C#/.Net Project Cluster

C# vs. Java

Peter Sestoft
KVL and IT University of Copenhagen

The course materials

- C# Precisely, available from the ITU bookstore.
- Example files at http://www.dina.kvl.dk/~sestoft/csharpprecisely/
- C# 2.0 standard at http://www.dina.kvl.dk/~sestoft/ecma/

Software

- Microsoft Visual Studio 2005 (includes C# compiler and .NET runtime) is installed on classroom computers.
- The free Microsoft Visual C# 2005 Express can be downloaded from http://msdn.microsoft.com/vstudio/express/
- You can use also Mono from http://www.mono-project.com/, especially if you use Linux.

The teacher

MSc and PhD in computer science from DIKU, University of Copenhagen.
Co-developer of the C5 generic collection library for C#/.NET, see http://www.itu.dk/research/c5

Books:

- Jones, Gomard, Sestoft: Partial Evaluation and Automatic Program Generation (Prentice-Hall 1993)
- Sestoft and Hansen: C# Precisely (MIT Press 2004).

Member of the Ecma standardization committees for C# and CLI (Common Language Infrastructure, part of .Net).
What is C#?
A class-based single-inheritance object-oriented programming language with a managed execution environment. Just like Java, but with many improvements — and also with additional complexity. Designed to be the main programming language for Microsoft .Net, alias CLI = Common Language Infrastructure. CLI is the infrastructure of VB.Net, JScript.Net and Managed C++; and of future Microsoft products.

C# has considerable syntactic similarity to Java:
```csharp
using System;
class Hello {
    static void Main(String[] args) {
        Console.WriteLine("Hello," + args[0]);
    }
}
```

C# 2.0 was released as part of Microsoft Visual Studio 2005 in November 2005, for Microsoft Windows only. Mono version 1.1.15 (April 2005) gives a good beta-quality preview of C# 2.0 for Linux and other platforms.

Comparison of Java and C#
Most of C# is immediately recognizable to a Java programmer. Some differences:
- Virtual and non-virtual instance methods.
- Implicit boxing and unboxing of primitive types (also in Java 5.0).
- Properties — field-style method calls.
- Indexers — array-style method calls.
- Enumerators (iterators) and the foreach statement (also in Java 5.0).
- Operator overloading — as in C++.
- Delegates — method closures.
- Value types and structs — allocated on stack, inlined in objects and arrays, copied on assignment and so on.
- Enum types — as in C/C++ (also in Java 5.0).
- Reference parameters (ref and out) — much like Pascal, Ada, C++.
- Variable-arity methods (params modifier; also in Java 5.0).
- No inner classes, no throw clause on methods.
- Unsafe code with pointer arithmetic and so on — discouraged, but possible.

Comparison of Java, C#, C++ and C

<table>
<thead>
<tr>
<th>Feature</th>
<th>Java</th>
<th>C#</th>
<th>C++</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autoboxing value types</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Exception handling</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Array bounds checking</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Classes</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Inheritance</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Multiple inheritance</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Non-virtual methods</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>User-defined methods</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Nested classes</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Call-by-value parameters</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Reference types</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Local variable types</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Inner classes</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Generic types and methods</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Compile-time conditional code</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Generics</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Managed (attribute) annotations</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
</tr>
</tbody>
</table>

Genealogy of some programming languages

![Genealogy of programming languages](image_url)
### Different conventions

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>MyClass</td>
<td>MyClass</td>
</tr>
<tr>
<td>Interfaces</td>
<td>MyInterface</td>
<td>IMyInterface</td>
</tr>
<tr>
<td>Methods</td>
<td>myMethod</td>
<td>MyMethod</td>
</tr>
<tr>
<td>Fields</td>
<td>myField</td>
<td>myField</td>
</tr>
<tr>
<td>Local variables</td>
<td>myVariable</td>
<td>myVariable</td>
</tr>
<tr>
<td>Packages/Namespaces</td>
<td>java.lang.reflect</td>
<td>System.Reflection</td>
</tr>
</tbody>
</table>


### Value types and reference types (C# Precisely section 5.1, examples 84 and 86)

A C# type is either a reference type (class, interface, array type) or a value type (int, double, ...).

- A value (object) of reference type is always stored in the managed (garbage-collected) heap.
- A value of value type is stored in a local variable or parameter, or inline in an array or object or struct value.

Assignment to a variable of reference type copies only the reference.

Assignment to a variable of value type copies the entire value.

Just as in Java. But in C#, there are also user defined value types, called struct types (as in C/C++):

```
struct Frac : IComparable {
    public readonly long n, d; // NB: Meaningful only if d!=0
    public Frac(long n, long d) {
        long f = Gcd(n, d); this.n = n/f; this.d = d/f;
    }
    ... 
    private static long Gcd(long m, long n) { ... }
}
```

A struct type is a value type (C# Precisely section 11 and 14)

Objects are always allocated in the heap. Allocating many small objects is expensive.

In C#, use struct types to represent pairs, fractions, complex numbers and other small values.

A struct type is similar to a class; but has no base type and no virtual methods; may implement interfaces.

Example: Representing fractions n/d:

```
struct Frac : IComparable {
    public readonly long n, d; // NB: Meaningful only if d!=0
    public Frac(long n, long d) {
        long f = Gcd(n, d); this.n = n/f; this.d = d/f;
    }
    ... 
    private static long Gcd(long m, long n) { ... }
}
```

Assigning a struct value copies it, so struct values should usually be immutable: fields should be `readonly`.

### The machine model (C# Precisely example 64)

Class instances (objects) are individuals in the heap.

Struct instances are in the stack, or inline in other structs, objects or arrays.

```
  Stack       Heap
  M2 frame    M2 frame   M2 frame
  1 0
  1 1
  1 2
  sarr
  iarr2
  iarr1
  s
    r 55
      y 555
  x
    r 66
      y 555
  q
  M1 frame
  p
```

```
  Stack     Heap
  M2 frame   M2 frame   M2 frame
  1 0
  1 1
  1 2
  sarr
  iarr2
  iarr1
  s
    r 55
      y 555
  x
    r 66
      y 555
  q
  M1 frame
  p
```

```
  Stack     Heap
  M2 frame   M2 frame   M2 frame
  1 0
  1 1
  1 2
  sarr
  iarr2
  iarr1
  s
    r 55
      y 555
  x
    r 66
      y 555
  q
  M1 frame
  p
```

```
  Stack     Heap
  M2 frame   M2 frame   M2 frame
  1 0
  1 1
  1 2
  sarr
  iarr2
  iarr1
  s
    r 55
      y 555
  x
    r 66
      y 555
  q
  M1 frame
  p
```
Rectangular multi-dimensional arrays (C# Precisely section 9.2.1)

In Java, a 'multi-dimensional' array is a one-dimensional array of arrays. C# in addition has C-style rectangular multi-dimensional arrays.

This improves memory locality (speed) and reduces memory consumption (space).

Example: Two equivalent ways to allocate and initialize a rectangular two-dimensional array:

```csharp
double[,] r1 = { { 0.0, 0.1 }, { 1.0, 1.1 }, { 2.0, 2.1 } };
double[,] r2 = new double[3, 2];
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 2; j++)
        r2[i, j] = i + 0.1 * j;
```

Non-virtual methods (C# Precisely section 10.7 and 10.8)

In Java, a method that is not static or private is automatically virtual.

C# has four kinds of (non-abstract) method declarations; the three first ones are known from C++:

- Static: static void M()
- Non-virtual and non-static: void M()
- Virtual and non-static: virtual void M()
- Explicit interface member implementation: void I.M() — see C# Precisely section 15.3.

A virtual method call is governed by the runtime class of a; a non-virtual method call by the compile-time type.

The override and new keywords are needed (their purpose is to prevent accidental overriding or hiding).

C#: Virtual and non-virtual method example

```csharp
class A {
    public virtual void m1() { Console.WriteLine("A.m1()"); }
    public void m2() { Console.WriteLine("A.m2()"); }
}
class B : A {
    public override void m1() { Console.WriteLine("B.m1()"); }
    public new void m2() { Console.WriteLine("B.m2()"); }
}
class Override {
    static void Main(string[] args) {
        B b = new B();
        A a = b;
        a.m1(); // B.m1()
        b.m1(); // B.m1()
        a.m2(); // A.m2()
        b.m2(); // B.m2()
    }
}
```

Array of arrays (double[][])

Example: Storing the USD/EUR exchange rate for every day in the years 2000–2009.

A 2D array of 1D arrays of double: 10 years times 12 months times a varying number of days.

```csharp
double[,,] rate = new double[10, 12][];
rate[0, 0] = new double[31]; // Jan 2000 has 31 days
rate[0, 1] = new double[29]; // Feb 2000 has 29 days
rate[0, 2] = new double[31]; // Mar 2000 has 31 days
...
rate[0, 1][27] = 0.9748; // 28 Feb 2000
rate[0, 1][28] = 0.9723; // 29 Feb 2000
rate[0, 2][0] = 0.9651; // 1 Mar 2000
```
Reference parameters (C# Precisely sections 10.7, 12.15.2; examples 84, 86)

Call-by-value is similar to assignment: copies the argument value to the method parameter.

Call-by-reference makes the method parameter refer directly to the argument.

The argument must be a variable or field or array element (that is, must have an lvalue).

```csharp
void swapV(int x, int y) {
    int tmp = x; x = y; y = tmp;
}

void swapR(ref int x, ref int y) {
    int tmp = x; x = y; y = tmp;
}

a = 11; b = 22;
swapV(a, b);
```

Reference parameters, continued

A reference parameter is declared using the ref or out modifier:

- A ref parameter is used for input and/or output. Using a ref parameter may avoid copying arguments at parameter passing.
- An out parameter is used for output only. Using an out parameter, a method may return multiple values.

Example: Return true if removal succeeded and use out parameter to return the removed value:

```csharp
class TreeDictionary<K, V> {
    bool Remove(K key, out V val);
    ...
}
```

Example use:

```csharp
TreeDictionary<String, Person> students = ...;
Person res;
if (students.Remove("Ulrik Funder", out res))
    res.MailTo("You have been thrown out of university");
else
    ShowMsg("Student not found");
```

Properties (C# Precisely section 10.13, 12.16)

Fields are usually made private. Then getP and setP methods are often used to get and set fields.

In C#, such methods are called properties. They have method-like declarations and field-like call syntax.

Example: A last-in first-out stack of doubles:

```csharp
class Lifo {
    private double[] stack = new double[10]; // Holds stack elements
    private int sp = -1; // Points to stack top
    public void Push(double x) { stack[++sp] = x; }
    public double Pop() { return stack[sp--]; }
    public int Count { // Read-only property
        get { return sp + 1; }
    }
    public double Top { // Read-write property
        get { return stack[sp]; }
        set { stack[sp] = value; }
    }
}
```

Example use:

```csharp
Lifo lifo = new Lifo();
lifo.Push(2.3); lifo.Push(5.4);
Console.WriteLine(lifo.Count); // 2
Console.WriteLine(lifo.Top);   // 5.4
lifo.Top = 6.7;
```

Indexers (C# Precisely section 10.14, 12.17)

An indexer permits array-like method calls.

Example: Last-in first-out stack; indexing relative to the stack top:

```csharp
class Lifo {
    private double[] stack = new double[10]; // Holds stack elements
    private int sp = -1; // Points to stack top
    ...
    public double this[int i] { // Read-write indexers
        get { return stack[sp - i]; }
        set { stack[sp - i] = value; }
    }
}
```

Example use:

```csharp
Console.WriteLine(lifo[0]);
lifo[1] = 7.8;
```

The indexers of a class or struct are distinguished only by their signature (parameters).
Operator overloading (C# Precisely section 10.15)

As in C++, operators (+, *, <, ==,...) can be overloaded.

An overloaded operator is a public static method. At least one argument must have the enclosing type.

Operators work well with value-oriented types, such as struct types; no risk of arguments being null.

Example: Arithmetic operators on fractions:

```csharp
struct Frac : IComparable
{
    public static Frac operator+(Frac r1, Frac r2) {
        return new Frac(r1.n*r2.d+r2.n*r1.d, r1.d*r2.d);
    }
}
```

Delegates and delegate types (C# Precisely section 17)

A delegate encapsulates a (static or non-static) method. A delegate type is a method type.

Example: Declare IntPredicate as type of method that takes an int argument and returns a bool result:
```csharp
delare bool IntPredicate(int x);
```

Example: Methods Even and PrintBig has that type:
```csharp
static bool Even(int x) { return x%2 == 0; }
static bool PrintBig(int x) { Console.WriteLine(x); return x > 100; }
```

Therefore method Even can be wrapped as an IntPredicate delegate object:
```csharp
IntPredicate m = new IntPredicate(Even);
Console.WriteLine(m(4)); // Prints: True
```

A delegate is multicast: it can encapsulate more than one method, and more than one instance of a method:
```csharp
m += new IntPredicate(PrintBig);
m += new IntPredicate(PrintBig);
Console.WriteLine(m(7)); // Prints: 7 7 False
```

In Java, there are no delegates; instead a method in an instance of an anonymous inner class may be used.

User-defined conversions (C# Precisely section 10.16)

A user-defined conversion is an operator named by the conversion’s target type.

Conversions may be implicit (require no cast) or explicit (require a type cast).

Converting between integers, doubles and fractions is useful:
```csharp
struct Frac : IComparable {
    public readonly long n, d;
    public Frac(long n, long d) { ... }
    public static implicit operator Frac(int n) { return new Frac(n, 1); }
    public static implicit operator Frac(long n) { return new Frac(n, 1); }
    public static explicit operator long(Frac r) { return r.n/r.d; }
    public static explicit operator float(Frac r) { return (float)r.n/r.d; }
    ...
}
```

Example of user-defined conversions:
```csharp
Frac f1 = (byte)5;  // Implicit int-->Frac
Frac f2 = 1234567890123L;  // Implicit long-->Frac
int i1 = (int)f1;  // Explicit Frac-->int
double d2 = (double)f2;  // Explicit Frac-->float
```

An implicit conversion should lose no information and throw no exceptions; an explicit conversion may.

Enum types (C# Precisely section 16)

Unlike in Java, an enum value is really an integer, and enum types cannot have methods.

The integer value of an enum member can be given explicitly.

Example: Month codes 1–12:
```csharp
public enum Month {
    Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
}
```

Unlike in Java, enum values admit arithmetic operations such as ++:
```csharp
public static Date FromDaynumber(int n) {
    ... Month m = Month.Jan;
    int mdays;
    while ((mdays = MonthDays(y, m)) < d) {
        d -= mdays;
        m++;
    }
    return new Date(y, m, d);
}
```

Unlike in Java, computing with an enum type may produce an integer value not corresponding to an enum value.
The **foreach** statement (C# Precisely section 13.6.2)

Similar to Java's enhanced for statement.
Permits iteration over types that implement `IEnumerable` or `IEnumerable<T>`, including arrays.
Later we shall see how to easily write enumerable types.

Example: Truncating and summing the elements of an array:

```csharp
double[,] arr = { { 9.213, 91.345, 410.0, 323.5, 930.25 } };  
int sum = 0;  
foreach (double v in arr)  
    sum += v;  
```

The **foreach** loop is equivalent to a while-loop over an enumerator, followed by a call to **Dispose**:

```csharp
IEnumerator enm = arr.GetEnumerator();  
try {  
    while (enm.MoveNext()) {  
        int v = (int)(double)enm.Current;  // Explicit cast, not in Java  
        sum += v;  
    }  
} finally {  
    IDisposable disp = enm as System.IDisposable;  
    if (disp != null) disp.Dispose();  
}
```

The **IDisposable** interface (C# Precisely section 13.10)

```csharp
interface IDisposable {  
    void Dispose();  
}
```

The **Dispose** method should release resources held by an enumerator or file or other object.
This permits early explicit resource deallocation, when there is no need to wait for the garbage collector.

The **using** statement (C# Precisely section 13.10)

C#'s **using** statement:

```csharp
using (t x = e)  
    body
```

is equivalent to:

```csharp
{  
    t x = e;  
    try { body }  
    finally { if (x != null) ((IDisposable)x).Dispose(); }  
}
```

Creation of some resource by `x = e` and its deallocation by `x.Dispose()` are textually linked.
A simple but useful idiom.

Variable-arity methods (C# Precisely section 10.7 and 12.15.3)

As in Java, the last parameter of a method may be a parameter array, thus permitting variable-arity methods.
The C# declaration syntax is `(params t[] xs)` where in Java is `(t... xs)`.

Example: A `Max` method taking one or more arguments, and a `Max` method taking two:

```csharp
static int Max(int x1, params int[] xr) {  
    int res = x1;  
    foreach (int x in xr)  
        res = Max(res, x);  
    return res;  
}  
static int Max(int x, int y) {  
    return x > y ? x : y;  
}
```

Overloading resolution prefers the explicit two-argument method over the one-or-more argument method.
Example uses:

```csharp
Console.WriteLine(Max(69, 42));  // Max(int,int)  
Console.WriteLine(Max(2, 5, 7, 11, 3));  // Max(int,int[])  
Console.WriteLine(Max(2, new int[] {5, 7, 11, 3}));  // Max(int,int[])  
```

Namespaces (C# Precisely section 25)

In Java:

- Types can be organized in packages, which have accessibility effect (default accessibility).
- The `import` declaration is used to access another package.
- The source file hierarchy reflects the package structure.

In C#, an analogous concept is the namespace and the `using` declaration.

But namespaces are not reflected in the source code structure.

And namespaces have no accessibility effect: the **internal** modifier concerns compilation units.

```csharp
using System;  
namespace N1 {  
    public class C11 { N3.C31 c31; }  // N1 depends on N3  
namespace N2 {  
    public class C121 {  
    }  
}  
}  
namespace N1 {  
    public struct S13 { }  // Default accessibility: internal  
namespace N1.N2 { internal class C122 { } }  
namespace N3 { class C31 { N1.C11 c11; } }  // N3 depends on N1
```

```
ITU May 2006  
C# .NET Project Cluster  
C# vs. Java-25
```
String formatting (C# Precisely section 7.2)

C# has a rich and convenient machinery for formatting strings, numbers, dates, and times.

Example: Assume $freq[c]$ is the number of times rolling two dice gave $c$ eyes.

Basic printing, using $\{0\}$ to specify first value, and so on:

```csharp
for (int $c=2; c<=12; c++)
    Console.WriteLine("\{0\} came up \{1\} times", $c$, $freq[c-1]$);
```

Specifying formatting widths 2 and 4 using $\{0,2\}$ and $\{1,4\}$:

```csharp
for (int $c=2; c<=12; c++)
    Console.WriteLine("\{0,2\} came up \{1,4\} times", $c$, $freq[c-1]$);
```

The two outputs:

<table>
<thead>
<tr>
<th>Roll</th>
<th>Frequency</th>
<th>Width 2</th>
<th>Width 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>264</td>
<td>264</td>
<td>264</td>
</tr>
<tr>
<td>3</td>
<td>572</td>
<td>572</td>
<td>572</td>
</tr>
<tr>
<td>4</td>
<td>787</td>
<td>787</td>
<td>787</td>
</tr>
<tr>
<td>5</td>
<td>1188</td>
<td>1188</td>
<td>1188</td>
</tr>
<tr>
<td>6</td>
<td>1426</td>
<td>1426</td>
<td>1426</td>
</tr>
<tr>
<td>7</td>
<td>1723</td>
<td>1723</td>
<td>1723</td>
</tr>
<tr>
<td>8</td>
<td>1353</td>
<td>1353</td>
<td>1353</td>
</tr>
<tr>
<td>9</td>
<td>1086</td>
<td>1086</td>
<td>1086</td>
</tr>
<tr>
<td>10</td>
<td>767</td>
<td>767</td>
<td>767</td>
</tr>
<tr>
<td>11</td>
<td>558</td>
<td>558</td>
<td>558</td>
</tr>
<tr>
<td>12</td>
<td>266</td>
<td>266</td>
<td>266</td>
</tr>
</tbody>
</table>

Plain database access via ODBC (C# Precisely example 194)

Assume the server `sql.dina.kvl.dk` has a user `sestoft` with a database `test`.

We can connect to that database as follows:

```csharp
String setup =
    "DRIVER=\{MySQLODBC3.51Driver\};" +
    "SERVER=sql.dina.kvl.dk;" +
    "DATABASE=test;" +
    "UID=sestoft;" +
    "PASSWORD=........;";
using (OdbcConnection conn = new OdbcConnection(setup)) {
    conn.Open();
    // query the database (next slide) ...
}
```

Assume that database table has a table `Message`:

<table>
<thead>
<tr>
<th>name</th>
<th>msg</th>
<th>severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>lunch</td>
<td>30</td>
</tr>
<tr>
<td>Kasper</td>
<td>day off</td>
<td>10</td>
</tr>
<tr>
<td>Dan</td>
<td>coffee break</td>
<td>20</td>
</tr>
</tbody>
</table>

Quering the database and reading result sets

Using the connection object `conn` previously obtained:

```csharp
conn.Open();
String query = "SELECT name, msg, severity FROM Message ORDER BY name";
OdbcCommand cmd = new OdbcCommand(query, conn);
OdbcDataReader r = cmd.ExecuteReader();
while (r.Read()) {
    String name = r.GetString(0);
    String msg = r.GetString(1);
    int severity = r.GetInt32(2);
    Console.WriteLine("\{0\} \{1\} (\{2\})", name, msg, severity);
}
r.Close();
```

This is very similar to Java’s JDBC-bindings.

There’s a far more elaborate machinery in ADO.NET; see the classes in namespace `System.Data`.

The approach shown above is hard to maintain, and it is difficult to make sure that C# code and database agree.

In the future (around 2008) database access from .Net will have much better compile-time safety.

Namely, C# 3.0 and VB.NET 9 will have Language Integrated Query (Linq), see http://msdn.microsoft.com/netframework/future/linq/.

Command line compilation and execution of C# programs (C# Precisely section 1)

To edit C# programs, use Visual C# Express 2005, or any editor (Emacs, UltraEdit, Notepad,...).

C# programs can be compiled and run from Visual C# Express, or from a .Net Framework SDK Command Prompt.

To open a .Net Framework SDK Command Prompt, choose

Start
--- All Programs
--- Microsoft .Net Framework SDK v2.0
--- .Net Framework SDK Command Prompt

To compile a C# program in file `Foo.cs`, type

```bash
csc Foo.cs
```

This creates a file `Foo.exe` — looks like a classic x86 executable, but isn’t. It requires .Net 2.0.

To execute the compiled program, type

```bash
Foo
```