
Shows the syntax basics of dynamics, dynamic expressions in C# and dynamic dispatching.

Syntactical "Mechanics" of dynamic in C#4 – Part I

- The static type of dynamic objects is dynamic. Variables still need a declaration.
  - For the "programming experience" dynamic is just a type (you can have dynamic members and closed generic types using dynamic (e.g. *List<dynamic>*)).
- Almost each type is implicitly convertible to dynamic.
- But dynamic expressions are not convertible to anything, except in:
  - overload resolution and assignment conversion
  - and in control flow: if clauses, using clauses and foreach loops.
- Dynamic Expressions use values of type dynamic. They're evaluated at run time.
- From a syntax perspective dynamic works like object w/o the need to cast.
  - You can write direct calls to the methods you expect to be present (duck typing).

- The keyword dynamic can be understood as "*System.Object* with run time binding". At compile time, dynamic variables have only those methods defined in *System.Object*, at run time possibly more.
- But dynamic can be used as type argument for a generic type as base class (not a generic interface).

Syntactical "Mechanics" of dynamic in C#4 – Part II

- The type dynamic isn't a real CLR type, but C# mimics this for us.
- The type dynamic is just *System.Object* with run time binding.
  - For non-local instances of dynamic the compiler applies the custom attribute *System.Runtime.CompilerServices.DynamicAttribute*.
    - (Fields, properties and method's and delegate's parameters and return types.)
- Restrictions:
  - The type dynamic can not be used as base class and can not be extended.
  - The type dynamic can not be used as type constraint for generics.
  - You can't get the typeof(dynamic).
  - You can neither access constructors nor static members via dynamic dispatch.
  - You can only call accessible methods via dynamic dispatch.
  - Dynamic dispatch can't find extension methods.
  - Dynamic dispatch can't find methods of explicitly implemented interfaces.

10

- If you need to do more with a dynamic object than, say, assigning values to it, e.g. calling methods, the DLR starts its work to dispatch the resulting dynamic expressions. In C#, the C# language binder will come into play in that situation, and if you apply dynamic dispatch in your code, you will need to add a reference to the assembly Microsoft.CSharp.dll in VS 2010. – The good news is that this assembly will be referenced by default, when you setup a new C# project in VS2010.
- Ctors and static members can't be dynamically dispatched! You have to stick to instance methods or you have to use polymorphic static typing and factories.

Thank you!