A compiler-writer's guide to C#

Mark Shields
University of Melbourne
`http://www.cse.ogi.edu/~mbs`

Overview

- Microsoft's contractual-obligations alternative to **Java**
- Now ECMA Standard 334
- The usual features:
  - Single-implementation-inheritance/multiple-interface-inheritance object-orientation
  - Overloading and overriding of methods
  - Exceptions
  - Subtyping, explicit conversions
  - Garbage collection

Not just Microsoft Java

- Some nice touches:
  - Enumeration types
  - Pass-by-reference and output-only params
  - Variable argument lists
  - User-defined operators
  - User-defined conversions
- Sugar for some common programming situations:
  - Properties and indexed access
  - Collection enumeration
  - Events and “delegates”
- Many efficiency-inspired features:
  - Virtual and non-virtual methods
  - Unboxed values, with automatic boxing/unboxing support
  - Direct (“unsafe”) pointer manipulation
  - Checked and unchecked arithmetic

Open to growth?

- Possible language extensions:
  - Generics *[Syme, Kenedy, PLDI 2001]*
  - Join patterns *[Benton, Cardelli, Fournet, FOOL 9]*
  - Closures, staged computation, higher-kinded generics, ...
- All beyond ECMA though

Implementations

- Typically will be run on (some implementation of) the Microsoft Common Language Runtime:
  - Dynamic “assembly” loading and JIT compilation
  - Class-centric stack-based intermediate language
  - Support for type reflection
- Targeting the CLR is easy
- So far though, not many compilers:
  - Microsoft Visual Studio.Net includes **C#** compiler
    [Commercial] [Windows only]
  - Microsoft .NET SDK includes command-line **C#** compiler
    [Free (adverb)] [Windows only]
  - Microsoft Rotor includes source of old version of command-line **C#** compiler
    [`sscli/clr/src/csharp/csharp/sccomp`, 77kl **C++** code]
    [Free (adjective)] [Windows and BSD]
  - Ximian Mono includes source of (alpha) **C#** compiler
    [`mcs-0.11/mcs`, 31kl **C#** code] [Free (adjective)] [Windows and Linux/BSD]

Plan

- Give overview of language from point of view of compiler writer
- Code generation for CLR is trivial, so we'll focus on type checking
- Specification is 410 pages, almost no formal methods, and frustratingly verbosely written
- We'll boil much of it down to five parts:
  1. Syntactic Quirks
  2. Types and Declarations
  3. Procedural sub-language
  4. Inheritance
  5. Sugar

I: Syntactic Quirks

Lexical

- No lexical distinction between type and term identifiers
- Unicode escapes in identifiers and string literals aka **Java**
- Keywords may be used as identifiers by prefixing with @
- Conditional compilation: `#define`, `#undef`, `#if`/`#elif`/`#else`/`#endif` (*no* macro expansion)
- Most infix/postfix operator names may be used as member function names by prefixing with `operator`
  - But only for declarations:

    ```
    public static MyInt operator + (MyInt x, MyInt y) { ... }
    ```
  - And only for built-in operator names
  - Hence only useful for overloading built-in operators
  - Can overload literals `true` and `false`
  - None of this in **Java**

Namespace control

- *Compilation units* identified with files, but filename not significant
- *Assemblies* units of dynamically loaded code (executable or library)
- Assemblies contain multiple compilation units
- Type declarations cannot span compilation units
- Compilation units may contain multiple type declarations
- Explicit *namespace* blocks qualify enclosed defined names

```
namespace N1 {
  class C { ... }   /* == N1.C */
  namespace N2 {
    class C { ... } /* == N1.N2.C */
  }
}
namespace N3.N4 {
  class C { ... }   /* == N3.N4.C */
}
```

Namespaces *cont.*

- Same namespace may be distributed over many scopes within many compilation units

```
namespace N1 {
  class D { ... }   /* == N1.D */
}
```

- Types may be qualified
- Types (but *not* namespaces) may be imported from a namespace

```
using N1;

/* C    == N1.C */
/* D    == N1.D */
/* N2.C undefined */
```

- Name clashes tested lazily (aka **Haskell**)
- Types names and namespace names may be abbreviated

```
using N12 = N1.N2;
/* N12.C == N1.N2.C */
using N1C = N1.C;
/* N1C   == N1.C */
```

- Abbreviations not significant in other abbreviations

Access control

- Most declarations specify their visibility

Modifier	Visibility
`public`	everywhere
`protected internal`	subclasses and assembly
`protected`	defining class and subclasses
`internal`	assembly
`private`	defining class

- Not all modifiers apply to all declaration forms in all contexts
- Accessibility must be *consistent*. Eg:

```
public class C {
  private enum D { MkD }
  public D m() { ... }
}
```

Error since `m` is `public` but result is `private`

Nested types

- Classes and structs (but not interfaces) may be nested, but (unlike **Java**) only significance is in qualification and accessibility
- Many other named definitions similarly qualified

```
class C {
  private static void f() { return; } /* == C.f */

  class D {                           /* == C.D */
    public static void g() { f(); }   /* == C.D.g */
  }
}
```

- No namespaces within classes
- Let M range over namespace names. Let Q range over *namespace contexts* $(M\,\mathtt{.})*((C \mid S)\mathtt{.})*$.
- Can formalize all this in the well-kinding judgement

$$Q \mid \Gamma \vdash \tau : \mathtt{Type} \hookrightarrow \tau'$$

"In namespace context Q, τ is well-formed type under Γ, and is fully-qualified as τ'"

II: Types and Declarations

Types (τ)

- Value types: instances stored on stack, passed by value
  - Primitive value types: (υ)

Type	Size
`sbyte`, `byte`	1 byte
`short`, `ushort`	2 bytes
`int`, `uint`	4 bytes
`long`, `ulong`	8 bytes
`char`	2 bytes
`float`	4 bytes
`double`	8 bytes (IEEE)
`decimal`	16 bytes (base 10 exponent, no NaN's)
`bool`	1 byte

  - Enumeration types: E where E is a declared enumeration
  - Structure types: S where S is a declared structure

Types *cont.*

- Reference types: instances stored in heap, passed by reference
  - Class types: C where C is a declared class
  - Interface types: I where I is a declared interface
  - Array types: τ`[`$,^n$`]` for $n \geq 0$. The array's *rank* is $n+1$.
    (One of only two structural types!)
  - Strings: `string`
- Pointer types (more later)

Subtypes

- **C#** uses implicit subtype polymorphism
- (cf **Haskell**'s implicit parametric polymorphism)
  - $\tau \leq \tau'$ (τ is a *subtype* of τ') if any value of type τ may be coerced to a value of type τ'
  - For reference types, coercion will always be the identity
  - For value types, coercion will change bits without changing value
- The subtype relation is rich and user-extensible
- Actually quite a few relations which come into play for various features

Declarations

- Compilation unit is set of namespace and type declarations (no global functions or values)
- Type declarations include classes, structures, interfaces and enumerations
- Within class and struct declarations may declare types, fields, constructors, destructors, methods, and constants
- (To make examples easier, we'll assume an interpreter for statements:

  ```
  > int i = 1 + 1;
  > Console.WriteLine(i);
  ==> 2
  ```

  This avoids having to wrap every code fragment within a declaration)

Class declarations

- Key unit of declaration: `class` C `:` $\overline{C'} + \overline{I}$ `{` de^{cls} `}`
  - C is the class name
  - $\overline{C'}$ is either empty or the single *base class* of C
  - $\overline{I}$ are *interfaces implemented by* C
  - $decls$ are the *class members* of C
- Base class is implicitly `object` (cosmic root) if none given
- A class declaration does four things:
  - Introduces a new nominal record type as an extension of an existing nominal record type
  - Introduces some (stylized) procedures operating on references to instances of that record type
  - Asserts that $decls$ implement the interfaces declared $\overline{I}$
  - Extends the subtyping relation to make C a subtype of each of $\overline{C'} + \overline{I}$
- That $C \leq C'$ is standard “by-width” record subtyping
- That $C \leq I_i$ is more subtle (more later)

Class declarations: example

```
public class Point {
  const int zero = 0;                /* constant */
  int x;                             /* fields */
  int y;
  public Point(int x, int y) {       /* constructor */
    this.x = x; this.y = y;
  }
  public void move(int dx, int dy) { /* member decls */
    this.x += dx; this.dy += dy;
  }
  public Point moved(int dx, int dy) {
    return new Point(this.x + dx, this.y + dy);
  }
}

public class ColoredPoint : Point {
  enum Color { Red, Blue, Green }    /* type decl */
  Color c;                           /* extended field decl */
  public void makeRed() { c = Red; } /* extended member decl */
}
```

Structure declarations

- As for classes, but:
  - May only derive (implicitly) from `object`
  - (May still implement any number of interfaces)
  - No virtual methods
- Restrictions ensure structures are mostly just pass-by-value records

```
struct Point {
  public int x;
  public int y;

  public void move(int dx, int dy) {
    this.x += dx; this.y += dy;
  }
}
```

Interface declarations

- Declares a name for a set of members which may be implemented by classes (says nothing about fields, constructors, types or constants):

  `interface` $I : \overline{I'}$ `{` $decls$ `}`

  - I is the interface name
  - $\overline{I'}$ are the *base interfaces* which I extend
  - $decls$ must be bodiless member declarations only

- Simple example:

```
public interface IPoint {
  public void move(int dx, int dy);
}

public class Point : IPoint {
  ...
  public void move(int dx, int dy) { ... }
  ...
}

> Point p = new Point(1, 2);
> IPoint ip = p;
> ip.move(3, 4);
```

Interface declarations *cont.*

- Interface extension is mostly just set union:

```
public interface IColoredPoint {
  public void makeRed();
}

public class ColoredPoint : Point, IColoredPoint {
  ...
  public void makeRed() { c = Red; }
  ...
}
```

Enumeration declarations

- Not in **Java**
- Introduces a type and a discrete set of constants of that type

```
enum Color { Red, Green, Blue }
```

```
> Color c = Red;
> Console.WriteLine(c);
==> Red
```

- Constants may be explicitly assigned from integers
- Some arithmetic on enumeration constants
- Underlying representation may be specified

```
enum Color : byte { Red = 2, Green = Red - 1, Blue = Green - 1 }
```

- No enumeration extension `:-)`

Constant declarations

- May declare compile-time constants, but not for structures or arrays (or pointers)

```
class C {
  public const double pi = 3.141592653589793238462643D
}
```

- Value must be computable at compile-time:
  - Literals, built-in operators and other constants only
  - No cycles
  - No method calls
  - No instance creation (so constants of reference type must be `null`)

Constructor/destructor declarations

- Two flavors of constructor declarations:
  - *Static constructor*: called before first access of static field, static member or instance constructor to initialize static fields of class
  - *Instance constructor*: called after space for new instance has been created to initialize instance fields.
- Destructors called when instance about to be garbage collected (no “static destructors”)
- Instance constructors and destructors passed an implicit reference to instance named `this`

Constructor/destructer declarations *cont.*

- Example:

```
class C {
  int i;
  public static C() { Console.WriteLine("loaded"); }
  public C(int i)
    { this.i = i; Console.WriteLine("created {0}", i); }
  ~C() { Console.WriteLine("destroyed {0}", i); }
}

> C c = new C(1);
==> loaded
==> created 1
> c = new C(2);
==> created 2
<arbitrary operations>
==> destroyed 1
```

- May explicitly call constructors from constructors

```
class D : C {
  int j;
  public D() : this(0, 0) { }
  public D(int j) : this(0, j) { }
  public D(int i, int j) : base(i) { this.j = j; }
}
```

Field declarations

- Two flavors of field declarations:
  - *Static fields*: global variable, name qualified by class
  - *Instance fields*: record field within each instance, accessible via `.` notation
- Eg:

```
class C {
  int i;
  public static int j;
  public C(int i) { this.i = i; }
}

> C c = new C(1);
> c.i = c.i + 2;
> Console.WriteLine(c.i);
==> 3

> C.j = 1;
> Console.WriteLine(C.j);
==> 1
```

Fields declarations *cont.*

- Fields may be explicitly initialized
- Fields may be read-only
  - for instance fields: initialized when instance constructed
  - for static fields: initialized when class loaded

```
class D {
  readonly int i;
  public static readonly C c = new C(1);
  ...
  public C(int i) { this.i = i; }
}
```

Field declarations *cont.*

- Instance/static fields initialized in sequence
  1. to default value appropriate for type (`0`, `null`, `false`, etc); then
  2. by explicit initializers; then
  3. by assignments within instance/static constructor
- Example:

```
class E {
  public static int a = b + 1;
  public static int b = a + 1;

  public static E() { a = a + 2; b = b + 3; }
}

> Console.WriteLine("a = {0}, b = {1}", E.a, E.b)
==> a = 3, b = 5
```

Member declarations

- Three flavors of member functions:
  - *Static*: global function, name qualified by class
  - *Instance, non-virtual*: global procedure, name qualified by class, implicit `this` parameter
  - *Instance, virtual*: (conceptually) readonly field of instance (of procedural type) with default binding to given body, implicit `this` parameter (more later)
- Complicated by *overloading* and *overriding* (more later)

Member declarations *cont.*

- Non-virtual instance methods always take implicit reference to instance as implicit argument named `this`

$$\tau_r\ x(args)\ \ \{\ \ stats\ \ \}$$

  within class/interface N is implemented as

$$\tau_r\ N\ .\ x(\tau'\ \texttt{this}, args)\ \ \{\ \ stats\ \ \}$$

- Calls to non-virtual instance methods always with respect to particular instance: $exp\,\texttt{.}\,x(args)$ is implemented as $N\ .\ x(exp, args)$
- (But we must determine *which* N and which x within N to call (more later))
- (Virtual methods bit more complicated (more later))
- Instance fields implicitly in scope within instance methods

```
class C {
  public int i;
  public C(int i) { this.i = i; }
  public int next() { i++; return this.i; }
}
```

Member declarations *cont.*

- Result may be any type or `void` (`void` is *not* a type!)
- Arguments may be of any type, and use any of 3 calling conventions:
  - By value: `int x`
  - By reference: `ref int x` (cf **C++** `int& x`)
  - By output: `out int x`
- Calls must also specify convention, which must match:

```
class C {
  public static int m(int x, ref int y, out int z) {
    x = x + 1;
    y = y + 1;
    z = x + y;
    return x + y + z;
  }
}

> int x = 1, y = 2, z;
> Console.WriteLine("result = {0}, x = {1}, y = {2}, z = {3}",
>                   C.m(x, ref y, out z), x, y, z);
==> result = 10, x = 1, y = 3, z = 5
```

Member declarations *cont.*

- Variable arguments possible using *parameter arrays*

```
class C {
  public static int m(int x, params int[] args) {
    Console.WriteLine("Called with {0} args", args.Length + 1)
  }
}

> C.m(1);
==> Called with 1 args
> C.m(1, 2, 3);
==> Called with 3 args
> int[] a = {2, 3, 4};
> C.m(1, a);
==> Called with 4 args
```

- Neither of above in **Java**
- Covariant subtyping for value arguments, invariant for `ref` and `out` arguments

III: Procedural sub-language

Literals

Type	Example	
`int`	`42`, `0x2a`	
`uint`	`42U`	
`long`	`42L`	
`ulong`	`42UL`	
`char`	`'a'`, `'\n'`	
`float`	`42.0F`	
`double`	`42.0D`	
`decimal`	`42.0M`	
`bool`	`true`, `false`	
`string`	`null`, `"Hello world!"`	
E	x	where E is an enum-type with enum-member e
τ	`null`	where tau is a class/interface/array type

Expressions

- Side-effecting of course!
- Static and instance method call
- Fields access
- Usual operators, compound assignment, and conditional expressions of **C**:
  `++`, `-`, `+`, `-`, `!`, `~`, `*`, `/`, %, «, », `<`, `>`, `<=`, `>=`, `==`, `!=`, &, ^, `|`, &&, `||`, `?:`, `=`
  - logical operators typed with `bool` rather than `int`
  - `char` is not `int`
  - built-in operators for `string` and `decimal`
  - Covariant subtyping for assignment:

    ```
    > C c = new C();
    > object o = c;
    ```
- Compound assignment: $exp\ \ op$= exp' equivalent to

  `{` τ `x =` exp`; x = (`τ`)(x` $op\ \ exp'$`); return x; }`

  where $exp : \tau$ and $op \in \{$`*`, `/`, %, `+`, `-`, «, », &, ^, `|`$\}$
- Instance creation: `new` C`(`$\overline{exp}$`)`
- Array creation: `new` τ`[`$\overline{exp}$`]` (more later)

Expressions *cont.*

- Type reflection: `typeof(`τ`)` yields instance of `System.Type` representing τ
- Run-time type testing: exp `is` τ yields `true` iff exp is not `null` and run-time type of exp is subtype of τ
- Explicit conversion: `(`τ`)`exp where $exp : \tau'$ invokes *explicit conversion* from τ' to τ (which must exist) (more later)
- Silent conversion: exp `as` τ where τ reference type. Yields `null` if exp is `null` or the run-time type of exp is not a subtype of τ
- Numeric overflow: `checked(`exp`)` enables overflow checking for evaluation of numeric operators *lexically* within exp (throws `System.OverflowException`). Dually for `unchecked(`exp`)`

Statements

- Usual statements of **C**: `if`, `if/else`, `while`, `do/while`, `for`, `break`, `continue`, `return`
- Variables introduced at start of any block
- `switch` on value types (including enumerations) and `string` (unlike **Java**), no fallthrough:

```
switch (i) {
case 0:
  ...
  break;
case 1:
  ...
  goto case 0;
case 2:
case 3:
  ...
  goto default;
default:
  ...
  break;
}
```

- labeled statements, `goto` label or case arm

Arrays

- multi-dimensional, base zero indexed, index checked (raises `System.IndexOutOfRangeException` if fail)

```
> int[,] a = new int[5,2];
> a[1,2] = 1;
> Console.WriteLine(a[0,0]);
==> 0
```

- May be initialized (but no "array literals")

```
> int[,] a = {{0, 1}, {2, 3}, {4, 5}}
> Console.WriteLine(a[2,1]);
==> 5
```

- Covariantly subtyped for reference types (!)

```
> string [] strs = { "A", "B", "C" };
> object [] objs = strs;
> Console.WriteLine(objs[1])
==> "B"
```

- Hence every update requires type compatability test (raises `System.ArrayTypeMismatchExcepiton` if fail)

```
> objs[2] = (object)2;
==> Uncaught exception: ArrayTypeMismatchException
```

- Invariantly subtyped for value types

Exceptions

- As for **Java**

```
try {
  ...
  throw (new Exception("fail"));
  ...
}
catch (System.NullReferenceException e) { ... }
catch (Exception e) { ... }
catch { ... }
finally  { ... }
```

- User-defined exceptions declared by deriving from `System.Exception`
- No `throws` declarations on methods (hence all exceptions “unchecked” in **Java** parlance)

Exceptions *cont.*

- *All* failures cause exceptions:

Run-time system	`OutOfMemoryException`
	`StackOverflowException`
Arithmetic	`ArithmeticException`
	`DivideByZeroException`
	`OverflowException`
Dereferences	`NullReferenceException`
	`IndexOutOfRangeException`
Casts	`InvalidCastExpression`
	`ArrayTypeMismatchException`
	`MulticastNotSupportedException`
Load-time ex.	`TypeInitializationException`

- Blocks may be prefixed by `checked`/`unchecked` with same effect as expressions (lexical scope only)

Unsafe code

- C pointer types and pointer arithmetic available as sub-language
- Lexically delimited by `unsafe` modifier on declaration or an `unsafe` block.
- Ok to call an unsafe method from a safe method
- Types
  - Pointer types: τ `*` or `void *` (The only other structural type!)
  - Declaration of τ must not contain reference types, ie must be primitive value type, pointer type, or structure with fields only of these types

Unsafe code *cont.*

- Expressions
  - Usual dereferences: `*`exp, exp`->`id, exp`[`exp'`]`
  - Usual arithmetic: `++`, `-`, `+`, `-`
  - Usual comparison: `==`, `>`, etc
  - `sizeof(`τ`)` only in unsafe context
  - `&`exp legal if exp denotes a *fixed variable*, ie is a local variable, field of fixed variable, or pointer dereference
  - Addresses of *movable variables* can only be taken within specific scope (so gc can be told not to move relevant object)

```
static unsafe void Test() {
  int[] a = new int[100];
  fixed (int* p = a) {
    for (int i = 0; i < 100; i++)
      *p++ = 1
  }
}
```

Unsafe code *cont.*

- Can allocate on stack

```
static unsafe void Test() {
  char* buf = stackalloc char[16];
  fixed (char *p = "A string") {
    char *q = buf;
    while (*p != 0)
      *q++ = *p++;
    *q = 0;
  }
}
```

IV : Inheritance

Now it starts getting interesting...

- Our goal now is to understand the interaction of three features:
  - Subtyping (implicit conversions)
  - Overloading (same name, different type signatures)
  - Overriding (stylized form of second-order programming)

Signatures

- Let $conv$ (parameter passing convention) range over $\{\texttt{ref}, \texttt{out}, \epsilon\}$
- Define the *type signature* of a sequence of method parameter declarations by:

$$\begin{array}{rcl} signature(\overline{conv\ \tau\ x}) & = & \overline{conv\ \tau} \\ signature(\overline{conv\ \tau\ x}, \texttt{params}\ \tau_p\texttt{[]}\ y) & = & \overline{conv\ \tau} \end{array}$$

- Extend $signature$ to method declarations by

$$signature(\tau_r\ y\texttt{(}\,args\,\texttt{)}\ \texttt{\{}\ \ stats\ \ \texttt{\}}) = signature(args)$$

- Note: return type and `params` are *not* part of type signature
- Define the *expanded signature set* by:

$$\begin{array}{rcl} expsigs(\overline{conv\ \tau\ x}) & = & \{\overline{conv\ \tau}\} \\ expsigs(\overline{conv\ \tau\ x}, \texttt{params}\ \tau_p\texttt{[]}\ y) & = & \{\overline{conv\ \tau} +\!\!+\ \underbrace{\tau_p, \ldots, \tau_p}_{n} \mid n \geq 0\} \\ & \cup & \{\overline{conv\ \tau} +\!\!+\ \tau_p\texttt{[]}\} \end{array}$$

- Likewise extend to method declarations

Overloading

- Method (but not field) declarations may be *overloaded*:
  - method x may have ≥ 1 definitions within class and (transitive) base classes provided their signatures are distinct
  - methods in unrelated classes are already distinguished by respective class names

```
class C {
  public static int not(int i) { return 1 - i; }
  public static int not(bool b) { return b ? 0 : 1; }
}

> Console.WriteLine(C.not(1));
==> 0
> Console.WriteLine(C.not(true));
==> false
```

- Exploited for all the built-in operators

```
public static int operator +(int x, int y);
public static uint operator +(uint x, uint y);
public static float operator +(float x, float y);
public static string operator +(string x, string y);
```

etc

Overloading *cont.*

- Implicit subtyping on argument types may make more than one method *applicable* to a call, overloading resolution chooses “best”

```
class A { ... }
class B : A { ... }
class C : B { ... }
class D {
  public static void m(A a) { Console.WriteLine("D.m(A)"); }
  public static void m(B b) { Console.WriteLine("D.m(B)"); }
  public static void m(A a, B b) { Console.WriteLine("D.m(A, B)");
  public static void m(B b, A a) { Console.WriteLine("D.m(B, A)");
}

> A a = new A();
> C c = new C();
> D.m(a);      /* { D.m(A a) } */
==> D.m(A)
> D.m(c);      /* { D.m(A a), D.m(B b) }, D.m(B b) < D.m(A a) */
==> D.m(B)
> D.m(a, c);   /* { D.m(A a, B b) } */
==> D.m(A, B)
> D.m(c, c);   /* { D.m(A a, B b), D.m(B b, A a) } */
==> Error: D.m(A a, B b) and D.m(B b, A a) incomparable
```

- All else being equal, methods without `params` prefered to those with

Virtual methods

- In **Java**, all methods virtual
- In **C#** (like **C++**), must explicitly distinguish virtual from non-virtual
- Conceptually: virtual methods are stored in instance, non-virtual methods are global procedures:

```
class C {
  int i = 1;
  public virtual void v(int j)
    { Console.WriteLine("C.v({0})", j); }
  public void nv(int j)
    { Console.WriteLine("C.nv({0})", j); }
}
```

is conceptually sugar for:

```
class C {
  int i = 1;
  readonly void v(C, int) =
    \(C this, int j) -> Console.WriteLine("C.v({0})", j);
}

public void C.nv(C this, int j)
  { Console.WriteLine("C.nv({0})", j); }
```

Virtual methods *cont.*

- Similarly, virtual method call is via instance, non-virtual method call is via global procedure:

  ```
  > C c = new C();
  > c.v(2);
  > c.nv(3);
  ```

  is conceptually sugar for:

  ```
  > C c = new C();
  > c.v(c, 2);
  > C.nv(c, 3);
  ```

- In popular parlance: virtual methods “dispatch on run-time type”, non-virtual methods “dispatch on compile-time type”
- In practice: implemented as in **C++** using pointer in instance to vtable of virtual method function pointers in inheritance order

Virtual methods *cont.*

- Virtual functions of (transitive) base class may be (explicitly) *overridden* in derived classes
  - Name, return type and signature in derived class must match declaration in (transitive) base class
  - *Must* declare using `override` keyword
  - If simply wish to shadow an inherited virtual function, *should* declare using `new` keyword (otherwise warning, since could be unintended shadowing of newly introduced virtual member)
- Conceptually: field for virtual method is re-initialized with overridden method when derived class initialized
- Interaction of virtual methods and subtyping gives us a stylized form of *second-order programming*
  - **Conjecture:** this is real reason why OO works quite well
  - Claims of “data encapsulation” are for most part bogus (witness research on representation escape analysis for OO languages)

Virtual methods *cont.*

- For example:

```
class A {
  public void nv() { Console.WriteLine("A.nv"); }
  public virtual void v() { Console.WriteLine("A.v"); }
}

class B : A {
  public new void nv() { Console.WriteLine("B.nv"); }
  public override void v() { Console.WriteLine("B.v"); }
}

class C : B { }

> C c = new C();
> A a = c;
> a.nv();
==> A.nv
> c.nv();
==> B.nv
> a.v();
==> B.v
> c.v();
==> B.v
```

Virtual methods *cont.*

- Concetually sugar for:

```
class A {
  readonly void v(A) =
    \(this) -> Console.WriteLine("A.v");
}
public void A.nv(A this) { Console.WriteLine("A.nv"); }
class B : A {
  public B() { this.v = \(B this) -> Console.WriteLine("B.nv"); }
}
public void B.nv(B this) { Console.WriteLine("B.nv"); }
class C : B { }

> C c = new C();
> A a = c;
> A.nv(a);
==> A.nv
> B.nv(c);
==> B.nv
> a.v(a);
==> B.v
> c.v(c);
==> B.v
```

- But notice contravariance on type of `this`

Virtual methods *cont.*

- Virtual functions may be *abstract* (ie declare a field to hold member function without also supplying a binding).
- Containing class must be similarly declared abstract (and cannot have instances):

```
abstract class A {
  public abstract void v();
}

class B : A {
  public override void v() { Console.WriteLine("B.v"); }
}
```

- Overriding definition may also be abstract
- An overriding definition may be *sealed*, preventing any further overriding in (transitive) derived classes

```
abstract class A {
  public abstract void v() { Console.WriteLine("A.v"); }
}

class B : A {
  public sealed override void v() { Console.WriteLine("B.v"); }
}
```

Interfaces

- Listing an interface I in a class C's base class list:
  - Implies C and its (transitive) base classes provide an implementation for each member function declared in I and its (transitive) base interfaces. (Matching is by name, return type and type signature)
  - Makes C a subtype of I
- Conceptually: interfaces are abstract classes containing only abstract virtual methods.
  - Coercion from C to I fills-in virtual functions of I from member functions (virtual or otherwise) of C
  - This (conceptual) process termed "interface matching"
- In practice: calling a member function through an interface is handled specially
  - Coercion from C to I is the identity
  - Each object has extra pointer to interface dispatch table
  - Interface member functions hashed to offsets
  - Stub code checks for collisions at run-time
  - Since tables can be large, cache performance suffers
  - Hence could optimistically branch and test run-time type information

Interfaces *cont.*

```
interface I {
  public void m();
}
interface J {
  public void n();
}

class C : I, J {
  public void m() { Console.WriteLine("C.m"); }
  public void n() { Console.WriteLine("C.n"); }
}

> C c = new C();
> I i = c;
> J j = c;
> i.m();
==> C.m
> j.n();
==> C.n
```

Interfaces *cont.*

- Unlike **Java**, implementation of interface members may be supplied without polluting class interface

  ```
  class C : I, J {
    void I.m() { Console.WriteLine("C.m"); }
    void J.n() { Console.WriteLine("C.n"); }
  }
  ```

  `m` and `n` not visible from `C`, only via `I` and `J`

- Derived classes may re-implement an interface already implemented by a base class

Type Equality

- Types fully-qualified during kind checking
- Most types nominal

$$\frac{Q = Q' \quad \tau = \tau'}{Q\texttt{.}\tau = Q'\texttt{.}\tau'}$$

$$\frac{}{v = v}$$

$$\frac{}{\texttt{string} = \texttt{string}}$$

$$\frac{}{C/I/S/E = C/I/S/E}$$

$$\frac{\tau = \tau' \quad n = n'}{\tau\texttt{[,}^{n}\texttt{]} = \tau'\texttt{[,}^{n'}\texttt{]}}$$

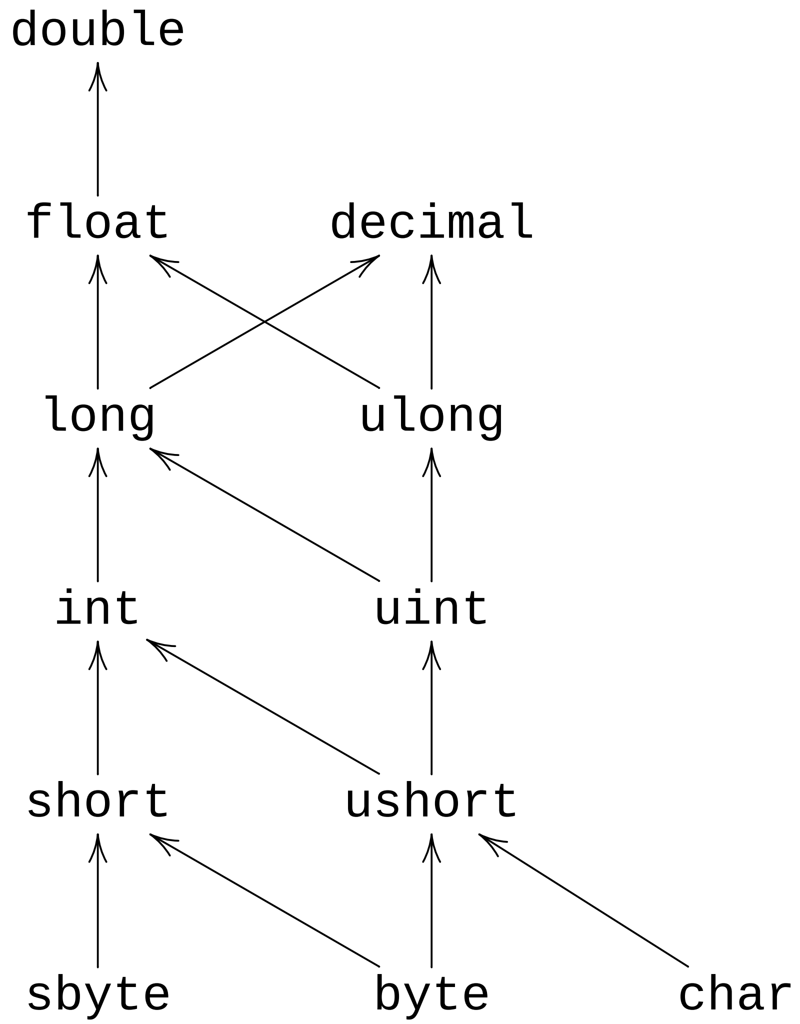
$$\frac{\tau = \tau'}{\tau\texttt{*} = \tau'\texttt{*}}$$

Simple Subtyping: $\leq$

- There are *four* (!) notions of subtyping coexisting in **C#**
- $\tau \leq \tau'$ is the “natural” subtyping relation
- For numeric types: coercion changes representation (bits), but not value (number)
- For reference types: coercion is identity
- Applies in
  - Run-time type testing (`is`)
  - Run-time silent conversion (`as`)
- Constructed from
  - Subtyping on built-in types
  - User-defined class and interface inheritance

Simple Subtyping *cont.*

- For value types:



Simple Subtyping *cont.*

- For other types:

$$\frac{\tau = \tau'}{\tau \leq \tau'} \qquad \frac{\texttt{interface}\ I\ \texttt{:}\ \overline{I'}\ \texttt{\{}\ \mathit{decls}\ \texttt{\}}}{I \leq I'_i}$$

$$\frac{\tau \leq \tau'' \quad \tau'' \leq \tau'}{\tau \leq \tau'} \qquad \frac{\texttt{struct}\ S\ \texttt{:}\ \overline{I}\ \texttt{\{}\ \mathit{decls}\ \texttt{\}}}{S \leq I_i}$$

$$\frac{\texttt{class}\ C\ \texttt{:}\ \overline{C'} +\!\!+ \overline{I}\ \texttt{\{}\ \mathit{decls}\ \texttt{\}}}{C \leq C'_i} \qquad \frac{}{C/I \leq \texttt{object}}$$

$$\frac{\texttt{class}\ C\ \texttt{:}\ \overline{C'} +\!\!+ \overline{I}\ \texttt{\{}\ \mathit{decls}\ \texttt{\}}}{C \leq I_i} \qquad \frac{\tau \text{ reference type} \quad \tau \leq \tau'}{\tau\texttt{[,}^n\texttt{]} \leq \tau'\texttt{[,}^n\texttt{]}}$$

$$\frac{}{\tau\texttt{[,}^n\texttt{]} \leq \texttt{System.Array}/\texttt{System.ICloneable}}$$

User-defined conversion operators

- User may define *implicit* and *explicit* conversion operators

```
class C {
  int i;

  public C(int i) { this.i = i; }
  public static implicit operator int(C c) { return c.i; }
  public static explicit operator sbyte(C c)
    { return (checked ((sbyte)c.i)); }
}

> C c = new C(128);
> Console.WriteLine(c);
==> 128
> Console.WriteLine((sbyte)c);
==> Uncaught exception: OverflowException
```

- In an implicit(explicit) coercion, at most one user-defined implicit(explicit) coercion operator is used
- No user-defined coercions in **Java**

Implicit Subtyping: $\leq^i$

- $\tau \leq^i \tau'$ iff an *implicit conversion* exists from τ to τ'
- Coercion may involve arbitrary computation
- Applies in
  - Assignment, even to array elements
  - Method call, for call-by-value arguments
- Constructed from
  - $\leq$
  - User-defined *implicit conversion operators*
- We can coerce by composing simple subtyping with *at most one* user-defined implicit conversion operator, provided there is a “best” such operator

Implicit Subtyping *cont.*

- Define $\tau \leq^{iud} \tau'$ (τ has an implicit user-defined coercion to τ') iff
  - τ and τ' are not interface types
  - The set $applicable(\tau, \tau')$ ordered under $\leq^{iop}$ has a least element
- Where

$$\begin{array}{l} applicable(\tau_1, \tau_2) = \\ \quad \{\texttt{implicit operator } \tau_4(\tau_3\ x)\ \{stats\} \in impops(\tau_1) \cup impops(\tau_2) \mid \\ \qquad \tau_1 \leq \tau_3, \tau_4 \leq \tau_2\} \end{array}$$

$impops(\tau) =$
the set of implicit conversion operators declared in class or structure type τ and its (transitive) base classes

$$\begin{array}{l} \texttt{implicit operator } \tau_2(\tau_1\ x)\ \{stats\} \leq^{iop} \\ \quad \texttt{implicit operator } \tau_4(\tau_3\ y)\ \{stats'\} \Longleftrightarrow \\ \qquad \tau_1 \leq \tau_3 \wedge \tau_4 \leq \tau_2 \end{array}$$

- Now define $\leq^{i} = \leq \cup \leq^{iop}$
- Note: $\leq^{i}$ need not be transitive!

Explicit Subtyping: $\leq^e$

- $\tau \leq^e \tau'$ iff an *explicit conversion* exists from τ to τ'
- For numeric types: coercion may loose information or raise exception (in `checked` context)
- For reference types: if $\tau \leq \tau'$, coercion is identity.
  If $\tau' \leq \tau$, *downcast* coercion is

```
\x -> let τ'' = run-time-type-of(x)
      in if τ'' ≤ τ' then x
         else throw InvalidCastExpression
```

- Coercion may involve arbitrary computation
- Applies in
  - Explicit casts
- Constructed from
  - Conversions on built-in types
  - User-defined *explicit conversion operators*
- Defined much as for $\leq^i$

Overloading Subtyping: $\leq^{ov}$

- Need variation on $\leq^i$ to ensure methods on signed integers considered "better" than those on unsigned integers
- Define $\leq^{ov}$ as for $\leq^i$, but with additional subtyping on value types:

$$\begin{array}{lllllll}
\texttt{sbyte} & \leq^{ov} & \texttt{byte} & \quad & \texttt{short} & \leq^{ov} & \texttt{ushort} \\
\texttt{int} & \leq^{ov} & \texttt{uint} & \quad & \texttt{long} & \leq^{ov} & \texttt{ulong}
\end{array}$$

- Now define the *better-than ordering* on method definitions as:

$$\begin{array}{l}
\tau_r\ x(\texttt{args})\ \{stats\} \leq^{meth} \tau'_r\ x'(\texttt{args})\ \{stats'\} \iff \\
\quad \exists \overline{conv\ \tau}, \overline{conv'\ \tau'}\ . \\
\qquad\quad x = x' \wedge \tau_r = \tau'_r \wedge |\overline{conv\ \tau}| = |\overline{conv'\ \tau'}| \\
\qquad \wedge\ \overline{conv\ \tau} \in expsigs(args) \\
\qquad \wedge\ \overline{conv'\ \tau'} \in expsigs(args') \\
\qquad \wedge\ \forall i\ .\ conv_i = conv'_i \wedge \tau_i \leq^{ov} \tau'_i
\end{array}$$

- Plus special tiebreaker rule: if methods equal under $\leq^{meth}$, place method with `params` above method without `params`

Members

- Define $members(N)$, the set of member functions accessible within class/interface/structure definition N as follows

$$\frac{\begin{array}{c}\texttt{class/interface/struct}\ N : \overline{N'}\{defs\} \\ \tau_r\ x(args)\{stats\} \in defs \\ \text{“}x\text{ is visible in current context”} \\ \text{“}x\text{ is not an }\texttt{override}\text{ method”}\end{array}}{\tau_r\ x(args)\{stats\} \in members(N)}$$

$$\frac{\begin{array}{c}\texttt{class/interface/struct}\ N : \overline{N'}\{defs\} \\ \tau_r\ x(args)\{stats\} \in members(N'_i) \\ \forall j \cdot \text{“}defs_j\text{ defines }x\text{”} \implies signature(args) \neq signature(defs_j) \\ \text{“no field, constant, type or enumeration member }x\text{ defined in }defs\text{”}\end{array}}{\tau_r\ x(args)\{stats\} \in members(N)}$$

- Notice we don’t keep track of class-of-origin of methods
  - can easily do so explicitly; or
  - *stamp* every member definition with a unique natural number to keep them apart

Method resolution

- Given a method call $exp\ .\ x(\overline{conv\ \ exp'})$
  - where $exp : \tau$ and $\overline{exp' : \tau'}$ (we can always determine types bottom-up, hence no interaction between typing and resolution)
  - and where τ is a class/interface/structure type N

  define the set of applicable members S as

$$\left\{ \begin{array}{l} \tau_r\ y(args)\{stats\} \in members(N) \mid \\ \quad \exists\overline{\tau''}\ .\quad \begin{array}[t]{l} x = y, \\ \overline{conv\ \tau''} \in expsigs(args), \\ \forall i\ .\ conv_i \in \{\texttt{ref}, \texttt{out}\} \Longrightarrow \tau'_i = \tau''_i, \\ \forall i\ .\ conv_i = \epsilon \Longrightarrow \tau'_i \leq^i \tau''_i \end{array} \end{array} \right\}$$

- The call is well-typed (with type τ'_r) if the set S has a least method definition under $\leq^{meth}$ ordering whose return type is τ'_r

- And you thought **Haskell**'s type system was complicated **:** -)

V : Sugar

Base access

- `base` allows access to fields or methods of a base class hidden by a derived class
- Within a declaration within class C with base class C', `base` is type-checked as `((`C'`)this)`
- `base.`$x(\overline{conv\ exp})$ is sugar for (using internal form of method declarations) $C'\ .\ x(\mathtt{this}, \overline{conv\ \mathrm{exp}})$
- This applies even if x is virtual within C and overridden within C'
- So need to be a bit more precise with our encoding of virtual methods...

Properties

- Overload field access and assignment syntax

```
class Even {
  int i = 0;
  public int Value {
    get { return i * 2; }
    set { i = value / 2; }
  }
}

> Even e = new Even();
> e.Value = 11;
> Console.WriteLine(e.Value);
==> 10
```

Properties *cont.*

- Declaration
  - $access\ \tau\ x$ `{ get {` $stats$ `} set {` $stats'$ `} }`
  - Sugar for

    $access\ \tau$ `get_`x`() {` $stats$ `}`

    $access$ `void set_`x`(`τ `value) {` $stats'$ `}`
- Access
  - exp`.`x sugar for exp . `get_`$x()$
  - exp`.`x `=` exp' sugar for exp . `set_`$x(exp')$
- Read-only fields: declare `get` accessor only
- Write-only fields: declare `set` accessor only
- May be virtual

Indexers

- Overload array access syntax

```
class BitArray {
  int[] bits;
  public BitArray(int len) { bits = new int[((len - 1) >> 5) + 1]

  public bool this[int index] {
    get { return (bits[index >> 5] & 1 << index) != 0; }
    set { if (value)
            bits[index >> 5] |= 1 << index;
          else
            bits[index >> 5] &= ~(1 << index); }
  }
}

> BitArray a = new BitArray(10);
> a[1] = true;
> Console.WriteLine(a[1]);
==> true
```

Indexers *cont.*

- Declaration
  - τ `this[`args`] { get {` stats `} set {` stats' `} }`
  - Sugar for

    τ `get_Index(`args`) {` stats `}`

    `void set_Index(`args`,` τ `value) {` stats' `}`
- Access
  - $\mathit{exp}\,[\,\mathit{exps}\,]$ sugar for exp . `get_Index`(exps)
  - $\mathit{exp}\,[\,\mathit{exps}\,] = \mathit{exp}'$ sugar for exp . `set_Index`$(\mathit{exps}, \mathit{exp}')$
- May be virtual
- May be overloaded

Delegates

- Delegate declaration names a function type
- Delegate instance is pair of object pointer and method pointer
- A poor-man's closure:
  - Nominal, not structural
  - Captures state of one object only, no nesting

Delegates *cont.*

- **Haskell**

```
> let j = 1
> in let f = \i -> i + j
> in f 2 + f 3
==> 7
```

- **C#**

```
public delegate int IntToInt(int i);

class J {
  int j;
  public J(int j) { this.j = j; }
  public int add(int i) { return i + j; }
}

> J j = new J(1);
> IntToInt f = new IntToInt(j.add);
> Console.WriteLine(f(2) + f(3));
==> 7
```

Delegates *cont.*

- So, very roughly speaking:
  - Declaration

    ```
    public delegate int IntToInt(int i);
    ```

    is sugar for the (internal) declaration:

    ```
    newtype IntToInt = IntToInt ∃ a . (a, (a, Int) -> Int)
    ```

  - Creation new `IntToInt(j.add)` is sugar for the (internal) term

    ```
    IntToInt (j, add)
    ```

  - Call `f(2)` is abbreviation for the (internal) term

    ```
    case f of IntToInt (o, m) -> m(o, 2)
    ```

- Cf **Java** (using anonymous classes)

```
interface J {
  public int add(int i);
}

> J f = new J {
>   int j = 1;
>   public add(int i) { return i + j; }
> }
> System.out.println(f.add(2) + f.add(3));
==> 7
```

Delegates *cont.*

- Delegate types are reference types
- Instances of delegate types yielding `void` can may *combined* (chained) using +

```
public delegate void IntToVoid(int i);

class K {
  public static void printInt(int i) { Console.WriteLine(i); }
  public static void printSign(int i) {
    Console.WriteLine(i >= 0 ? "+" : "-");
  }
}

> IntToVoid f = new IntToVoid(K.printInt);
> IntToVoid g = new IntToVoid(K.printSign);
> Console.WriteLine((f + g)(1));
==> 1
==> +
```

- May remove delegate instance from combined delegate instance using -
- May also create delegate instances from static methods and other delegate instances

Events

- An *event* field stores a (possibly composite) delegate instance
- *Internally* is just field of delegate type
- *Externally* can be manipulated by `+=` and `-=` operators only
- (Presumably) only delegates types yielding `void` may be used

Events *cont.*

```
public delegate void IntToVoid(int i);

class C {
  int i;
  public event IntToVoid IntHandler;
  public C(int i) { this.i = i; }
  public void test() { IntHandler(i); }
}

class K {
  public static void printInt(int i) { Console.WriteLine(i); }
  public static void printSign(int i) {
    Console.WriteLine(i >= 0 ? "+" : "-");
  }
}

> C c = new C(1);
> c.IntHandler += new IntToVoid(K.printInt);
> c.IntHandler += new IntToVoid(K.printSign);
> c.test();
==> 1
==> +
```

Events *cont.*

- Definition of `+=` and `-=` may be overridden by class:
- Declaration $access$ `event` τ x where τ is a delegate type yielding `void`, is sugar for

  τ x`;`
  $access$ `void add_`x`(`τ `value) {` x `+= value; }`
  $access$ `void remove_`x`(`τ `value) {` x `-= value; }`

- Declaration $access$ `event` τ x `{ add {` $stats$ `} remove {` $stats'$ `} }` where τ is a delegate type yielding `void`, is sugar for

  $access$ `void add_`x`(`τ `value) {` $stats$ `}`
  $access$ `void remove_`x`(`τ `value) {` $stats'$ `}`

- Expression y`.`x `+=` exp is sugar for y`.add_`x`(`exp`)` (etc)

Iterators

- `foreach (`τ x `in` exp`)` $stats$
- Well-typed iff
  - exp has type C
  - C has member E `GetEnumerater()`
  - E has member `bool MoveNext()` and property τ `Current()`
  - $stats$ is well-typed assuming $x : \tau$
- Sugar for

```
C c = exp;
if (c == null) throw NullReferenceException();
E e = c.GetEnumerator();
if (e == null) throw NullReferenceException();
while (e.MoveNext()) {
  τ x = e.Current();
  stats
}
```

Boxing/Unboxing

- May *implicitly* coerce instances of value type to instances of `object`
- May *explicitly* coerce instances of `object` to instances of value type

```
> int i = 42;
> object box = i;
> if (box is int)
>   Console.WriteLine((int)box);
==> 42
```

Boxing/Unboxing *cont.*

- Each value type has corresponding (internal) class declaration.
- Hence above sugar for:

```
class Box_Int {
  int val;
  public Box_Int(int val) { this.val = val }
  public static implicit operator Box_Int(int val)
    { return new Box_Int(val); }
  public static explicit operator int(Box_Int box)
    { return box.val; }
}
```

```
> int i = 42;
> object box = i;
> if (box is Box_Int)
>   Console.WriteLine((int)((Box_Int)box))
```

- However, implicit/explicit conversions in these definitions are considered “built-in” (part of $\leq$) rather than “user-defined” (part of $\leq^i$/$\leq^e$)
- Similarly for enumeration and structure types

Boxing/Unboxing *cont.*

- Any interfaces implemented by a structure declaration are implicitly moved to the coresponding boxed structure class
- Thus

  ```
  struct Point : IPoint {
    public int x;
    public int y;
  }
  ```

  declares structure `Point` (sans interfaces) and additional class

  ```
  class Box_Point : IPoint {
    Point val;
    public Box_Point(Point val) { this.val = val }
    public static implicit operator Box_Point(Point val)
      { return new Box_Point(val); }
    public static explicit operator Point(Box_Point box)
      { return box.val; }
  }
  ```

- Thus implicit conversion from `Point` to `IPoint` will go via `Box_Point`

Threading

- Mostly library-level, but some sugar taken from **Java**
- `lock (`exp`)` $stats$
  - Well-typed if $exp : \tau$ and τ is a reference type
  - Sugar for

```
τ x = exp;
System.Threading.Monitor.Enter(x);
try { stats }
finally { System.Threading.Monitor.Exit(x); }
```

Resources

- `using (`τ x `=` exp`) {` $stats$ `}`
  - Well-typed if τ implements interface

    ```
    interface System.IDisposable {
      void Dispose();
    }
    ```
  - Sugar for

    ```
    τ x = exp;
    try { stats }
    finally { if x != null ((IDisposable)x).Dispose(); }
    ```

Attributes

- Every declaration may be annotated with an *attribute*
- Attributes are instances of classes derived from `System.Attribute`
- These instances available via run-time reflection
- Some built-in attributes for conditional and obsolete methods

```
public class C {
  [Conditional("DEBUG")]
  public void checkConsistent() { ... }

  [HelpString("Prints instance state to Console")]
  public void showState() { ... }
}
```

- Run-time system executes attribute expressions and builds meta-data when assembly loaded