C# 3.0 and 4.0 new features

Peter Sestoft
Wed 2011-02-02*
and
Mon 2011-02-07

Outline

• C# versions and standards
• New language features
  – Inferred variable types
  – Lambda expressions
  – Extension methods
  – Implicitly typed array creation
  – Anonymous object creation, anonymous types
  – Object initializers, collection initializers
• Linq: Language integrated query
  – On enumerables
  – On relational data
  – Support: Expression trees
• Named arguments and optional parameters
• Dynamically typed expressions
• Co-variant and contra-variant type parameters
• Task Parallel Library
C# and .NET versions

- .NET platform
  - Languages C#, VB.NET, JScript.NET, F#, ...
  - Class libraries
  - Bytecode and runtime system CLR
  - Development environment Visual Studio
- 2001: C# 1.0 and .NET 1.0
- 2003: C# 1.1 and .NET 1.1
- 2005: C# 2.0 and .NET 2.0
- 2006: .NET 3.0 (new: WPF, WCF, WF)
- 2007: C# 3.0 and .NET 3.5
- 2010: C# 4.0 and .NET 4.0

Linq drove C# 2.0 -> 3.0 evolution

- The goal of C# 3.0 is to support a type-safe embedded query language
- Previous experimental languages Xen and Cω by Erik Meijer and others
- Based on Haskell comprehensions and Kleisli query language (Buneman 1993)
- Linq is a uniform query language for collections, XML, relational databases
- Design by Hejlsberg, Meijer, Torgersen, ...
- Since July’08 Mono also implements C# 3.0
- Since Oct'10 Mono also implements C# 4.0
Inferred local variable types

• Initialized variable declaration may omit type

```csharp
var h = 3;    // Inferred: int
var z = 3*2.0;    // Inferred: double
var y = h > 2 ? h : z;  // Inferred: double
var a = new double[] { h, 17 };  // double[]
var dict = new Dictionary<int, string>();
```

• Very useful to avoid repetition:

```csharp
Dictionary<int, string> dict = new Dictionary<int, string>();
```

• Still static typing, **not** dynamic typing:

```csharp
var x = 3;
x = "foo";  // NO! Rejected by compiler
```

Inferred types in other statements

- `foreach (var x in ...)` ...
- `for (var x = ...; ...; ...)` ...
- `using (var x = ...)` ...

• **No** type inference for
  - Fields of classes and structs
  - Method parameters
  - Method return type
  - Exception variables, as in `catch (MyExn e) ...`
  - Const declarations, as in `const var y = 21;`
Anonymous functions (lambdas)

- New anonymous method (delegate) syntax:

  ```csharp
  delegate (int x) { return x%2==0; }
  (int x) => x%2==0
  x => x%2==0
  ```

  Same meaning

  Type inferred

.NET generic delegate types

<table>
<thead>
<tr>
<th>Action</th>
<th>Act</th>
<th>unit -&gt; unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action&lt;A1&gt;</td>
<td>Act&lt;A1&gt;</td>
<td>A1 -&gt; unit</td>
</tr>
<tr>
<td>Action&lt;A1,A2&gt;</td>
<td>Act&lt;A1,A2&gt;</td>
<td>A1*A2 -&gt; unit</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Func&lt;R&gt;</td>
<td>Fun&lt;R&gt;</td>
<td>unit -&gt; R</td>
</tr>
<tr>
<td>Func&lt;A1,R&gt;</td>
<td>Fun&lt;A1,R&gt;</td>
<td>A1 -&gt; R</td>
</tr>
<tr>
<td>Func&lt;A1,A2,R&gt;</td>
<td>Fun&lt;A1,A2,R&gt;</td>
<td>A1*A2 -&gt; R</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>


worst?  best?
**A little functional programming**

- A method to compose a function with itself

```csharp
public static Func<T, T> Twice<T>(Func<T, T> f) {
    return x => f(f(x));
}
```

- Some lambdas and computed functions

```csharp
var fun1 = Twice<int>(x => 3 * x);
Func<int, int> triple = x => 3 * x;
var fun2 = Twice(triple);
Func<Func<int, int>, Func<int, int>> twice = f => x => f(f(x));
var fun3 = twice(triple);
var res = fun1(4) + fun2(5) + fun3(6);
```

**Type system strangeness 1**

- Cannot infer type of anonymous function
  - Essentially, most general type is not expressible

```csharp
var triple = x => 3 * x;
var twice = f => x => f(f(x));
```

- But type inference from arguments

```csharp
static Func<T, T> Twice<T>(Func<T, T> f) { ... }
Twice(delegate(int x) { return 3 * x; })
```

The delegate is used to infer T=int in Twice<T>, wasn’t so in C#2.0
Extension methods

- Can be “added” to any existing type
- Are defined separately in a static class

```csharp
static class BoolExtensions {
    public static void Print(this bool b) {
        Console.WriteLine(b ? "ja" : "nej");
    }
}
```
- Are called like instance methods

```
(1==1).Print();
```

Extension methods as syntactic sugar

- A call to an extension method

```csharp
public static void M(this C x, int y) { ... }
```
```
o.M(42)
```

is just a static call with receiver as first arg:
```
M(o, 42)
```

- **Not** the same as an instance method call
- Especially not on struct types
**Week numbers (1-53) on DateTime**

- Let’s teach .NET about ISO week numbers

```csharp
public static int IsoWeek(this DateTime dt) {
    int yday = dt.DayOfYear - 1, wday = IsoWeekDay(dt), y = dt.Year;
    const int THU = 3;
    ... 10 lines of code omitted ...
}
```

- Use it like any System.DateTime method:

```csharp
int thisWeek = DateTime.Today.IsoWeek();
```

**More DateTime extension methods**

- Create sequence of dates a week apart

```csharp
public static IEnumerable<DateTime>
    Weekly(this DateTime dt, int n)
    {
        for (int added = 0; added < n; added++)
            yield return dt.AddDays(added * 7);
    }
```

```csharp
foreach (var dt in DateTime.Today.Weekly(52))
    Console.WriteLine(dt.ToShortDateString());
```

2008-08-13
2008-08-20
2008-08-27
...
Extension method on type instance

- A useful function on String arrays

```csharp
static class StringArrayExtensions {
    public static String ConcatWith(this String[] arr, String sep) {
        StringBuilder sb = new StringBuilder();
        if (arr.Length > 0) {
            sb.Append(arr[0]);
            for (int i = 1; i < arr.Length; i++) {
                sb.Append(sep).Append(arr[i]);
            }
            return sb.ToString();
        }
    }
}
```

- Call it like it was part of the .NET classes:

```csharp
String[] sarr = { "www", "edlund", "dk" };
sarr.ConcatWith(".");
```

value "www.edlund.dk"

Extension method on generic type

- E.g. define method on Func<A,A> for any A
- The method must be generic:

```csharp
static class FuncExtensions {
    public static Func<A,A> Twice<A>(this Func<A,A> f) {
        return x => f(f(x));
    }
}
```

```csharp
Func<int,int> triple = x => 3*x;
int res = triple.Twice()(7);
```
Extension methods on interfaces

public static int MyCount<T>(this IEnumerable<T> xs) {
    int count = 0;
    foreach (T x in xs)
        count++;
    return count;
}

• Can be used on all implementing types

double[] darr = { 4.5, 1.6, 5.6, 7.9 };
Console.WriteLine(darr.MyCount());
List<int> list = ...
Console.WriteLine(list.MyCount());

• Lots of extension mths on IEnumerable<T>

... and with type parameter constraint

public static bool IsSorted<T>(this IEnumerable<T> xs)
    where T : IComparable<T>
    {
        var etor = xs.GetEnumerator();
        if (elor.MoveNext()) {
            T prev = etor.Current;
            while (elor.MoveNext())
                if (prev.CompareTo(elor.Current) > 0)
                    return false;
                else
                    prev = etor.Current;
        }
        return true;
    }

double[] darr = { 4.5, 1.6, 5.6, 7.9 };
Console.WriteLine(darr.IsSorted());
Extension method subtleties 1

- An extension method is non-virtual
- So cannot be used to override `Object.ToString()` which is virtual:

```csharp
static class StringArrayExtensions {
    public static String ToString(this String[] arr) {
        ...
    }
}
```

- No "extension properties"
- No "extension indexers"

Extension method subtleties 2

- An instance call on a struct passes the struct by reference, so its fields can be updated
- An extension method call passes it by value
- So extension method is not an instance method

```csharp
struct MyStruct {
    internal int x;
    public void RealIncrement() {
        x++;  
    }
}
static class MyStructExtensions {
    public static void UselessIncrement(this MyStruct b) {
        b.x++;  
    }
}
```

By value: Pass a *copy* of struct

No effect on b.x
Controlling extension method scope

• Declare extension methods in a namespace

```csharp
namespace MyExtensions {
    public static class MyExtensionsClass {
        public static void MyMethod(this My my) {...}
    }
}
```

• Use namespaces and `using` to control scope

```csharp
namespace N1 {
    using MyExtensions;
    // My.MyMethod usable here
}
namespace N2 {
    // My.MyMethod not usable here
}
```

A very applicable extension method

• Be careful what extension methods you get in scope from a “foreign” namespace
• Be careful what classes have extension mths

```csharp
public static class ObjectExtensions {
    public static void toString(this Object receiver, params Object[] args) {
        Console.WriteLine("ToString, idiot!");
    }
}
```

Defined on all types

Matches all `toString` signatures
Anonymous object creation

- An anonymous object has named fields
  ```csharp
double z = 3.14;
var p1 = new { x = 13, y = "foo" };
var p2 = new { x = 42, p1.y, z };
var p3 = new { };
```
- The “field” is a readonly property
- Field name is given or inferred: y, z above
- Anon. objects compare and display neatly:
  ```csharp
  { x = 42, y = foo, z = 3.14 }
  ```
- Like SML record `{ x=42, y="foo", z=3.14 }`

Type system strangeness 2

- Anon object expressions have same type **iff** same fields of same type in same order
- But the type cannot be expressed in C#
  - Hence anon obj useful only due to type inference
- Only conversion to Object, not any interface
  - So useless for a method to return anonymous obj
  ```csharp
  Object ReturnAnonymousObject() {
    return new { x=4, y=5 };
  }
  ```
**Typing by example (Tomas Petricek)**

- Use type param to represent anon type
  
  ```csharp
  T AnonymousHack<T>(T dummy) {
      return (T)(object)(new { x=4, y=5 });
  }
  ```

- Pass a dummy value of the appropriate type
  
  ```csharp
  var res = AnonymousHack(new {x=0,y=0});
  Console.WriteLine(res.x+1);
  ```

  ... so res is known to have field x of type int

  Type inference binds T to anon type ...

**Implicitly typed array creation**

- Small type system improvement
- Can write
  
  ```csharp
  var arr1 = new[] { 4, 3.14 };
  ```

- Instead of
  
  ```csharp
  var arr1 = new double[] { 4, 3.14 };
  ```

- Infers “best common type” for array elems
- Essential for arrays of anonymous objects
  
  ```csharp
  var arr2 = new[] { new { x=3, y=4 },
                      new { x=6, y=7 } };
  foreach (var r in arr2) ... r.x ...
  ```
Field with automatic properties

- Declares field plus get and set properties

```csharp
class Account {
    public decimal Balance { get; set; }
}
```

```csharp
Account acc = ...;
acc.Balance = 23000M;
Console.WriteLine(acc.Balance);
```

- Can change class without breaking clients
- Can restrict accessibility individually:

```csharp
class Account {
    public decimal Balance { get; private set; }
}
```

Object initializer

- New syntax for creation+initialization
- Works for accessible fields and properties

```csharp
struct Point {
    private int _x;
    internal int Y;
    internal int X { get { return _x; } set { _x = value; } }
}
```

```csharp
Point p = new Point { X = 4, Y = 5 };
```

```csharp
Point p = new Point();
p.X = 4;
p.Y = 5;
```

Same meaning
**Collection initializer**

- Compiles to `Add(...)` method calls
- So work on third-party collections (C5) too
- Overloading resolution works as usual

```csharp
var list2 = new List<int>
    { { 2 }, { 3 }, { 2+3 }, { 5+2 } };

var list2 = new List<int>();
list2.Add(2);
list2.Add(3);
list2.Add(2+3);
list2.Add(5+2);
```

**Linq, language integrated query**

- Linq in C#:
  ```csharp
  from x in primes where x*x < 100 select 3*x
  ```

- Set comprehensions, ZF notation:
  ```
  { 3x | x ∈ primes, x^2 < 100 }
  ```

- List comprehensions Miranda (1985), Haskell
  ```
  [ 3*x | x <- primes, x*x < 100 ]
  ```
From queries to method calls

- A query such as
  
  ```csharp
  from x in primes where x*x < 100 select 3*x
  ```
  
is transformed to an ordinary C# expression:

  ```csharp
  primes.Where(x => x*x < 100)
  .Select(x => 3 * x)
  ```

- Only this expression gets type checked
- Where and Select may be extension methods

Basic extension methods for Linq

```csharp
IEnumerable<T> Where<T>(this IEnumerable<T> xs, Func<T,bool> p)
```

- As list comprehension:
  
  ```
  [ x | x <- xs, p(x) ]
  ```

```csharp
IEnumerable<U> Select<T,U>(this IEnumerable<T> xs, Func<T,U> f)
```

- As list comprehension:
  
  ```
  [ f(x) | x <- xs ]
  ```
Extension methods on IEnumerable

- SCG.IEnumerable<T> used to be simple:

```csharp
interface IEnumerable<T> {
    IEnumerator<T> GetEnumerator();
}
```

- Now these extension methods + 60 others:

```csharp
bool All<T>(this IEnumerable<T> xs, Func<T, bool> p)
bool Any<T>(this IEnumerable<T> xs, Func<T, bool> p)
bool Contains<T>(this IEnumerable<T> xs, T x)
int Count<T>(this IEnumerable<T> xs)
IEnumerable<T> Distinct(this IEnumerable<T> xs)
T First<T>(this IEnumerable<T> xs)
T Single<T>(this IEnumerable<T> xs)
IEnumerable<T> Union<T>(this IEnumerable<T> xs, IEnumerable<T> ys)
... 
```

- Most support Linq for collections
- But an enumerable is nearly a lazy list, so they also support functional programming
- (Except no caching of computed items)

```csharp
double sum = Enumerable.Range(1, 200)
    .Where(x => x % 5 != 0 && x % 7 != 0)
    .Select(x => 1.0 / x)
    .Sum();
```

```csharp
double sum =
    (from x in Enumerable.Range(1, 200)
     where x % 5 != 0 && x % 7 != 0
     select 1.0 / x).Sum();
```

**Same**
Performance of LINQ-style queries

• Query is only 1.7 times slower than this loop:

```csharp
double sum = 0.0;
for (int x=1; x<=200; x++)
    if (x%5!=0 && x%7!=0)
        sum += 1.0/x;
```

• For near-trivial loop contents, a factor 11.2:

```csharp
(from x in Enumerable.Range(1, 200)
    where (x & 1) == 0
    select 1 + x).Sum();
```

<table>
<thead>
<tr>
<th>.NET 3.5 SP1</th>
<th>7031 ns</th>
</tr>
</thead>
</table>
| double sum = 0.0;
for (int x=1; x<=200; x++)
    if ((x & 1) == 0)
        sum += 1 + x; | 625 ns |

The form of a query expression 1

```
from x in xs
query-body-clauses
select e
from y in ys
query-body-clauses
let x = e
where b
join y in ys on e1 equals e2
join y in ys on e1 equals e2 into z
orderby e1, e2, ...
query-body-clauses
```
The form of a query expression 2

- A query can have continuations, of form:

  ```
  ... into z
  query-body-clauses
  select e
  ... into z
  query-body-clauses
  group e by e
  ```

Some differences from SQL

- No DISTINCT in select, instead (from ... select ...).Distinct()
- UNION, INTERSECT, EXCEPT done like this (from ... select ...).Union(from ... select ...)
- No HAVING, instead 
  ... group ... into g where ... g ...
- COUNT done like this 
  (from ... select ...).Count()
- SUM, AVG, MIN, MAX done like this 
  (from ... select ...).Sum()
  - These work on a few numerical and nullable numerical types; MIN, MAX also on ordered types
What do they collect this week?

- Even weeks: garden waste; odd: recycling
- Friday (e.g. 15 Aug 2008), except holidays

```csharp
var collect =
    from dt in new DateTime(2008, 8, 15).Weekly(520)
    where !dt.IsHoliday()
    select new { dt, art=dt.IsoWeek()%2==0 ? "Haveaffald" : "Storskrald" };

foreach (var x in collect)
    Console.WriteLine(x);
```

- May have two odd weeks (53 and 1) in a row

When are two weeks in a row odd?

```csharp
var gentagelse =
    from dt in new DateTime(2010, 1, 29).Weekly(5200)
    let dtNext = dt.AddDays(7)
    where (dt.IsoWeek()%2==0) == (dtNext.IsoWeek()%2==0)
    select new { dt, dtNext };

foreach (var x in gentagelse)
    Console.WriteLine(x);
```

- Uses the let-clause
  - Normally used to avoid repeated computation
  - E.g. using result of aggregate in further query:

```csharp
var categories =
    from p in products
    group p by p.Category
    into g
    let min = g.Min(p => p.UnitPrice)
    select new {Category = g.Key, Cheapest = g.Where(p => p.Price == min)};
```
**Translation of let**

- The query
  
  ```csharp
  from x in xs
  let y = f ...
  ```

  is expanded to
  
  ```csharp
  from * in (xs.Select(x => new { x, y=f }));
  ```

  New sequence of pairs (x, y)

---

**Weekday distribution of holidays**

```csharp
var holidayWeekDays =
  from dt in DateTimeExtensions.holidays.Keys
  group dt by dt.DayOfWeek
  into g
  select new { g.Key, Count = g.Count() };
```

- `Key = Tuesday, Count = 15`
- `Key = Sunday, Count = 95`
- `Key = Thursday, Count = 60`
- `Key = Friday, Count = 61`
- `Key = Monday, Count = 52`
- `Key = Wednesday, Count = 17`
- `Key = Saturday, Count = 18`
Translation of group by

- The query
  ```csharp
  from x in xs group x by e
  ```
is expanded to
  ```csharp
  xs.GroupBy(x => e)
  ```
- The query
  ```csharp
  from x in xs group x.Name by x.Age
  ```
is expanded to
  ```csharp
  xs.GroupBy(x => x.Name, x => x.Age)
  ```

Extension methods for grouping

```csharp
IEnumerable<IGrouping<K,T>>
GroupBy<T,K>(this IEnumerable<T> xs, Func<T,K> h)
```

- As list comprehension
  - Compute ks = distinct([ h(x) | x <- xs ])
  - Return [ (k, [ x | x <- xs, h(x)=k ]) | k <- ks ]
- A grouping is an enumerable with a key:

```csharp
interface IGrouping<K,T> : IEnumerable<T> {
  K Key { get; }
}
```

- There are several additional overloads
Translations of join

- The query
  
  ```csharp
  from x in xs
  join y in ys on kx equals ky
  select e
  ```

  is expanded to
  
  ```csharp
  xs.Join(ys, x => kx, y => ky, (x,y) => e)
  ```

- The query
  
  ```csharp
  from x in xs
  join y in ys on kx equals ky into z
  select e
  ```

  is expanded to
  
  ```csharp
  xs.GroupJoin(ys, x => kx, y => ky, (x,z) => e)
  ```

Extension methods for join 1

```csharp
IEEnumerable<V> Join<T,U,K,V>(this IEEnumerable<T> xs,
IEEnumerable<U> ys,
Func<T,K> fx,
Func<U,K> fy,
Func<T,U,V> g)
```

- As list comprehension:
  
  ```csharp
  [ g(x,y) | x <- xs, y <- ys, fx(x)=fy(y) ]
  ```

- Efficient even on enumerables (I guess):
  - Make multidictionary fx(x) -> x for all x in xs
  - For each y in ys compute fy(y), look up matching x values, for each of them yield g(x,y)
Extension methods for join 2

```csharp
IEnumerable<V> GroupJoin<T,U,K,V>(
    this IEnumerable<T> xs,
    IEnumerable<U> ys,
    Func<T,K> fx,
    Func<U,K> fy,
    Func<T,IEnumerable<U>,V> g)
```

- As list comprehension:
  
  ```
  [ g(x, [y | y <- ys, fx(x)=fy(y) ]) | x <- xs ]
  ```

Order weekday distribution by weekday

```csharp
var holidayWeekDays =
    from dt in DateTimeExtensions.holidays.Keys
    group dt by dt.DayOfWeek
    into g
    orderby g.Key
    select new { g.Key, Count = g.Count() };
```

```csharp
{ Key = Sunday, Count = 95 }  
{ Key = Monday, Count = 52 }  
{ Key = Tuesday, Count = 15 }  
{ Key = Wednesday, Count = 17 }  
{ Key = Thursday, Count = 60 }  
{ Key = Friday, Count = 61 }  
{ Key = Saturday, Count = 18 }  
```
Translation of orderby

• The query

```csharp
from x in xs
orderby k1, k2, ...
```

is expanded to

```csharp
xs.OrderBy(x => k1)
 .ThenBy(x => k2)
 . . .
```

Extension methods for ordering

```csharp
IOrderedEnumerable<T>
 OrderBy<T,K>(this IEnumerable<T> xs, Func<T,K> h)
```

• Order xs by ascending h(x)

• An ordered enumerable
  – remembers its previous ordering criteria
  – supports ordering by further (secondary) criteria
  while respecting previous criteria

```csharp
IOrderedEnumerable<T>
 ThenBy<K>(this IOrderedEnumerable<T> xs, Func<T,K> h)
```
Query execution

- The `IEnumerable` extension methods are lazy
- So a query is executed only when – and every time – the result is demanded:

```csharp
int[] numbers = new int[] { 5, 4, 1, 3, 9 }; int j = 0;
var q = from n in numbers select new { n, i = ++j };
foreach (var v in q) Console.WriteLine(v);
foreach (var v in q) Console.WriteLine(v);
```

Expression trees

- A reflective representation of expressions
- Covers C# expressions except assignment
- A function can be called but not inspected:

```csharp
Func<int, int> f = x => 3 * x;
int res = f(7);
Expression<Func<int, int>> t = x => 3 * x;
```

- An expression tree can be inspected:

```csharp
Expression<Func<int, int>> t = x => 3 * x;
```

$t = \begin{cases} x & \Rightarrow \ \\
3 & \star \\
x & \end{cases}$

Abstract syntax for lambda
$x \Rightarrow 3 \ast x$
From lambda to expression tree

• A lambda may convert to expression tree at assignment to variable, field or parameter

```csharp
bool Where(Expression<Func<int,bool>> p) {...}

bool foo = Where(z => z>42);
```

• `p` will be a tree, not a function
• The tree may be analysed by the `Where` method

---

Linq to relational databases

• For Linq to collections (or in-memory XML), enumerable extension methods are efficient
• For Linq to relational database, they aren’t
• Instead,
  – the query gets rewritten to method calls, as usual
  – the `System.Linq.Data.Table.Where` method is `Where(Expression<Func<T,bool>> p)`
  – it captures the predicate `p` as an expression tree and rewrites it to an SQL fragment
  – same for `Select`, `Join`, `GroupBy`, `OrderBy`, `ThenBy`
  – the DB server executes SQL and returns results
• Works even if the delegates involve local (client-side) computation, but may be slow
**Linq samples in VS2010**

- Go Help > Samples > CSharpSamples > LinqSamples > SampleQueries
- Build and run project SampleQueries
- Shows the SQL generated by Linq to SQL
- See *101+ Linq to Sql Query Samples*

- If not installed, then first
  - go Help > Samples > local Samples folder > unzip CSharpSamples.zip
  - install in Program Files\Microsoft Visual Studio 10.0\Samples\1033\CSharpSamples\n
**Weird uses of Linq**

- Ray tracer in one Linq statement by Luke Hoban (now F# program manager)
LinQ-related patent applications

- Data retrieval from a database utilizing efficient eager loading and customized queries. US2008177716
- Syntax for members added through object protocol systems and methods. US2008046456
- A generic interface for deep embedding of expression trees in programming languages. US2007271233
- Generational global name table. US2007061092 (?)
- Query comprehensions. US2007050347
- Lambda expressions. US2007044083
- Integrating query-related operators in a programming language. US2007027849
- Compiler supporting programs as data objects. US2007028223
- Anonymous types for statically typed queries. US2007027862
- Extending expression-based syntax for creating object instances. US2007028212
- Retrieving and persisting objects from/to relational databases. US2007027906
- Intelligent SQL generation for persistent object retrieval. US2007027905
- Free/outer variable capture. US2007028222
- Architecture that extends types using extension methods. US2007028209

New in C# 4.0

- Default-value method parameters
- Named method arguments
- Type `dynamic`: Run-time resolution of method calls, method overloading, operators
- Co-variance and contra-variance in generic interfaces: `I<out T>` and `I<in T>`
- Task Parallel Library: simple concurrency

-----

Default-value (optional) parameters

- Parameters may have default values

```csharp
static uint Convert(String s, uint radix = 10) {
    ...
}
```
- Default value must be compile-time constant
- Default-value arguments may be left out

```csharp
Convert(“123”);
Convert(“123”, 10);
Convert(“123”, 8);
```

- Rules
  - Order: Required; default; parameter array
  - A `ref` or `out` parameter cannot have default value
Named arguments

- In calls, arguments may be named
  ```csharp
  s.IndexOf("foo", startIndex: 4, count: 3)
  s.Replace(oldValue: "foo", newValue: "bar")
  ```
  - Useful when several arguments have same type
- Named arguments can appear in any order
  ```csharp
  s.IndexOf("foo", count: 3, startIndex: 4)
  s.Replace(newValue: "bar", oldValue: "foo")
  ```
- Works with ref and out too
  ```csharp
  quotrem(7, 3, quot: out q, rem: out r);
  ```
- Named arguments must come last in call
  - Cannot really be used with parameter arrays

Subtlety:
Subtype may rename parameters

```csharp
interface II {
    void M(int xx, int yy);
}
class AA : II {
    public virtual void M(int x, int y) { ... }
}
class BB : AA {
    public override void M(int y, int x) { ... }
}

II obj;
obj = new AA();
obj.M(xx: 2, yy: 3);
obj = new BB();
obj.M(xx: 2, yy: 3);
```

What happens here?
And here?
Name is resolved by obj’s compile-time type: II
C# 4.0: Dynamically typed expressions

- Type `dynamic` is a compile-time type
- Contrast with `var`-declared variables

```
var x = 29;
x = "Kirkevej";
... x.ToUpper()
```

- Error at compile-time

```
dynamic x = 29;
x = "Kirkevej";
... x.ToUpper()
```

- OK at compile- and run-time

- Type `dynamic` is similar to `Object`: a value gets boxed and carries its exact type at run-time
- **Moreover**, resolution of methods, operators and overloading happens at run-time

Using expressions of type `dynamic`

```csharp
static void TryDynamic(dynamic x) {
    if (x is int)
        Console.WriteLine(x - 10);
    long xLong = x;  // Check and conversion
    x = M();
    Console.WriteLine(x + 10);
    dynamic[] arr = new dynamic[10];
dynamic[] arr = new dynamic[10];
    arr[0] = 19;
    arr[1] = "Electric Avenue";
    arr[2] = x;
    int number = arr[0] * 5;
    String street = arr[1].ToUpper();
}

static dynamic M() { return 16; }
```
Using the `dynamic` type

- **Can**
  - Use as method return type
  - Use as method parameter type, almost same as `Object`
  - Use as method argument, with overload res. at runtime
  - Use as type parameter in arrays and generic types:
    ```csharp
dynamic[] arr = new dynamic[10];
    List<dynamic> list = new List<dynamic>();
```
  - Cast to `dynamic`:
    ```csharp
    M(obj) versus M((dynamic)obj)
    ((dynamic)("foo".Clone())).ToUpper()
    ```
- **Cannot**
  - Use `dynamic` in generic type parameter constraint
  - Use extension methods on dynamic receiver `obj.Count()`
  - Pass anonymous method to a method with dynamic receiver
- **Type rules:**
  - An expression with a `dynamic` operand has type `dynamic`

Runtime resolution of `d.M(...)`

- `d` is a COM object
  - Resolves call at runtime through COM IDispatch
- `d` implements .NET interface `IDynamicObject`
  - Class of `d` determines how calls are resolved
  - More ...
- `d` is any other .NET object (or boxed value)
  - At runtime it will be checked whether `d` supports the operation, and if so it will be invoked
  - Method overloading is resolved at runtime
- **NO change to the .NET runtime system**
- **But new API, Dynamic Language Runtime**
Implementation of dynamic overloading

- **Example class**

```csharp
class TestDynamicImplementation {
    public static void Main() {
        dynamic x;
        x = "foo";
        int res = MyMethod(x);
        Console.WriteLine(res);
    }
    static int MyMethod(int x) { return x * x; }
    static int MyMethod(String s) { return s.Length; }
}
```

- **Compiled to ordinary .NET bytecode**
- **Using the Dynamic Language Runtime API**

```
.class TestDynamicImplementation {
    .class abstract sealed nested private '<Main>o__SiteContainer0' {
        .field static CallSite 1<Func 3<CallSite,object,int32>> '<Op_Site1'
        .field static CallSite 1<Func 4<CallSite,Type,object,object>> '<Op_Site2'
    }
    .method public hidebysig static void  Main() cil managed {
        .locals init (object V_0,
                     int32 V_1,
                     class CSharpArgumentInfo[] V_2)
        IL_0000:  ldstr      "foo"
        IL_0005:  stloc.0
        IL_0006:  ldsfld     '<Main>o__SiteContainer0'::'<>p__Site1'
        IL_000b:  brtrue.s   IL_0027
        IL_000d:  ldc.i4.0
        IL_000e:  ldtoken    [mscorlib]System.Int32
        IL_0013:  call       Type::GetTypeFromHandle(RuntimeTypeHandle)
        IL_0018:  ldc.i4.0
        IL_0019:  ldstr      "MyMethod"
**Runtime overhead of** `dynamic`

- A value of type `dynamic` must be boxed, and hence allocated in the heap, not stack
- Safety of `dynamic` relies on runtime checks
- Heap allocation and checks take time:

```java
double sum = 0.0;
for (int i=count; i>0; i--)
    sum += (1.0 + i) * i;
```

```
3.2 ns/iter
```

```java
dynamic sum = 0.0;
for (int i=count; i>0; i--)
    sum += (1.0 + i) * i;
```

```
24.5 ns/iter
```

- If a method call has a dynamic argument, overload resolution is done at runtime: slow

**Feature interaction/corner case: named arguments and dynamic**

```java
II obj = ...;
obj.M(xx: 2, yy: 3);
```

Arg. names resolved by compile-time type

```java
dynamic obj = ...;
obj.M(xx: 2, yy: 3);
```

No compile-time type, ehh??

- Runtime overload resolution seems to ...
  - first finds methods with the right number and names of parameters
  - next finds the best overload – e.g. `String` better than `Object` for argument “foo”
Reminder: Generics in C# and Java

- Polymorphic types
  ```csharp
  interface IEnumerable<T> { ... }
  class List<T> : IEnumerable<T> { ... }
  struct Pair<T,U> { T fst; U snd; ... }
  delegate R Func<A,R>(A x);
  ```

- Polymorphic methods
  ```csharp
  void Process<T>(Action<T> act, T[] xs)
  void <T> Process(Action<T> act, T[] arr)
  ```

- Type parameter constraints
  ```csharp
  void <T extends Comparable<T>> Sort(T[] arr)
  void <T> Sort(T[] arr) where T : IComparable<T>
  ```

Java

- Invariance in type parameters

  - Assume Student is subtype of Person
    ```csharp
    void PrintPeople(IEnumerable<Person> ps) { ... }
    ```
    ```java
    IEnumerable<Student> students = ...;
    PrintPeople(students);
    ```
    Java and C# 3 say
    NO: Ill-typed!

  - C# 3 and Java:
    - A generic type is invariant in its parameter
    - I<Student> is not subtype of I<Person>

  - Co-variance (co=with):
    - I<Student> is subtype of I<Person>

  - Contra-variance (contra=against):
    - I<Person> is subtype of I<Student>
Co-/contra-variance is unsafe in general

• Co-variance is unsafe in general

```csharp
List<Student> ss = new List<Student>();
List<Person> ps = ss;
ps.Add(new Person(...));
Student s0 = ss[0];
```

Wrong!
Because would allow writing Person to Student list

• Contra-variance is unsafe in general

```csharp
List<Person> ps = ...;
List<Student> ss = ps;
Student s0 = ss[0];
```

Wrong!
Because would allow reading Student from Person list

• But:
  – co-variance OK if we only read (output) from list
  – contra-variance OK if we only write (input) to list

Co-variance in interfaces, C# 4.0

• When an I<T> only produces/outputs T’s, it is safe to use an I<Student> where a I<Person> is expected
• This is co-variance
• Co-variance is declared with the `out` modifier

```csharp
interface IEnumerable<out T> {
    IEnumerator<T> GetEnumerator();
}
interface IEnumerator<out T> {
    T Current { get; }
}
```

• Type T can be used only in output position; e.g. not as method argument (input)
Contra-variance in interfaces, C# 4.0

- When an I<T> only consumes/inputs T’s, it is safe to use an I<Person> where an I<Student> is expected
- This is contra-variance
- Contra-variance is declared with in modifier

```
interface IComparer<in T> {
    int Compare(T x, T y);
}
```

- Type T can be used only in input position; e.g. not as method return type (output)

Variance in function types, C# 4.0

- A C# delegate type is
  - co-variant in return type (output)
  - contra-variant in parameters types (input)
- Return type co-variance:

```
Func<int,Student> nthStudent = ...
Func<int,Person> nthPerson = nthStudent;
```

- Argument type contra-variance:

```
Func<Person,int> personAge = ...  
Func<Student,int> studentAge = personAge;
```
How Java does it: use-side wildcards

- Use-side co-variance

```java
void PrintPeople(ArrayList<? extends Person> ps) {
    for (Person p : ps) { ... }
}
... PrintPeople(new ArrayList<Student>());
```

- Use-side contra-variance

```java
void AddStudentToList(ArrayList<? super Student> ss) {
    ss.add(new Student());
}
... AddStudentToList(new ArrayList<Person>());
```

Decidability of Java’s wildcard types

- It is unknown whether Java typechecking is decidable – so not clear that the compiler will always terminate

- Indeed, some short Java programs cause Sun’s Java compiler serious headaches:

```java
class T { }
class N<X> { }
class C<X> extends N<N<? super C<C<X>>> { 
    N<? super C<T>> cast(C<T> c) { 
        return c;
    }
}
```

Kennedy & Pierce 2007
The future of C#

• Further ahead
  – Compiler API usable by client code (PDC2010)
  – Software transactional memory?
  – Join patterns (a library by Claudio Russo 2007)?
  – Actors (à la Scala and F#, MSR library 2009)?

“C# will keep growing and some day it may collapse under its own weight”

Anders Hejlsberg at the IT University 2007 (quoted from memory)