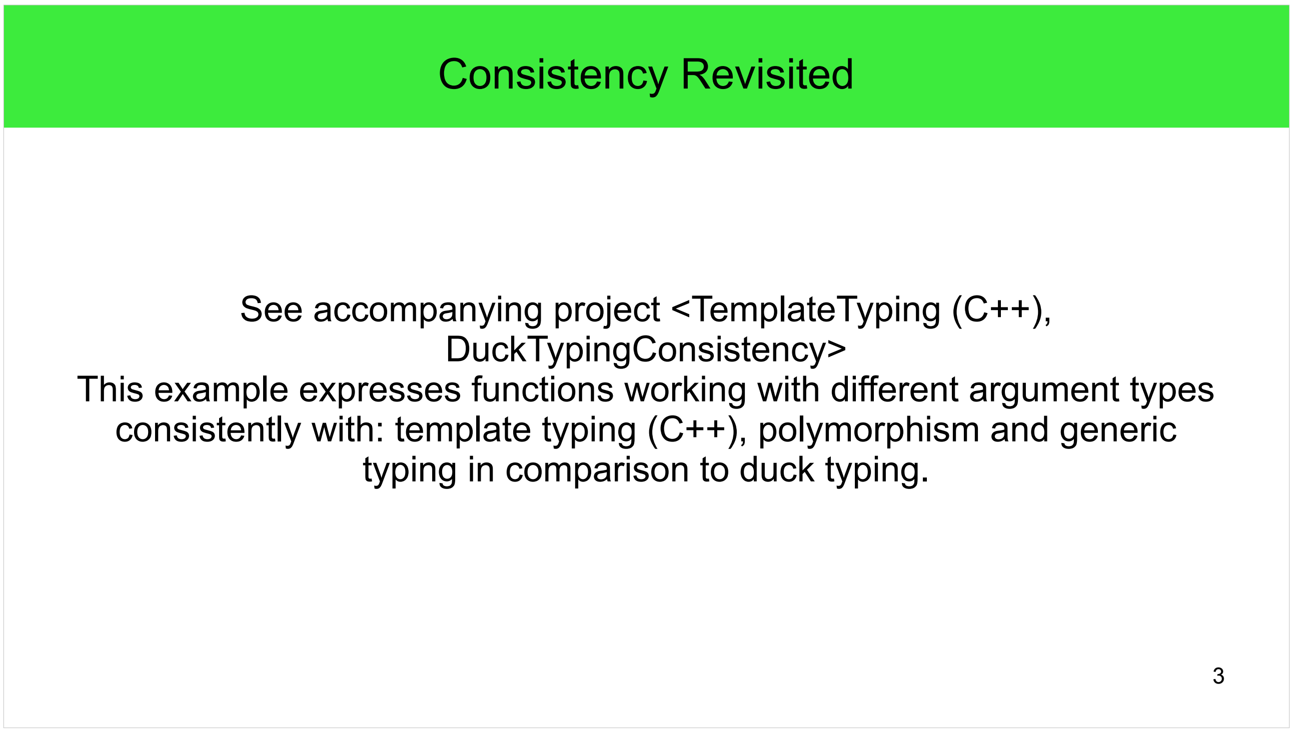
New C#4 Features – Part IV

Nico Ludwig (@ersatzteilchen)

TOC

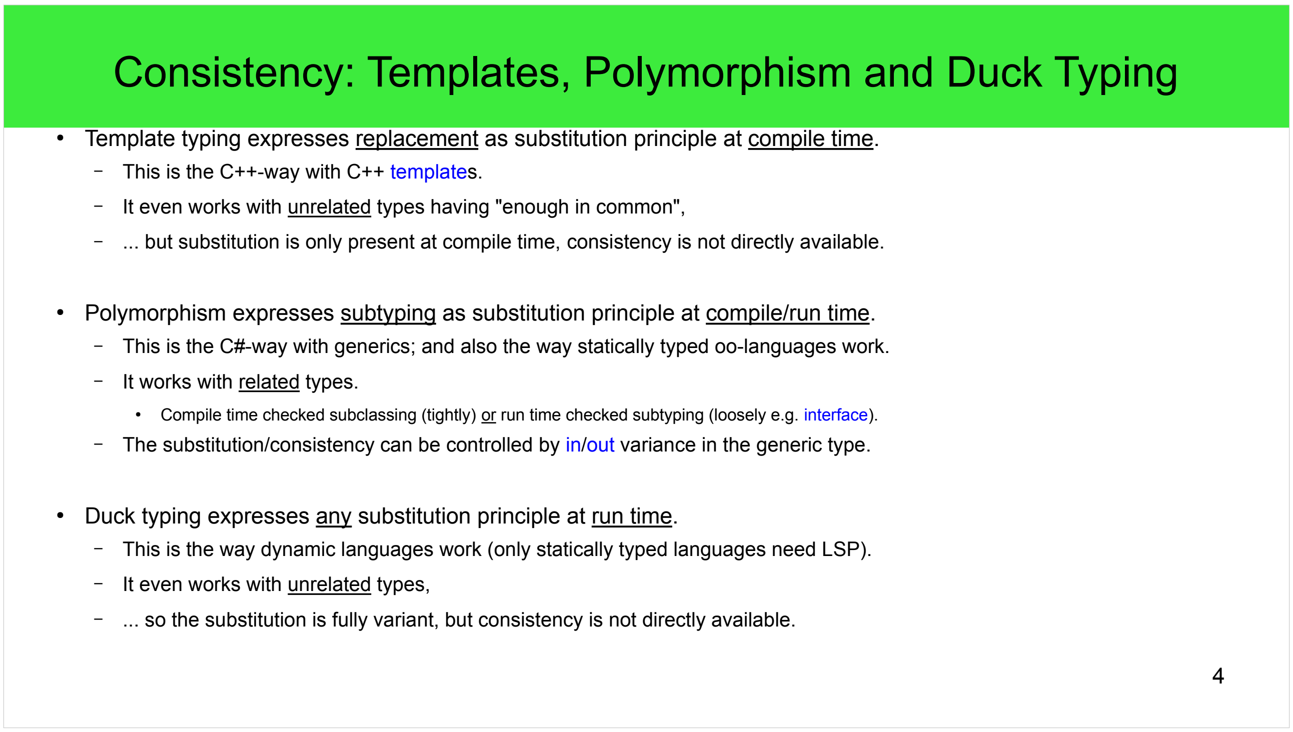
- Sources:
  - Jon Skeet, C# in Depth
  - Chaur Wu, Pro DLR in .NET 4



Consistency Revisited

See accompanying project <TemplateTyping (C++), DuckTypingConsistency>
This example expresses functions working with different argument types consistently with: template typing (C++), polymorphism and generic typing in comparison to duck typing.

- The individual examples show different ways to program consistent functions with different expressions of the substitution principle.

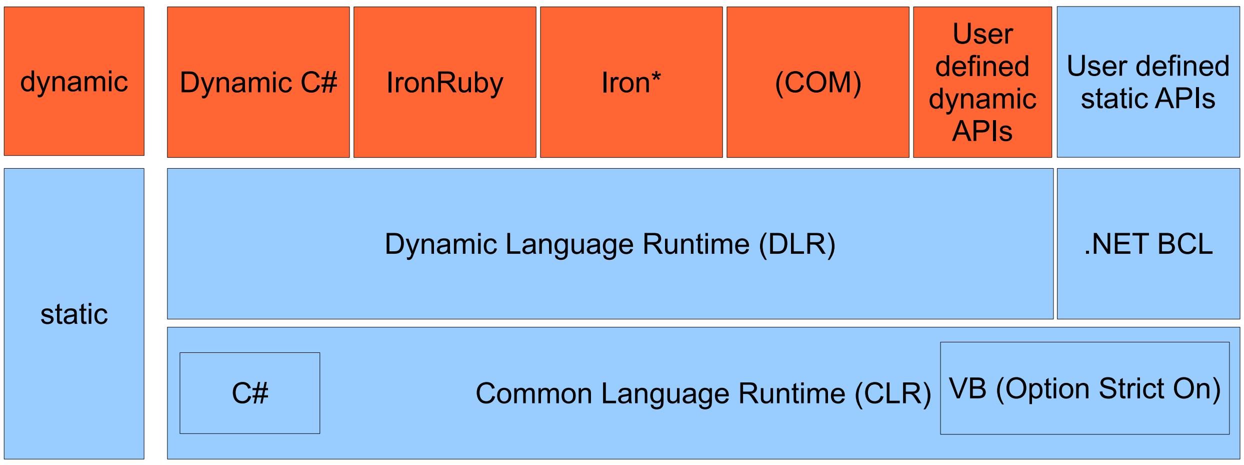


Consistency: Templates, Polymorphism and Duck Typing

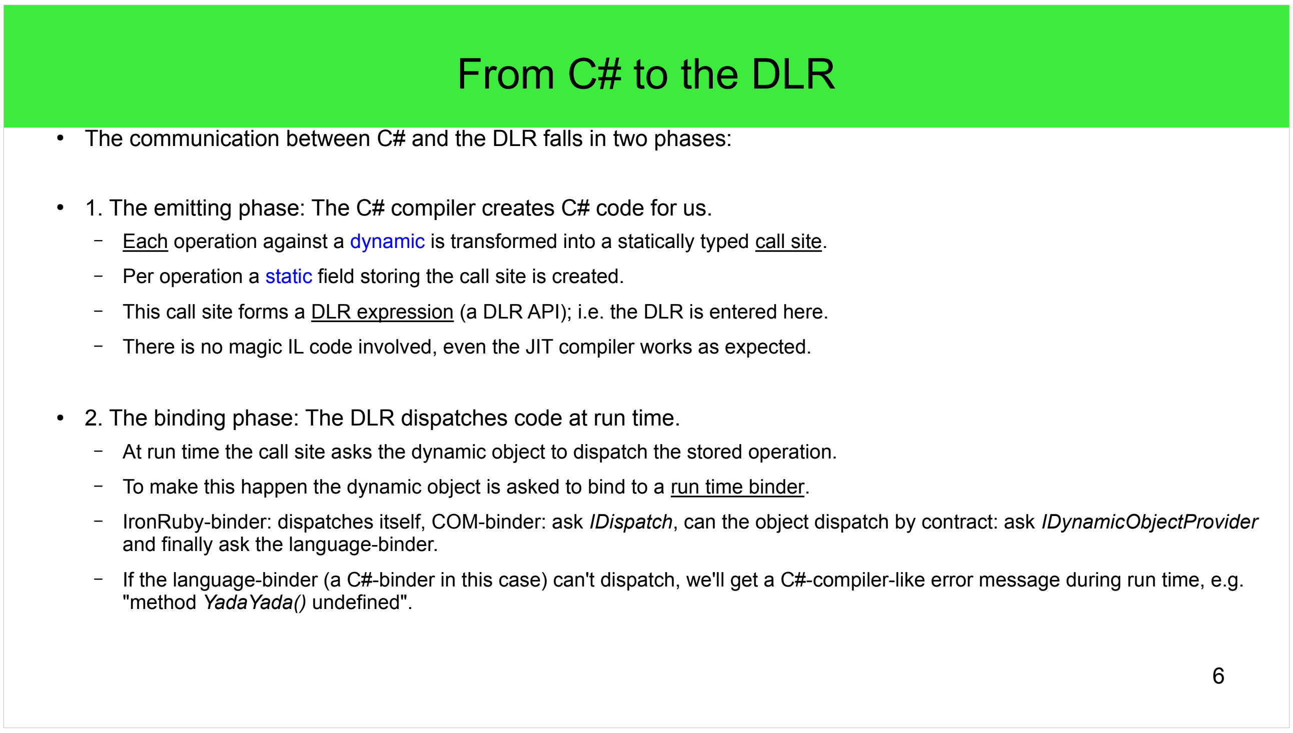
- Template typing expresses replacement as substitution principle at compile time.
  - This is the C++-way with C++ templates.
  - It even works with unrelated types having "enough in common",
  - ... but substitution is only present at compile time, consistency is not directly available.
- Polymorphism expresses subtyping as substitution principle at compile/run time.
  - This is the C#-way with generics; and also the way statically typed oo-languages work.
  - It works with related types.
    - Compile time checked subclassing (tightly) or run time checked subtyping (loosely e.g. interface).
  - The substitution/consistency can be controlled by in/out variance in the generic type.
- Duck typing expresses any substitution principle at run time.
  - This is the way dynamic languages work (only statically typed languages need LSP).
  - It even works with unrelated types,
  - ... so the substitution is fully variant, but consistency is not directly available.

- These are the ways to express interoperability: common interfaces or dynamics.
- C++ may use static asserts to express the traits of a type parameter. Also does the idea of concepts in a future version of C++.
- We left out the consistency got with generic methods, lambda expressions and type inference here, but it is still a mighty tool we should understand (esp. to get a better understanding of how LINQ works)!
- Once again: duck typing means that we assume e.g. methods to be present as a "convention".
- With interfaces we can more easily fool the compiler, because it accepts interface-casts ("cast-contact lenses") almost every time (but e.g. not on instances of sealed classes). Casts between classes undergo more checks at compile time!
- LSP in this case means that argument types must be contravariant and return types must be covariant in statically typed languages. – C# uses statically checked definition-site variance.
- In dynamic typing the contributing objects "in" the variables do really change; no polymorphism is applied!
- It is remarkable that developers were first told to code compile time safe (generics) and with dynamics we let the types only be checked at run time...

DLR as unified Subsystem for dynamic Coding with the CLR



- The DLR is not a new run time, it is a regular piece of code (library) contained in System.Core.dll in .NET 4, but it was developed as open source (on Codeplex with Apache v.2 license). There exist also variants of the DLR (and IronRuby and IronPython, both Apache v.2 license) that run on .NET 2 (then we'll need to deploy the DLR (Microsoft.Scripting.dll, Microsoft.Dynamic.dll etc.) ourselves).
  - DLR is an extension of the CLR that unifies dynamic programming on top of .NET.
  - As can be seen the primary goal of DLR is interop, because it allows bridging between the dynamic world and the CLR.
  - DLR brings a shared notion of dynamic dispatch among languages as CLR did for static dispatch.
- Support for the DLR has been added into VB's and C#'s core libraries. – The binders then allow different dynamic receivers.
  - The .NET C# runtime binder is contained in the assembly Microsoft.CSharp.dll.
- DLR unification means also that we can use .NET or COM from within other languages using the DLR (e.g. in Iron*).
- IRON: (kidding) Implementation Running on .NET
- The DLR and IronPython have been developed by the same person, Jim Hugunin. He designed the DLR to make dynamic programming possible to create dynamic languages like IronPython on top of .NET.

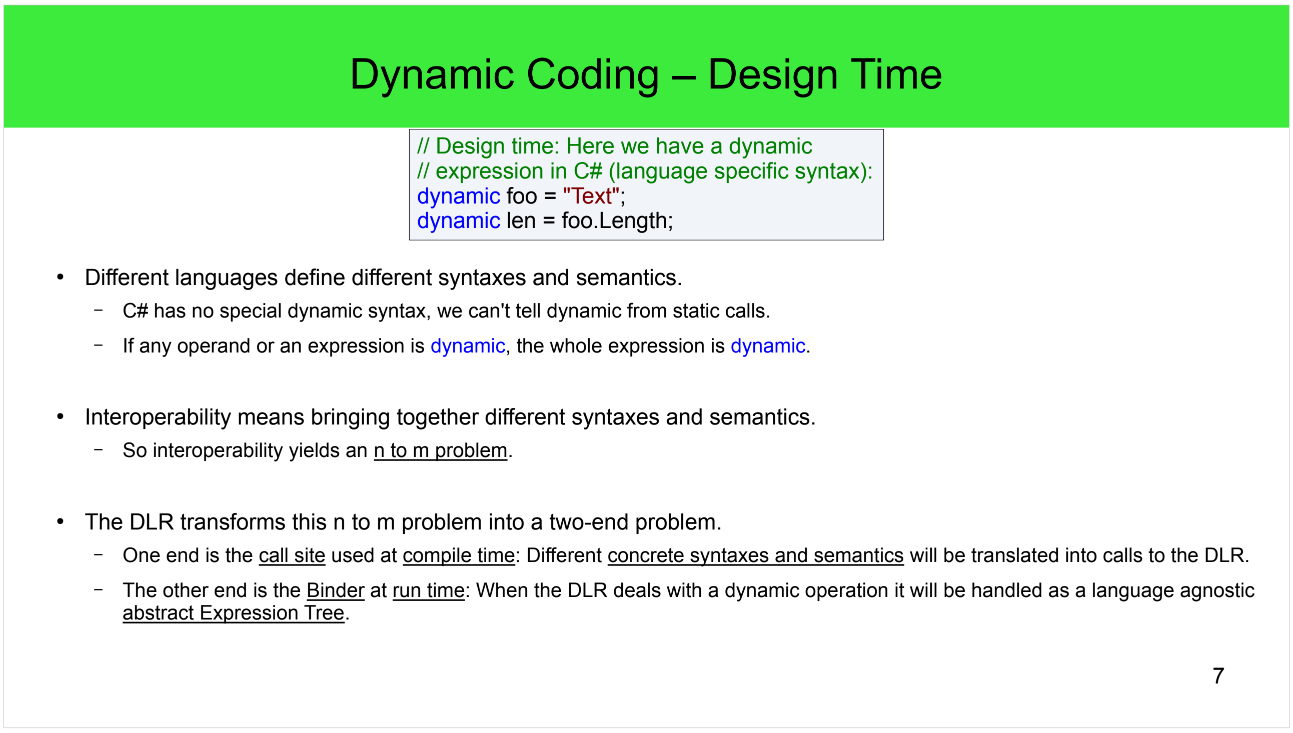


From C# to the DLR

- The communication between C# and the DLR falls in two phases:
- 1. The emitting phase: The C# compiler creates C# code for us.
  - Each operation against a dynamic is transformed into a statically typed call site.
  - Per operation a static field storing the call site is created.
  - This call site forms a DLR expression (a DLR API); i.e. the DLR is entered here.
  - There is no magic IL code involved, even the JIT compiler works as expected.
- 2. The binding phase: The DLR dispatches code at run time.
  - At run time the call site asks the dynamic object to dispatch the stored operation.
  - To make this happen the dynamic object is asked to bind to a run time binder.
  - IronRuby-binder: dispatches itself, COM-binder: ask *IDispatch*, can the object dispatch by contract: ask *IDynamicObjectProvider* and finally ask the language-binder.
  - If the language-binder (a C#-binder in this case) can't dispatch, we'll get a C#-compiler-like error message during run time, e.g. "method *YadaYada()* undefined".

6

- The DLR is able to express each dynamic operation (e.g. "GetMember" etc.; twelve operations that can be late bound (making up the "CTS" of the DLR) as common denominator among all dyn. langs.) on an object type syntax-independently as DLR expression but as statically typed code. Interestingly some operations "CreateInstance" (i.e. ctors) or "DeleteMember" can't be bound dynamically in C#.
- The explicit by-contract-binding is supported by *IDynamicObjectProvider*, we'll discuss this in another lesson (i.e. how to to define our own implementation of dynamic operations).
- The C# run time binder works with reflection. This binder is very complex, because of C#'s flexible syntax (overloads, conversions, unsafe contexts, and ambiguity (*Index()* -> is this a method call or a property of type *Action* that was immediately called?)).
- Actually the C# run time binder is a part of the C# compiler yanked out to the run time, therefor we'll get the same errors on run time (as Exceptions) we'd get on compile time with static typing (e.g. inaccessible due to the access modifier or conversion not supported).

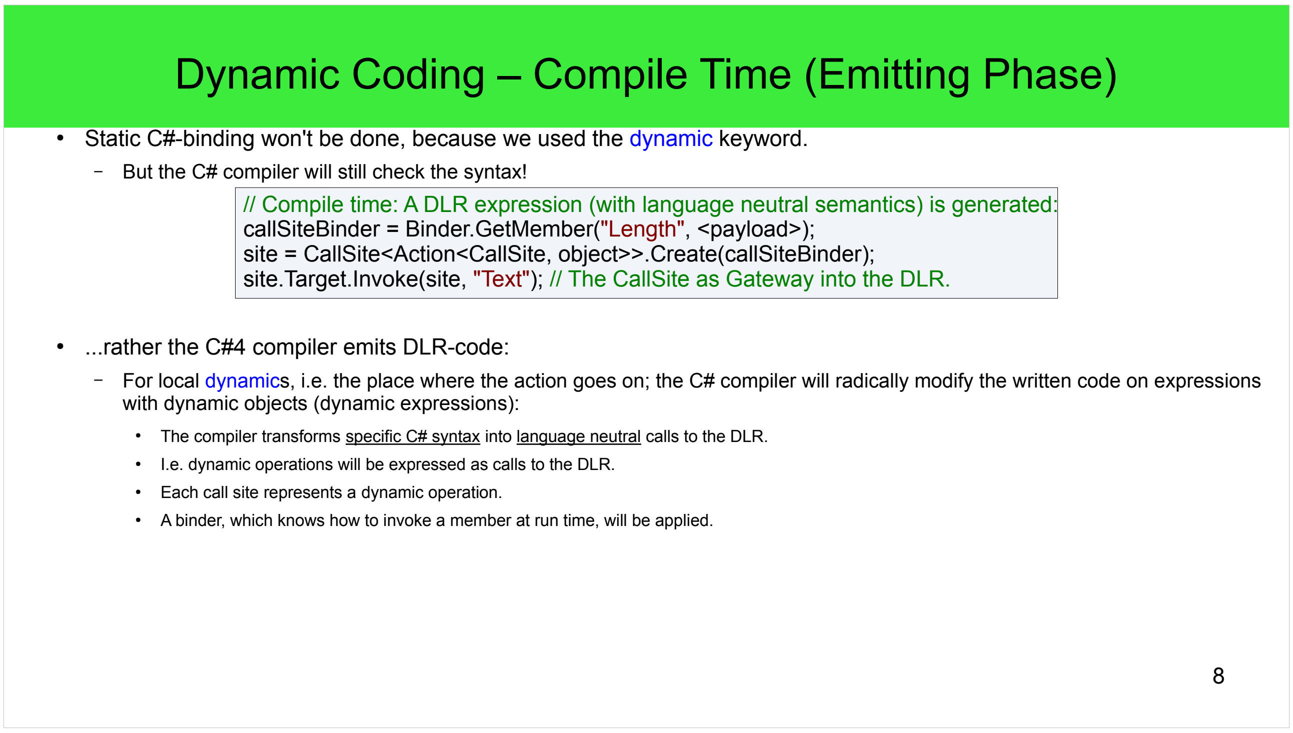


Dynamic Coding – Design Time

```
// Design time: Here we have a dynamic
// expression in C# (language specific syntax):
dynamic foo = "Text";
dynamic len = foo.Length;
```

- Different languages define different syntaxes and semantics.
  - C# has no special dynamic syntax, we can't tell dynamic from static calls.
  - If any operand or an expression is dynamic, the whole expression is dynamic.
- Interoperability means bringing together different syntaxes and semantics.
  - So interoperability yields an n to m problem.
- The DLR transforms this n to m problem into a two-end problem.
  - One end is the call site used at compile time: Different concrete syntaxes and semantics will be translated into calls to the DLR.
  - The other end is the Binder at run time: When the DLR deals with a dynamic operation it will be handled as a language agnostic abstract Expression Tree.

- We could use reflection explicitly to do it, but then we'll lose C#'s semantics.
- Expression Trees are able to hold the original C# code (concrete syntax) as a language neutral piece of data (i.e. the abstract syntax). We'd get the same Expression Tree with an equivalent dynamic VB-expression (another concrete syntax). – Expression Trees can also be cached.
- The DLR then executes the Expression Trees.



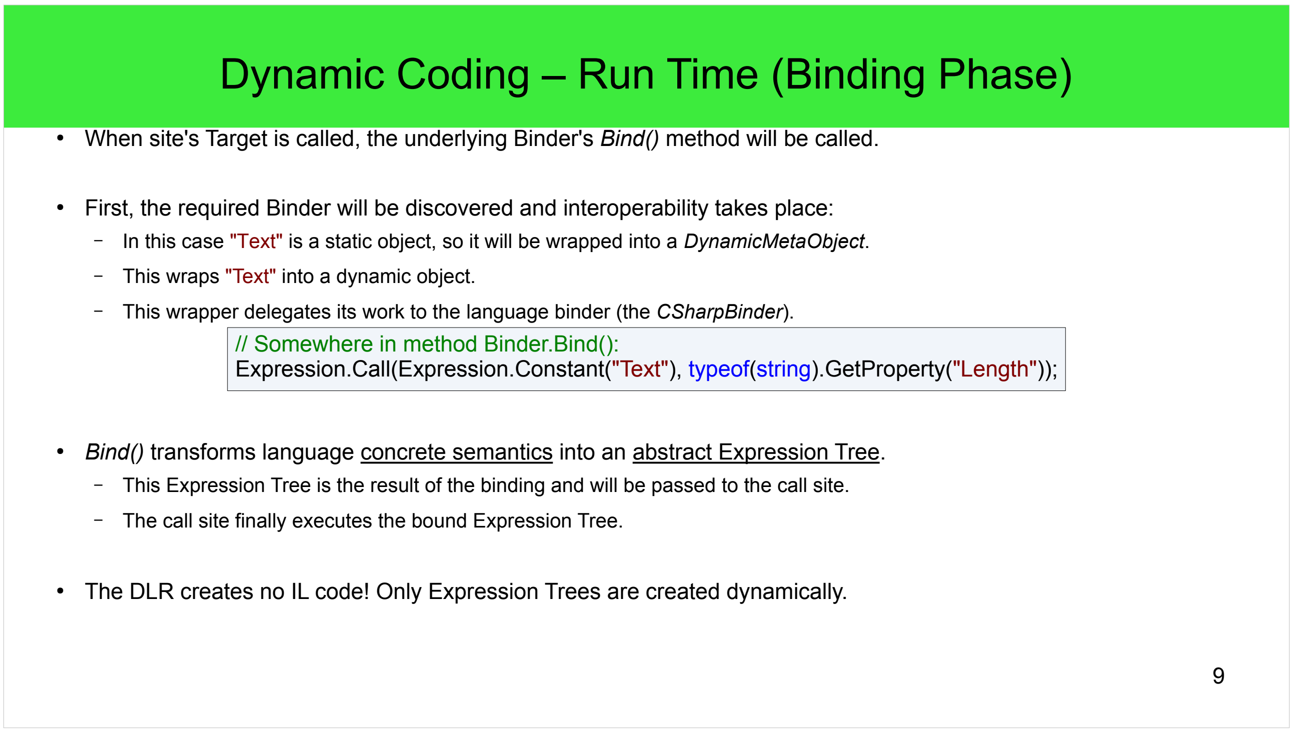
Dynamic Coding – Compile Time (Emitting Phase)

- Static C#-binding won't be done, because we used the dynamic keyword.
  - But the C# compiler will still check the syntax!

```
// Compile time: A DLR expression (with language neutral semantics) is generated:
callSiteBinder = Binder.GetMember("Length", <payload>);
site = CallSite<Action<CallSite, object>>.Create(callSiteBinder);
site.Target.Invoke(site, "Text"); // The CallSite as Gateway into the DLR.
```

- ...rather the C#4 compiler emits DLR-code:
  - For local dynamics, i.e. the place where the action goes on; the C# compiler will radically modify the written code on expressions with dynamic objects (dynamic expressions):
    - The compiler transforms specific C# syntax into language neutral calls to the DLR.
    - I.e. dynamic operations will be expressed as calls to the DLR.
    - Each call site represents a dynamic operation.
    - A binder, which knows how to invoke a member at run time, will be applied.

- In principle the C# compiler doesn't understand the code, it just compiles it...
- Besides e.g. the method name the payload carries the arguments, known compile time types and other information.
- It is needed to abstract an operation to a dedicated site, because even if a specific operation is called multiple times, it may not behave the same each time (e.g. polymorphism or duck typing). – Here the site's cache comes into play.



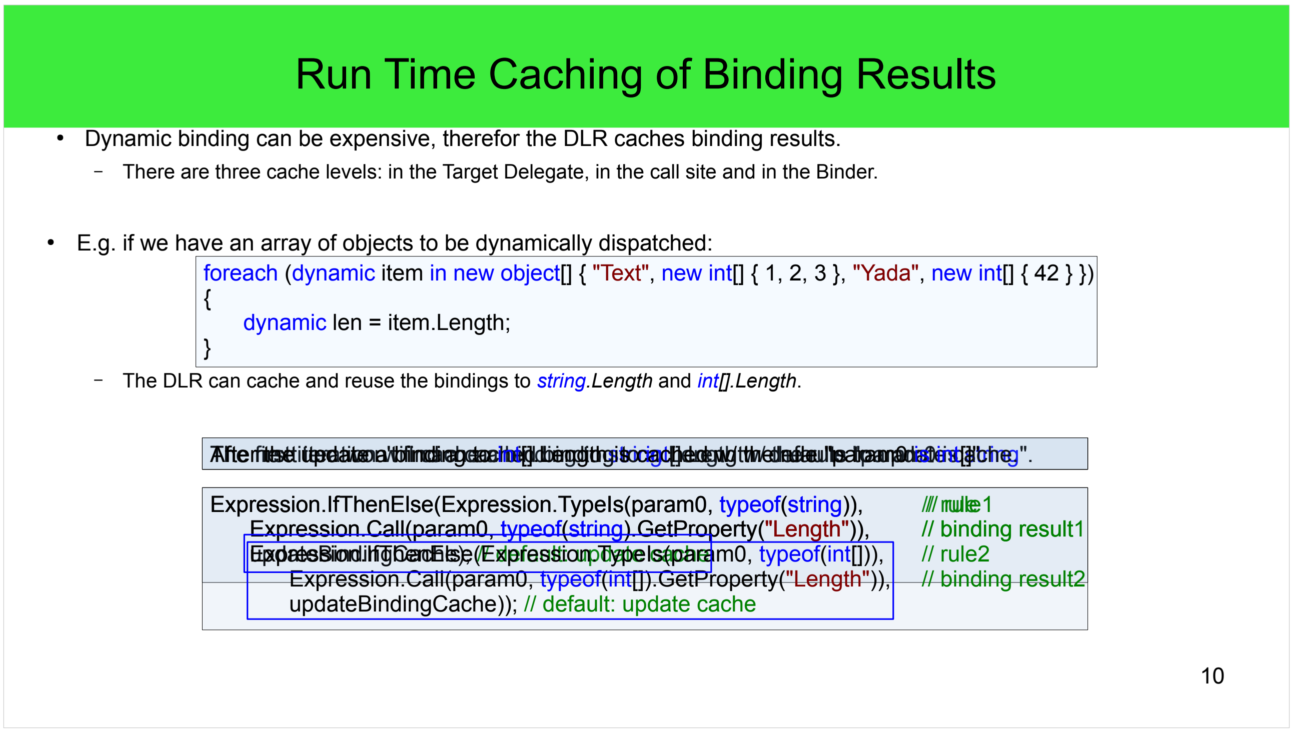
Dynamic Coding – Run Time (Binding Phase)

- When site's Target is called, the underlying Binder's *Bind()* method will be called.
- First, the required Binder will be discovered and interoperability takes place:
  - In this case "Text" is a static object, so it will be wrapped into a *DynamicMetaObject*.
  - This wraps "Text" into a dynamic object.
  - This wrapper delegates its work to the language binder (the *CSharpBinder*).

```
// Somewhere in method Binder.Bind():
Expression.Call(Expression.Constant("Text"), typeof(string).GetProperty("Length"));
```

- *Bind()* transforms language concrete semantics into an abstract Expression Tree.
  - This Expression Tree is the result of the binding and will be passed to the call site.
  - The call site finally executes the bound Expression Tree.
- The DLR creates no IL code! Only Expression Trees are created dynamically.

- Discovery of binders and interoperability: The late-binding logic resides in binders as well as in dynamic objects, because late-binding logic may pertain to a language or to an object.
- In VS 2010 we can inspect a language neutral textual representation of a complex Expression Tree in the debugger. Therefor objects of type Expression show an extra pseudo property "DebugView" in the debugger's watches.



Run Time Caching of Binding Results

- Dynamic binding can be expensive, therefor the DLR caches binding results.
  - There are three cache levels: in the Target Delegate, in the call site and in the Binder.

- E.g. if we have an array of objects to be dynamically dispatched:

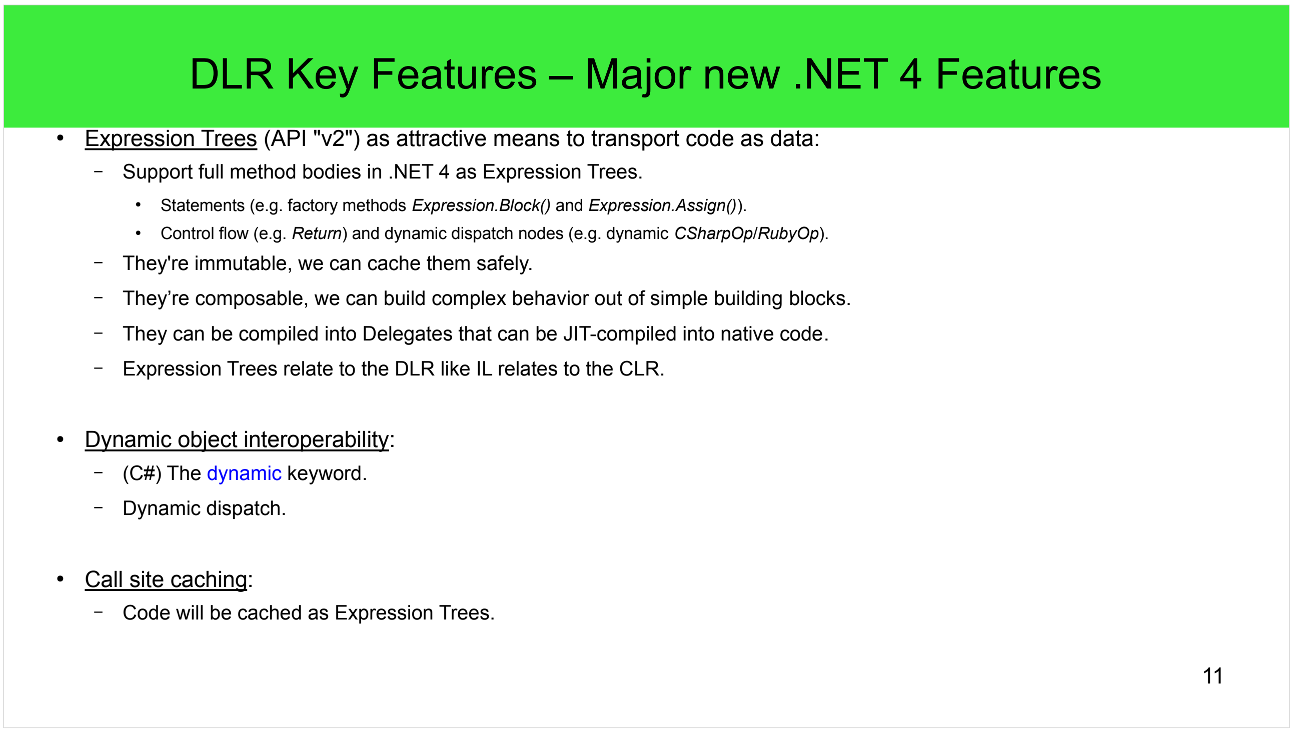
```
foreach (dynamic item in new object[] { "Text", new int[] { 1, 2, 3 }, "Yada", new int[] { 42 } })
{
    dynamic len = item.Length;
}
```

  - The DLR can cache and reuse the bindings to *string.Length* and *int[].Length*.

[illegible]

```
Expression.IfThenElse(Expression.TypeIs(param0, typeof(string)),        // rule1
    Expression.Call(param0, typeof(string).GetProperty("Length")),      // binding result1
    Expression.IfThenElse(Expression.TypeIs(param0, typeof(int[])),     // rule2
        Expression.Call(param0, typeof(int[]).GetProperty("Length")),   // binding result2
        updateBindingCache)); // default: update cache
```

- This is only pseudo code, it doesn't run. – Much more context is required (e.g. site and binder).
- Expression Trees can not be modified, rather it is required to create new ones by recycling existing subtrees.
- The DLR does not create code during run time, but creates Expression Trees that can be cached. I.e. we won't see such "written" Expression Tree code somewhere.
- This an example of polymorphic inline caching (also used in JavaScript and Smalltalk).



DLR Key Features – Major new .NET 4 Features

- Expression Trees (API "v2") as attractive means to transport code as data:
  - Support full method bodies in .NET 4 as Expression Trees.
    - Statements (e.g. factory methods *Expression.Block()* and *Expression.Assign()*).
    - Control flow (e.g. *Return*) and dynamic dispatch nodes (e.g. dynamic *CSharpOp*/*RubyOp*).
  - They're immutable, we can cache them safely.
  - They’re composable, we can build complex behavior out of simple building blocks.
  - They can be compiled into Delegates that can be JIT-compiled into native code.
  - Expression Trees relate to the DLR like IL relates to the CLR.
- Dynamic object interoperability:
  - (C#) The dynamic keyword.
  - Dynamic dispatch.
- Call site caching:
  - Code will be cached as Expression Trees.

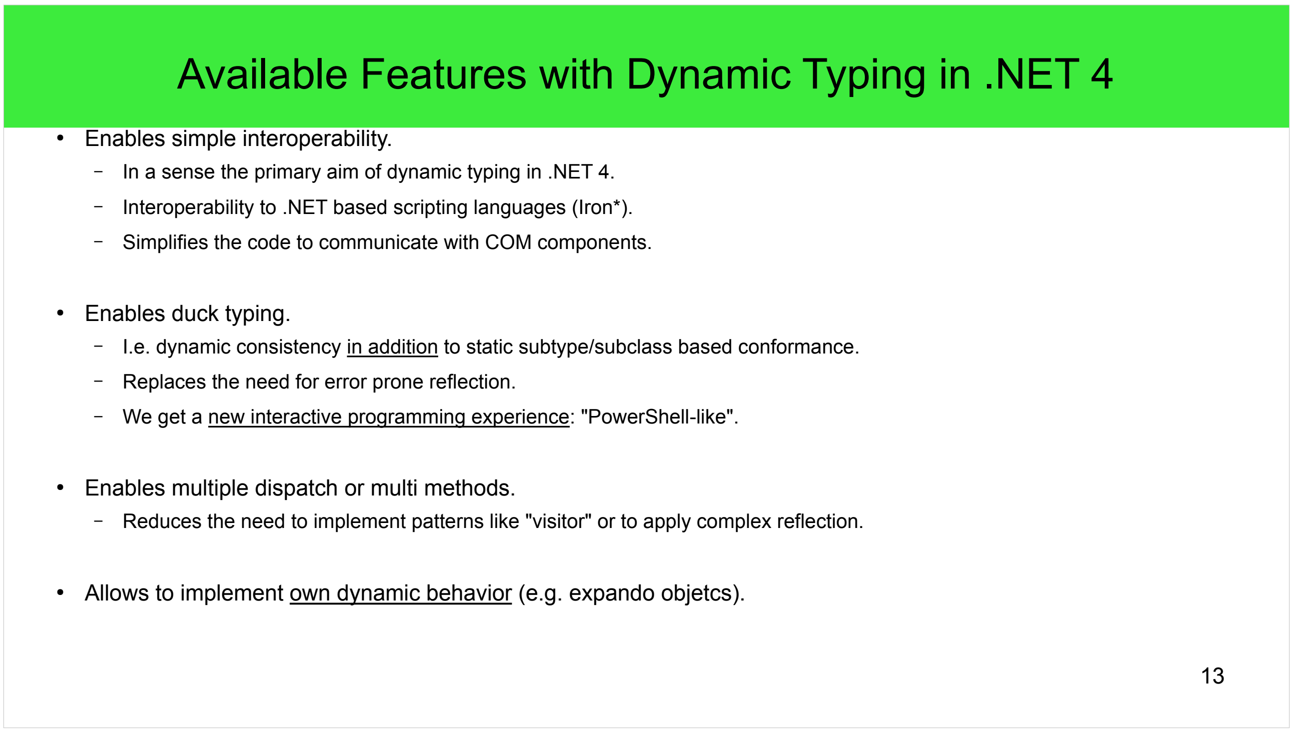
- At the time of writing (December 2011) 85 different node types (*Add*, *Or*, *Invoke* etc.) are supported for Expression Trees.
- The new ability we got with dynamic object interoperability in C# is that we can handle objects of unrelated types in a common way, if they have "enough in common" (e.g. compatible methods (name and signatures)). – This allows to write much simpler algorithms w/o the need to use reflection.

Summary of the DLR

- The DLR allows interoperability of C# with dynamic environments (incl. C# itself):
  - The compiler understands code that looks like C#, but bridges to other locations.
  - So dynamic in C# has a broad meaning, it allows to code everything dynamically typed, .NET and beyond.

- This interoperability is given by run time binders:
  - within C# itself; to allow dynamic coding,
  - with other .NET languages, being dynamically typed, like IronRuby or
  - with other (dynamic) environments, different from .NET (like COM).

- DLR's intend is not to encourage dynamic programming in C# generally!
  - It provides a transition between static typing (C#) and dynamically typed environments.

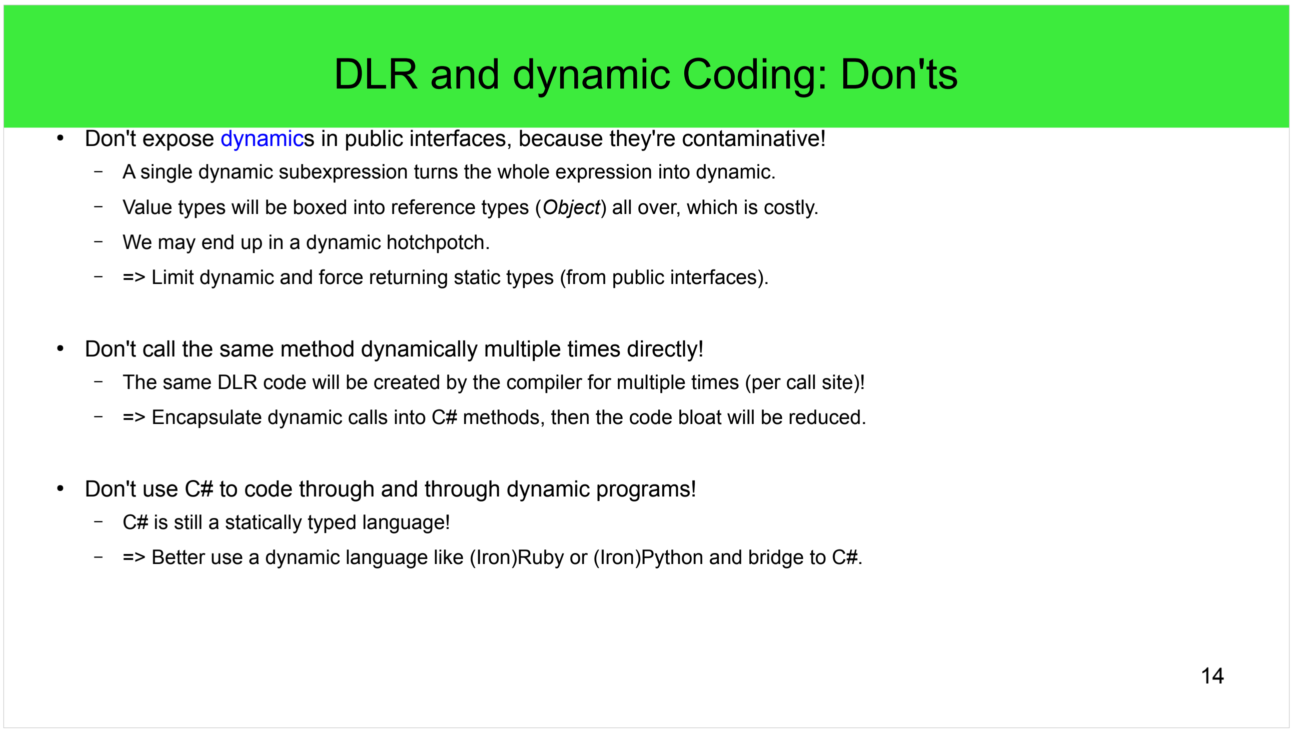


Available Features with Dynamic Typing in .NET 4

- Enables simple interoperability.
  - In a sense the primary aim of dynamic typing in .NET 4.
  - Interoperability to .NET based scripting languages (Iron*).
  - Simplifies the code to communicate with COM components.
- Enables duck typing.
  - I.e. dynamic consistency in addition to static subtype/subclass based conformance.
  - Replaces the need for error prone reflection.
  - We get a new interactive programming experience: "PowerShell-like".
- Enables multiple dispatch or multi methods.
  - Reduces the need to implement patterns like "visitor" or to apply complex reflection.
- Allows to implement own dynamic behavior (e.g. expando objetcs).

13

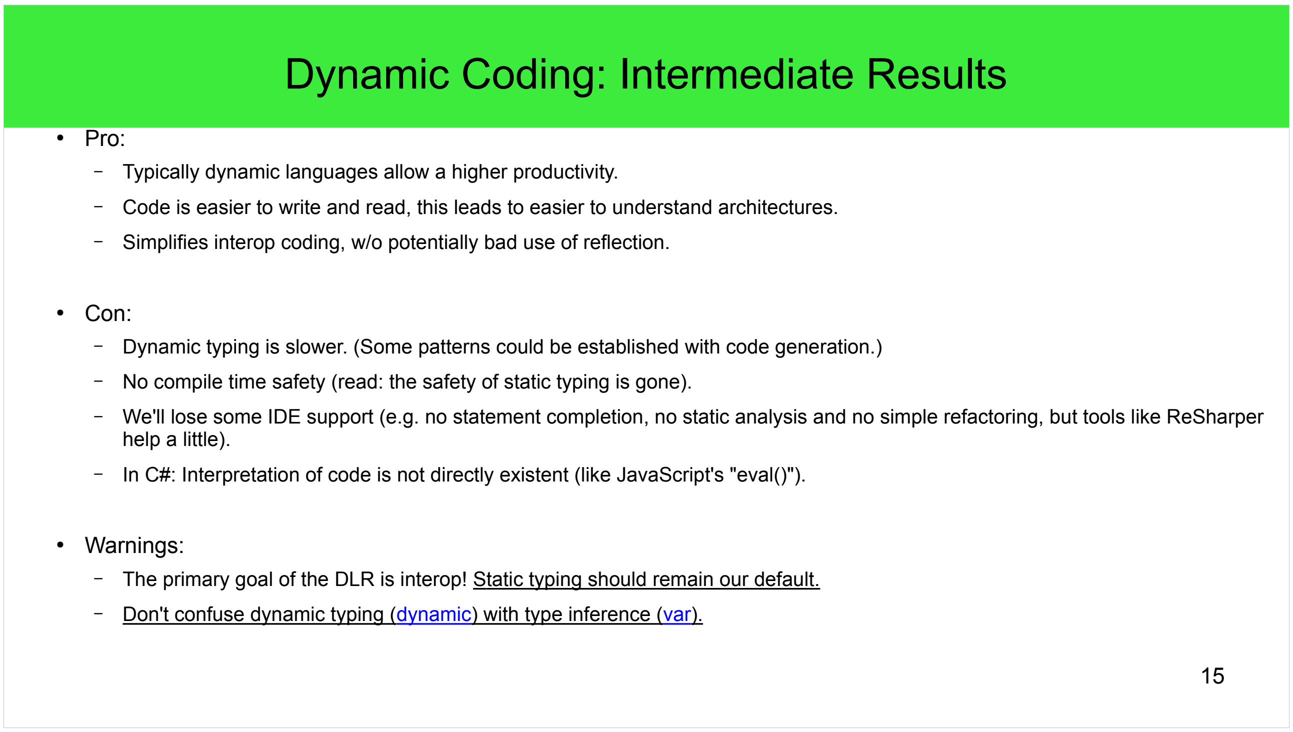
- Often we won't find patterns and Aspect Oriented Programming (AOP) being discussed in dynamic programming languages. – These concepts are mostly applied in statically typed languages to simulate abilities of dynamically typed languages...
- Own dynamic behavior means that we can create types, whose interfaces changes during run time; and these interfaces-changes are based on how we use the type.



DLR and dynamic Coding: Don'ts

- Don't expose dynamics in public interfaces, because they're contaminative!
  - A single dynamic subexpression turns the whole expression into dynamic.
  - Value types will be boxed into reference types (*Object*) all over, which is costly.
  - We may end up in a dynamic hotchpotch.
  - => Limit dynamic and force returning static types (from public interfaces).
- Don't call the same method dynamically multiple times directly!
  - The same DLR code will be created by the compiler for multiple times (per call site)!
  - => Encapsulate dynamic calls into C# methods, then the code bloat will be reduced.
- Don't use C# to code through and through dynamic programs!
  - C# is still a statically typed language!
  - => Better use a dynamic language like (Iron)Ruby or (Iron)Python and bridge to C#.

- Now having a fairly good overview over the abilities the DLR gives to us, we can discuss a resume of don'ts.



Dynamic Coding: Intermediate Results

- Pro:
  - Typically dynamic languages allow a higher productivity.
  - Code is easier to write and read, this leads to easier to understand architectures.
  - Simplifies interop coding, w/o potentially bad use of reflection.
- Con:
  - Dynamic typing is slower. (Some patterns could be established with code generation.)
  - No compile time safety (read: the safety of static typing is gone).
  - We'll lose some IDE support (e.g. no statement completion, no static analysis and no simple refactoring, but tools like ReSharper help a little).
  - In C#: Interpretation of code is not directly existent (like JavaScript's "eval()").
- Warnings:
  - The primary goal of the DLR is interop! Static typing should remain our default.
  - Don't confuse dynamic typing (dynamic) with type inference (var).

- If we're working w/ reflection we have at least similar speed problems, but the DLR caches binding results (the binding caches live as long as the surrounding AppDomain lives). The added run time cost is exactly the cost saved by the compiler not doing type checks.
- Developers fear loosing stability (there're no static type checks). This fear is destructed by test driven development. – This is the way dynamic programming works...
- Tools like Resharper help to improve the IDE support for dynamic typing.

Thank you!