Working with the CIL

Curtis Schlak

@realistschuckle

(twitter, github, codeplex)

curtissimo, llc

Welcome to Houston TechFest

Thank you for being a part of the 7th Annual Houston TechFest!

- Please turn off all electronic devices or set them to vibrate.
- If you must take a phone call, please do so in the lobby so as not to disturb others.
- Thanks to our Diamond Sponsors:

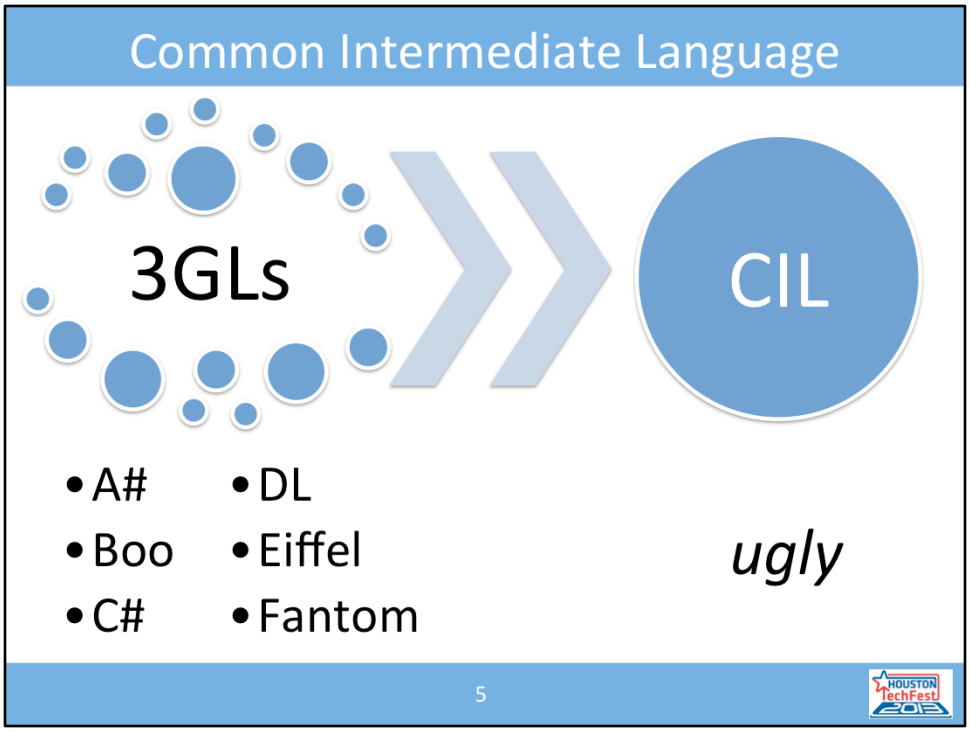
1

Information

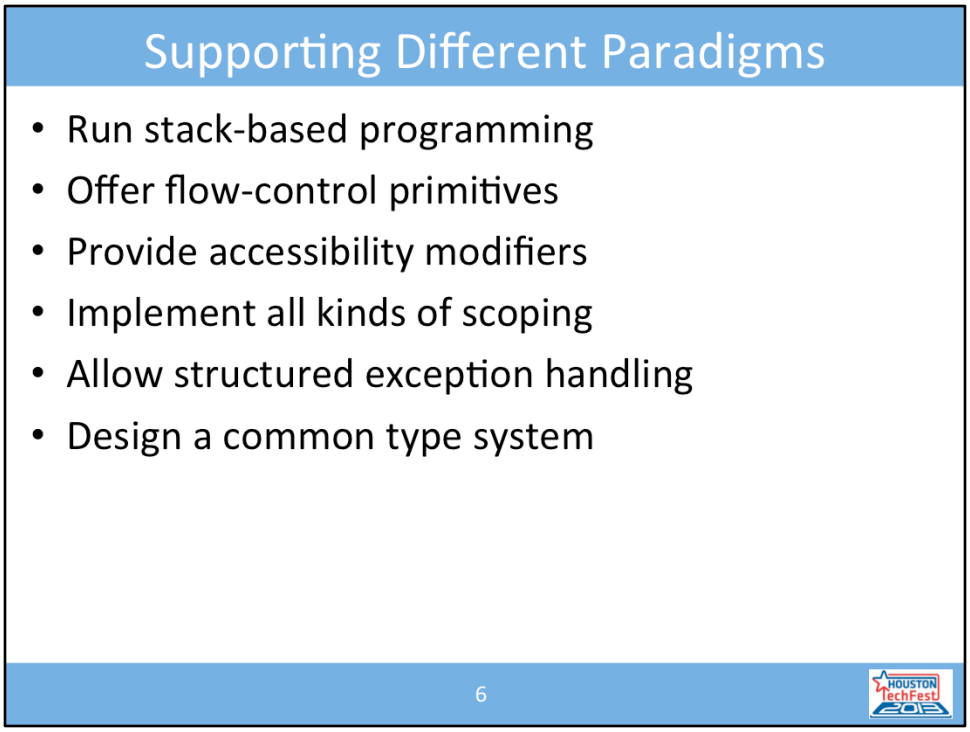
- Speaker presentation slides will be available at www.houstontechfest.org within a week
- Don't forget to complete the Bingo card to be eligible for door prizes

9:30	Meet the CIL (7.5 mins)
9:37.5	Some instructions (7.5 mins)
9:45	How C# compiles to IL (20 mins)
10:05	Crazy stuff with IL (15 mins)
10:20	Q&A

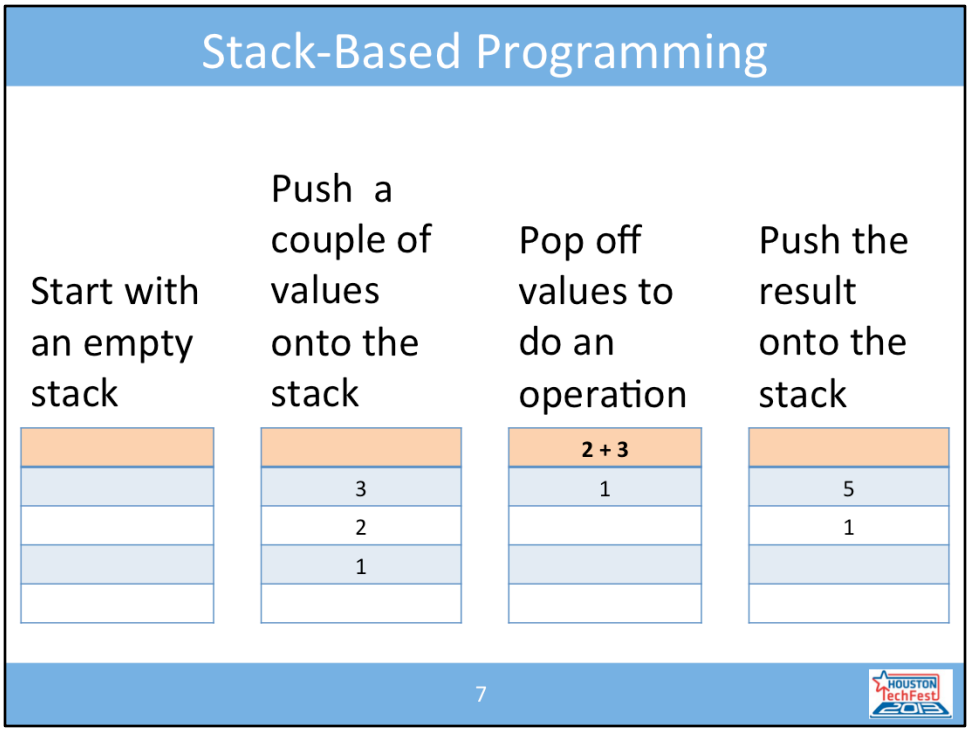
9:30	Meet the CIL (7.5 mins)
9:37.5	Some instructions (7.5 mins)
9:45	How C# compiles to IL (20 mins)
10:05	Crazy stuff with IL (15 mins)
10:20	Q&A



All the third- and fourth-generation languages that you can use to target the CLR get compiled, first, to the second-generation language Common Intermediate Language.



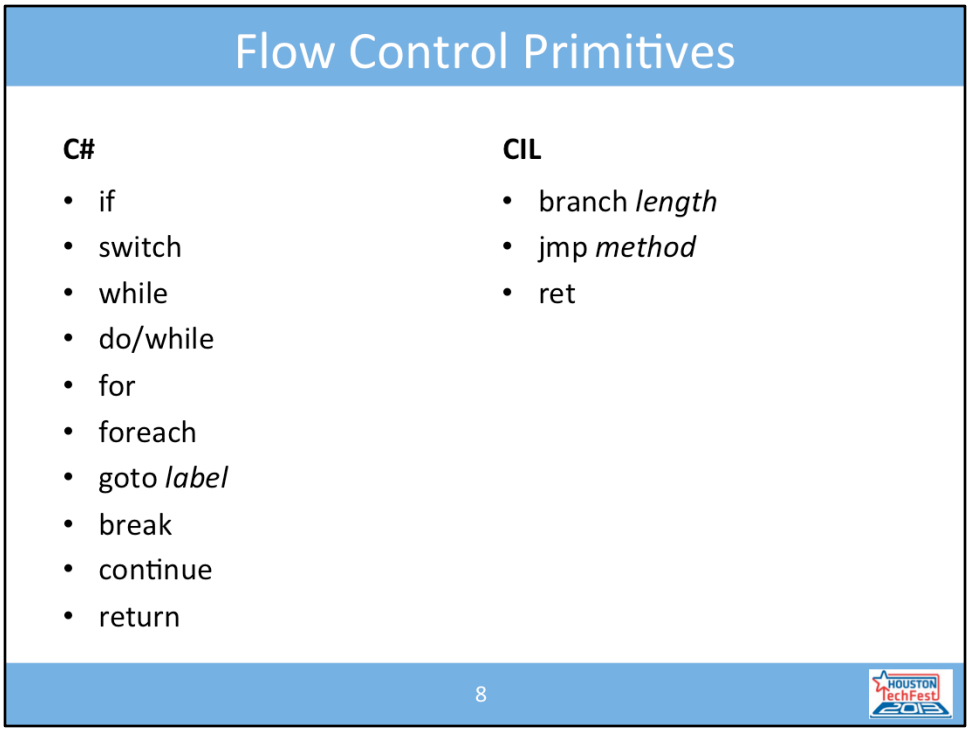
How can one language support all of the three different programming paradigms: procedural, object-oriented, functional, and the newer hybrid languages that combine features of all three? The CIL does it by addressing these six areas of language design. Let's take a look at each.



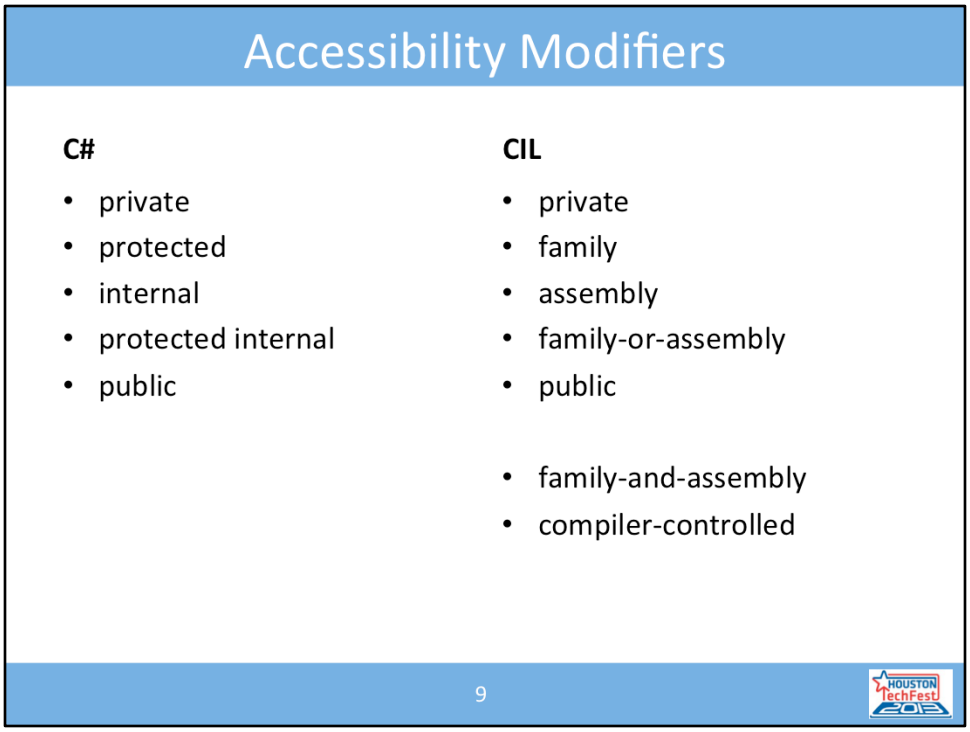
If you ever owned an older HP calculator, you may already know about stack-based programming due to its use of Reverse Polish Notation. If not, then you won't find it hard to learn.

Most of you know infix notation, where you specify an operand, the operator, and another operand. You experience that when you type 2 + 3 into a calculator or into some source code.

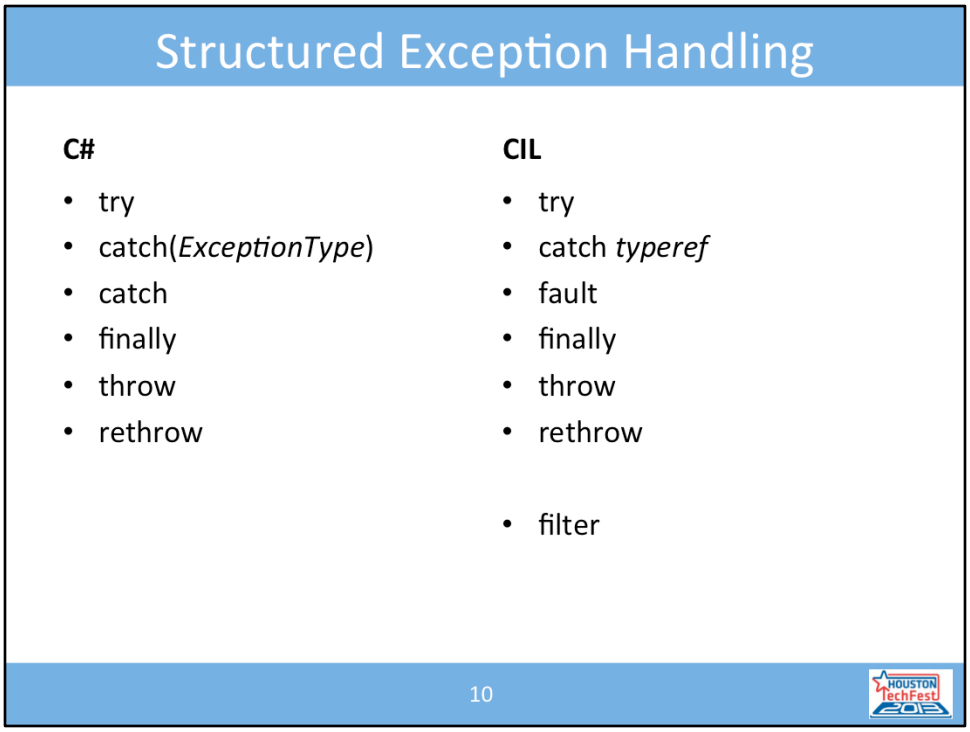
Stack-based programming uses postfix notation. You specify both operands and, then, the operator. As you see here, I specify the operands 1, 2, and 3. Then, I perform the + operation which uses the two most recent operands. The runtime then puts the result of that operation back onto the stack. Here, after the runtime performs the addition operation of 2 + 3, it puts the result of 5 back onto the stack.



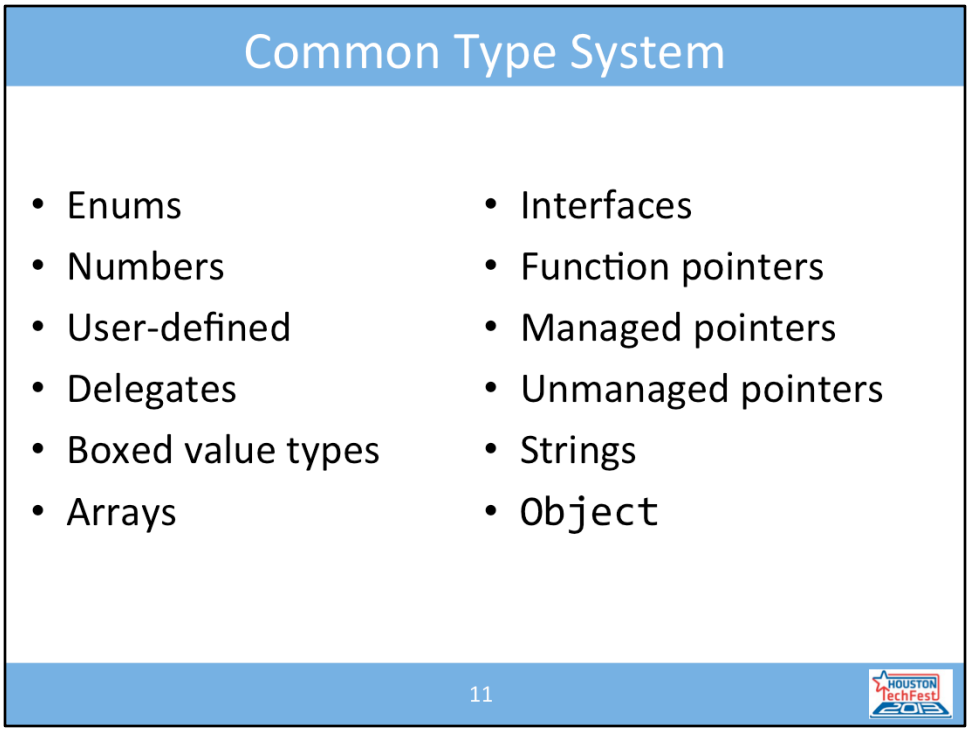
If you can't control the flow of execution in a program, then you don't have much of a programming language. On the left, you see all of the flow control statements that you get from C#. On the right, you'll see that the CIL uses only three statements to create the functionality of all of C#'s flow control.



What's an object-oriented language without encapsulation? C# provides five different accessibility levels for the types and members that you define in the language. The CIL has analogous accessibility modifiers, plus a couple extra.

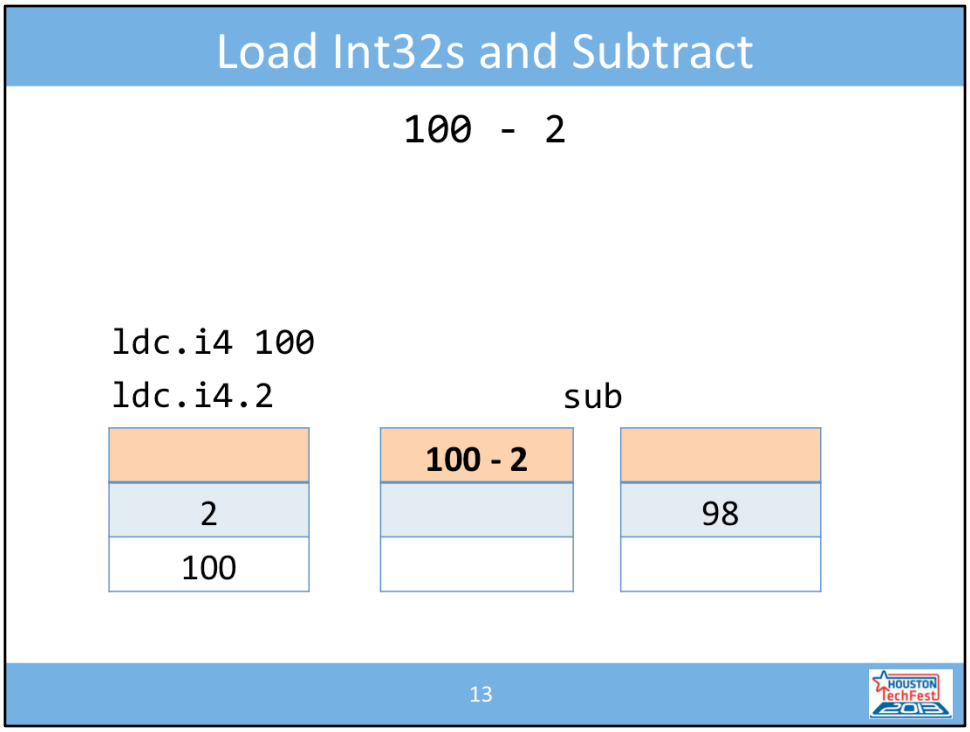


C# exposes most of the structured exception handling features of the CIL with one important exception, the filter instruction. To see that in action in a mainstream .NET language, you have to switch over to VB.NET and use the Catch/As/When form of exception handling block.



The Common Intermediate Language relies on a common type system provided by the Common Language Infrastructure. It only takes 12 ideas to provide the functionality of all of .NET. This way, as you all know, you can use the assemblies compiled from one language in the project based on another language.

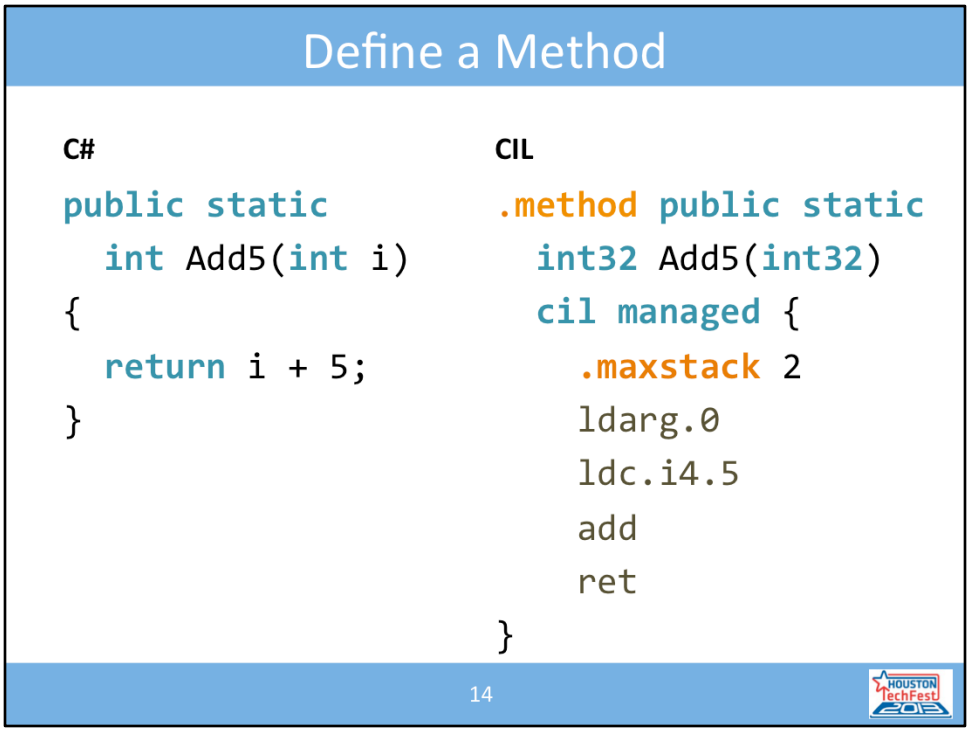
9:30	Meet the CIL (7.5 mins)
9:37.5	Some instructions (7.5 mins)
9:45	How C# compiles to IL (20 mins)
10:05	Crazy stuff with IL (15 mins)
10:20	Q&A



First up, let's look at some instructions that would perform a numerical calculation.

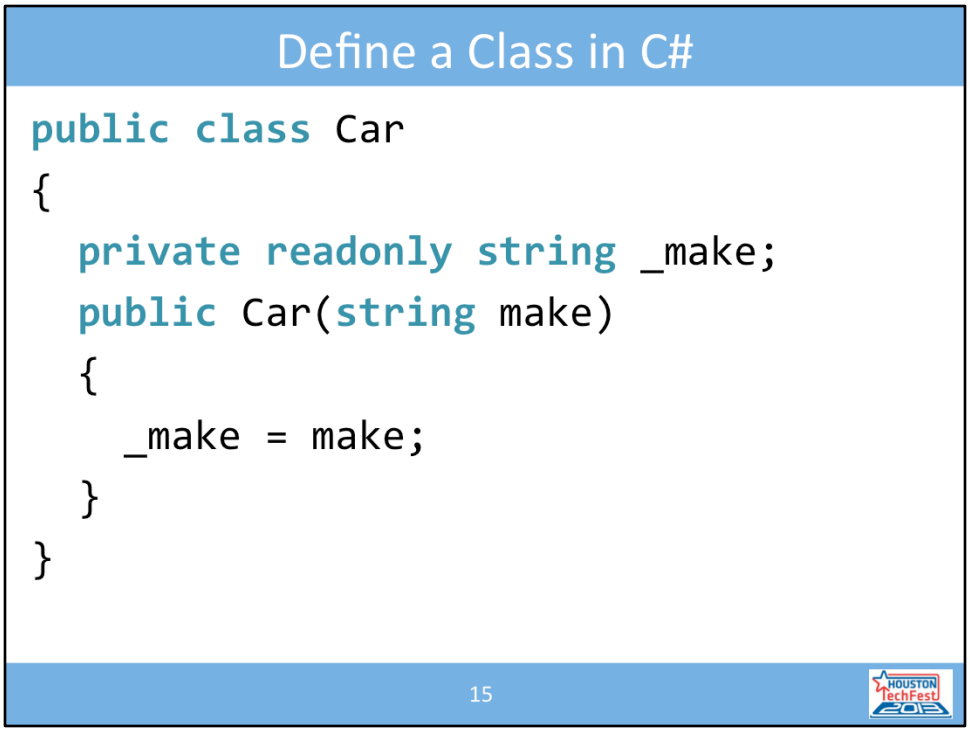
ldc.i4 loads a constant, four-byte integer value. In this case, it loads the value "100".
ldc.i4.2 is a shortcut method for loading the four-byte integer representation of the value "2".
sub is an operation that pops the two values from the stack, subtracts them, and pushes the result back onto the stack.

Now let's see how we can put that type of expression into a method…



cil managed means that it contains Common Intermediate Language statements and only managed code.
.maxstack provides an optimization for simple compilers.
ldarg.o loads onto the stack the zeroth argument which, for static methods, is the value of the first parameter of the method.
ldc.i4.5 loads onto the stack a four-byte integer with a value of 5.
add pops off the top two arguments, adds them, and pushes the result back onto the stack
ret returns from the method.

This method has a return type (int32) and, when ret gets invoked, the evaluation stack must have one value of type int32 on it. Otherwise, the CLR will note an “invalid program” and will throw a runtime exception.

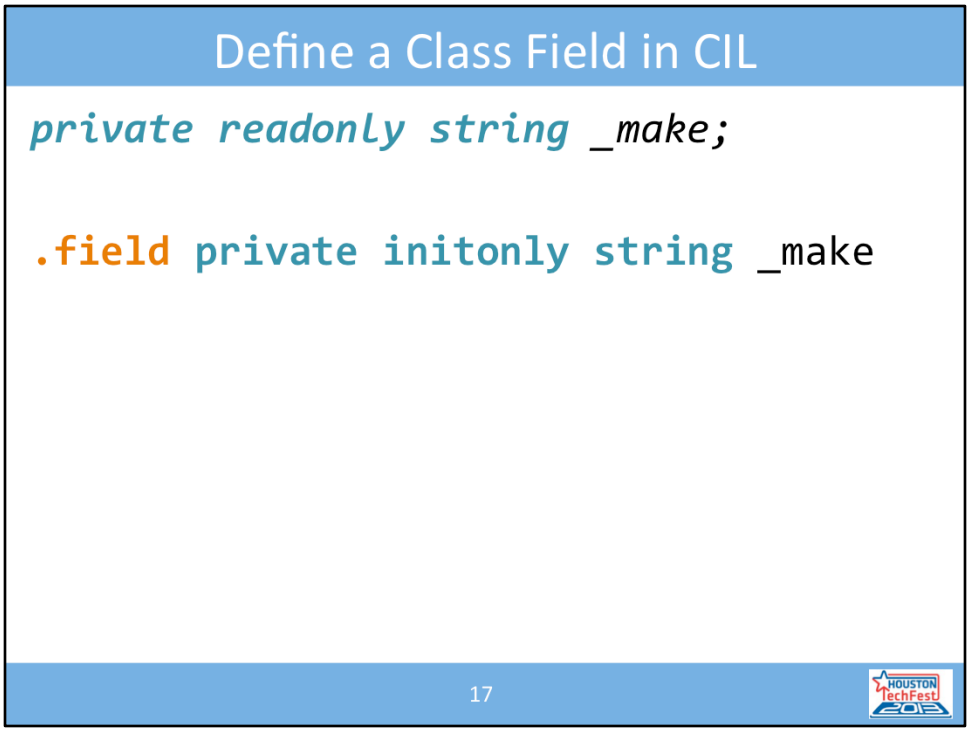


Here you see a normal class written in C#. Let's take a look at how to do that in CIL. We'll attack it in steps from the outside-in.

Define a Class in CIL

```
public class Car {...}

.class public auto ansi
  beforefieldinit Car
  extends [mscorlib]System.Object
{
  ...
}
```


auto means automatic layout of the memory.
ansi means that it defaults to ANSI strings when marshalling to unmanaged code. The runtime ignores this when working strictly in managed code where we get treated to Unicode delights.
beforefieldinit means that the type constructor, the static constructor, does not need invocation before access to static members. This provides an optimization.
extends means what you think it means: inherits from.



The CIL definition of a field in a class looks a lot like the C# syntax.

Define a Class Constructor in CIL

```
public Car(string make) {...}

.method public hidebysig specialname
  rtspecialname instance default
  void '.ctor' (string make)
  cil managed
{
  ...
}
```

18

The constructor method of a user-defined type (or class) looks very little like the simplicity of the C# declaration.

hidebysig is for tools like the C# debugger. The runtime ignores it.
specialname means that some tools think it has a special name, like debuggers.
rtspecialname means that the runtime thinks that this is a special name.
instance means that this method requires a “this” reference.
default means that the method accepts a finite and known number of arguments.
‘.ctor’ is the name of all constructor methods in the CIL.

I find it interesting that a constructor method returns nothing as we see by the “void” part of the signature. That means when we type “new Car” in C#, the new operator really manages the instantiation of the object and the constructor merely initializes it.

Assign a Parameter to a Field in CIL

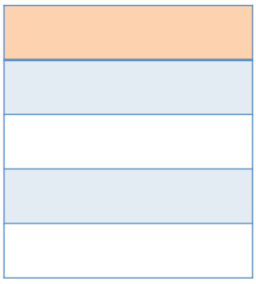
```
_make = make;

.maxstack 4
ldarg.0
call instance void object::'.ctor'()
ldarg.0
ldarg.1
stfld string Car::_make
ret
```

19

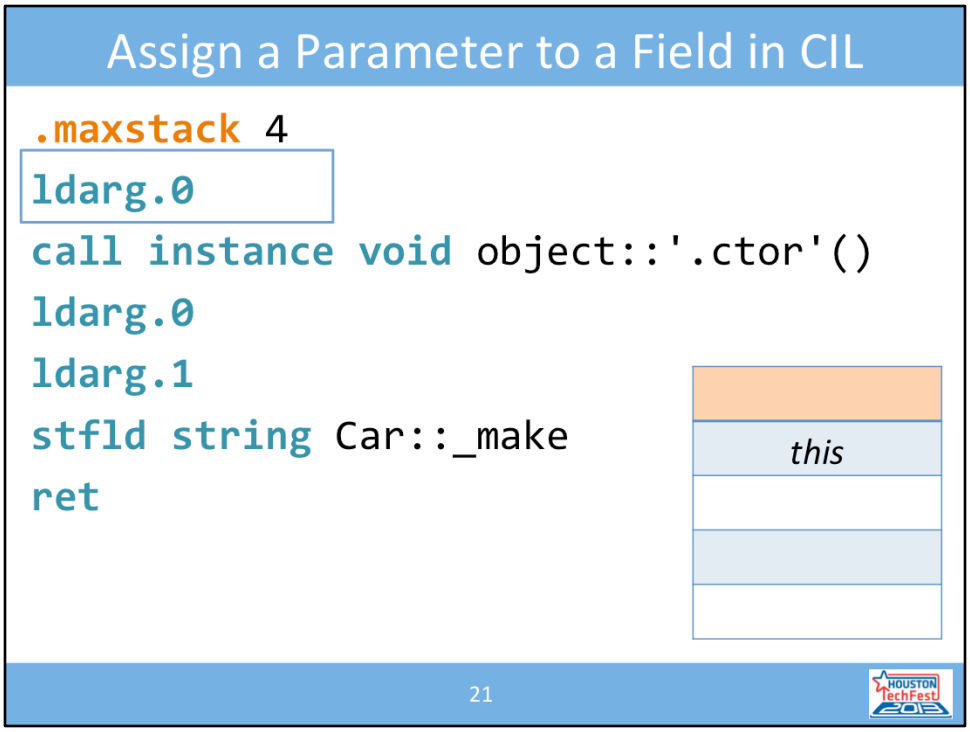
Inside the constructor, we assigned the “make” argument to the “underscore make” field. In CIL, you can see the instruction set. Let’s run through those instructions to see how it gets evaluated in stack-based programming.

Assign a Parameter to a Field in CIL

```
.maxstack 4
ldarg.0
call instance void object::'.ctor'()
ldarg.0
ldarg.1
stfld string Car::_make
ret
```



.maxstack really doesn't mean anything to the compiler other than a maximum size that the compiler should create to support the operations that will occur in the method. We could declare this as ".maxstack 1000" and it would make no difference to the C# compiler. You should note, though, that a .maxstack number smaller than the needs of the method will cause runtime errors.

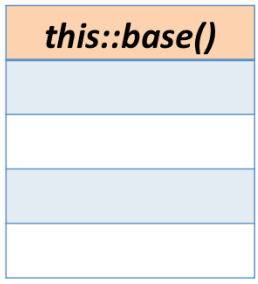


ldarg.0 loads the first argument which, in the case of an instance method, is the "this" reference. Those of you that program in Python should find this eerily familiar. For those of you that don't, all instance methods have a zeroth argument which points to the self reference.

Important to note about this: methods do not belong to classes in the CIL. I can call a method with any "this" parameter and, in some cases, it will succeed! This makes the CIL a little less object-oriented than what most people would consider object-oriented.

Assign a Parameter to a Field in CIL

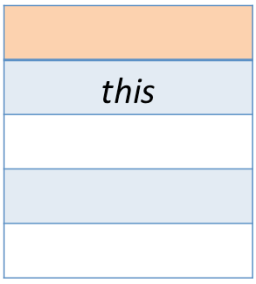
```
.maxstack 4
ldarg.0
call instance void object::'.ctor'()
ldarg.0
ldarg.1
stfld string Car::_make
ret
```



call pops off the top argument from the stack which is the “this” reference. It then calls the “object”’s constructor with the “this” as the “this” for the constructor.

Assign a Parameter to a Field in CIL

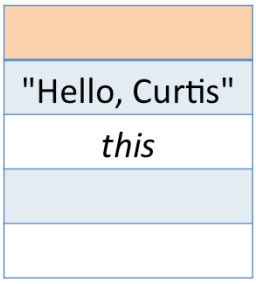
```
.maxstack 4
ldarg.0
call instance void object::'.ctor'()
ldarg.0
ldarg.1
stfld string Car::_make
ret
```



ldarg.0, again, loads the “this” reference

Assign a Parameter to a Field in CIL

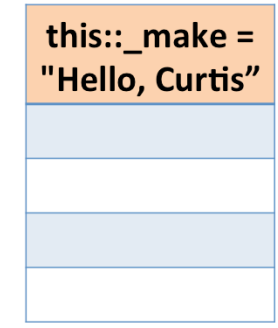
```
.maxstack 4
ldarg.0
call instance void object::'.ctor'()
ldarg.0
ldarg.1
stfld string Car::_make
ret
```



ldarg.1 loads the “make” argument to the constructor

Assign a Parameter to a Field in CIL

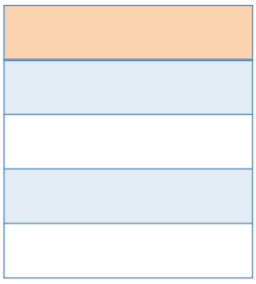
```
.maxstack 4
ldarg.0
call instance void object::'.ctor'()
ldarg.0
ldarg.1
stfld string Car::_make
ret
```



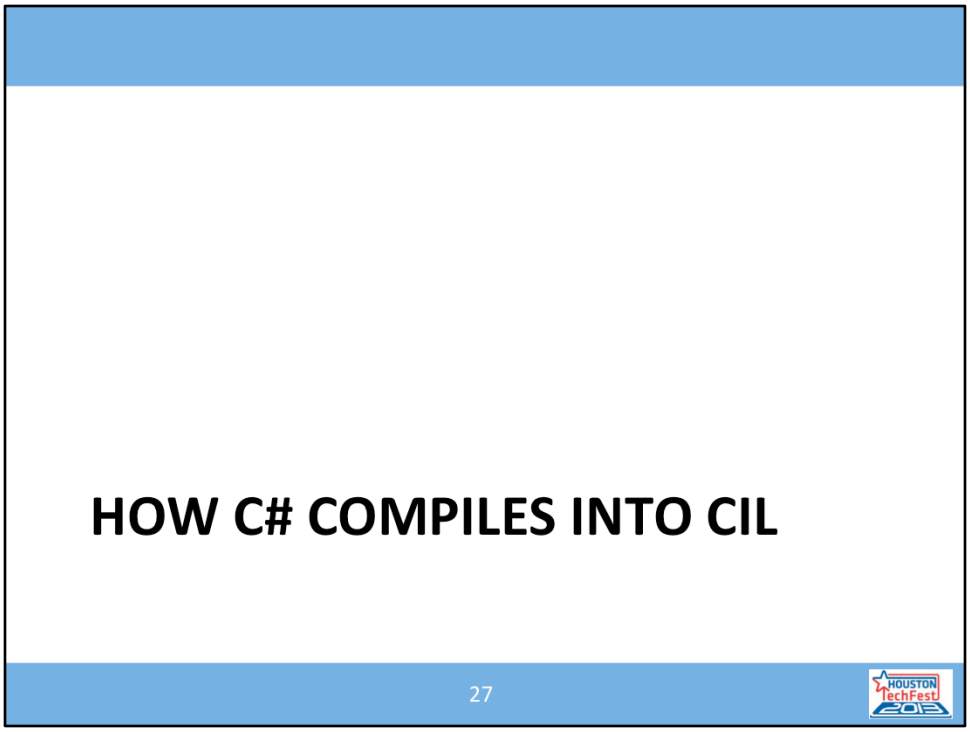
stfld pops off both argumens and stores the “make” argument into the field named “_make” of the instance on the stack. It does not push a value back onto the stack.

Assign a Parameter to a Field in CIL

```
.maxstack 4
ldarg.0
call instance void object::'.ctor'()
ldarg.0
ldarg.1
stfld string Car::_make
ret
```

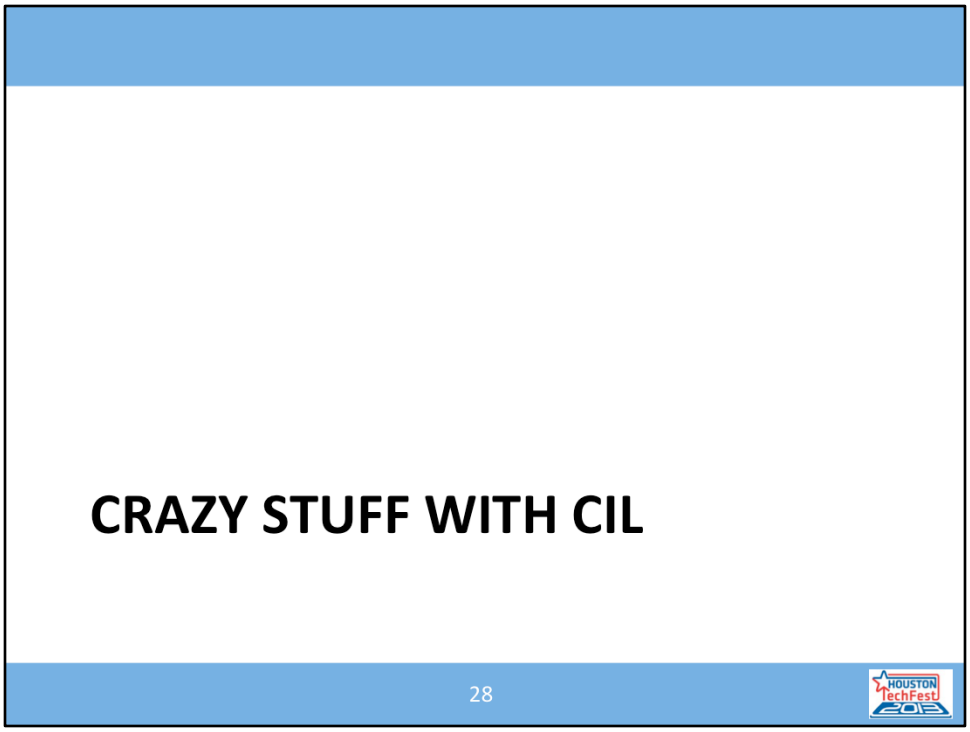


ret returns “void” because nothing exists on the stack and the .ctor method to which it belongs is declared void.



9:30	Meet the CIL (7.5 mins)
9:37.5	Some instructions (7.5 mins)
9:45	How C# compiles to IL (20 mins)
10:05	Crazy stuff with IL (15 mins)
10:20	Q&A

At this point, I switch over to the code in the C# project named Example.



9:30	Meet the CIL (7.5 mins)
9:37.5	Some instructions (7.5 mins)
9:45	How C# compiles to IL (20 mins)
10:05	Crazy stuff with IL (15 mins)
10:20	Q&A

At this point, I switch over to the code in the C# project named AdapterBuilder.

9:30	Meet the CIL (7.5 mins)
9:37.5	Some instructions (7.5 mins)
9:45	How C# compiles to IL (20 mins)
10:05	Crazy stuff with IL (15 mins)
10:20	Q&A

References

- ECMA-335: Common Language Infrastructure (CLI) Partitions I to VI
- Liberal use of **monodis** and **ilasm** tools from the Mono toolkit http://mono-project.com
- Code available next week in the **htechfest** repository of @realistschuckle over on GitHub http://github.com/realistschuckle/houstontechfest2013

Please Leave Feedback During Q&A

If you leave session feedback and provide contact information, you will be qualified for a prize

Scan the QR code to the right or go to bit.ly/htf130101