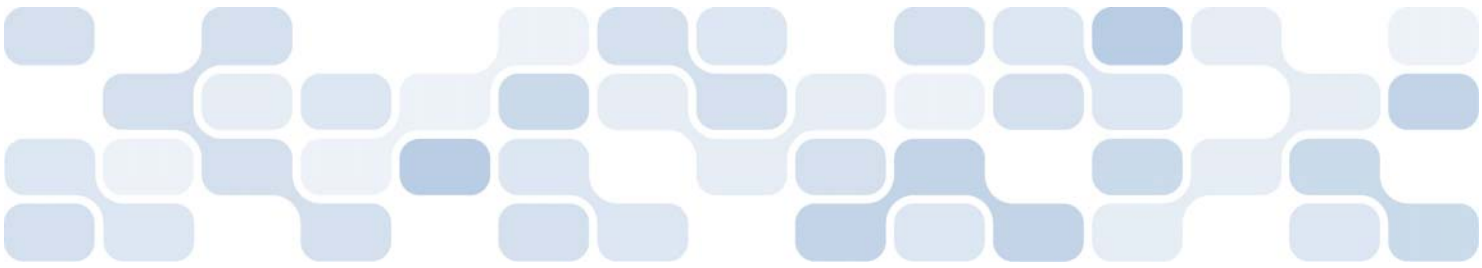
Microsoft

C#

バージョン 3.0 仕様

2005 年 9 月



注意

目次

26. C# 3.0 の概要

C# 3.0 (“C# Orcas”) は、C# 2.0 で構築された複数言語の拡張版を導入し、上位関数形式のクラス ライブラリの作成および使用をサポートしています。拡張版を使用すると、リレーショナル データベースや XML など、ドメイン内でのクエリ言語と同等の表現力を持つ、合成 API を構成できます。拡張版には次のものが含まれます。

- 暗黙に型付けされたローカル変数。これは、ローカル変数の型を変数の初期化に使用される式から推測できます。
- 拡張メソッド。これは、追加メソッドを使用して既存の型および構成された型を拡張できます。
- ラムダ式。デリゲート型および式階層両方に改良型の推測や変換を提供する匿名メソッドの展開。
- オブジェクト初期化子。オブジェクトの構築および初期化が簡単になります。
- 匿名型。これは、オブジェクト初期化子から自動的に推測、作成されるタプル型です。
- 暗黙に型付けされた配列。配列の要素型を配列初期化子から推測する配列作成および初期化の形式。
- クエリ式。これは、SQL や XQuery などのリレーショナル クエリ言語および階層クエリ言語に類似したクエリ用言語の統合された構文を提供します。
- 式階層。これは、ラムダ式をコード (デリゲート) の代わりにデータ (式階層) として表現できます。

この文書は、上記機能の技術的概要です。この文書は、C# 言語仕様 1.2 (§1 ～ §18) および C# 言語仕様 2.0 (§19 ～ §25) に言及しています。両方とも C# 言語のホームページ (http://msdn.microsoft.com/vcsharp/language) から入手可能です。

26.1 暗黙に型付けされたローカル変数

暗黙に型付けされたローカル変数の宣言で、宣言されるローカル変数の型は、変数の初期化に使用される式から推測されます。ローカル変数宣言が型として `var` を指定し、`var` という名前の型が有効範囲内にない場合、この宣言は暗黙に型付けされたローカル変数宣言になります。たとえば、次の通りです。

```
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

上記の暗黙に型付けされたローカル変数宣言は、次の明示的に型付けされた宣言とまったく同じです。

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

暗黙に型付けされたローカル変数宣言のローカル宣言子は、次の制限を受けます。

- 宣言子には初期化子が含まれていなければなりません。
- 初期化子は式でなければなりません。初期化子はそれ自体 オブジェクト初期化子または コレクション初期化子 (§26.4) にすることはできませんが、オブジェクト初期化子または コレクション初期化子を含む新しい式にすることができます。
- 初期化子式のコンパイル時型は null 型にすることはできません。
- ローカル変数宣言に複数の宣言子が含まれる場合、初期化子はすべて同じコンパイル時型でなければなりません。

暗黙に型付けされたローカル変数宣言の不正な例を次に示します。

```
var x;                // エラー、型を推測する初期化子がない
var y = {1, 2, 3};    // エラー、コレクション初期化子は許可されていない
var z = null;         // エラー、null 型は許可されていない
```

下位互換性のために、ローカル変数宣言が型として `var` を指定し、`var` という名前の型が有効範囲内にある場合、その宣言はその型を参照しますが、あいまいさに注意を促す警告が生成されます。`var` という名前の型は、型名を大文字で始めるという制定された規定に違反するので、この状況が発生するのはきわめてまれです。

`for` 文 (§8.8.3) の *for-initializer* および `using` 文 (§8.13) の *resource-acquisition* は、暗黙に型付けされたローカル変数宣言にすることができます。同様に、`foreach` 文 (§8.8.4) の反復変数を暗黙に型付けされたローカル変数として宣言してもかまいませんが、この場合、反復変数の型は列挙される コレクションの要素型になると推測されます。

```
int[] numbers = { 1, 3, 5, 7, 9 };
foreach (var n in numbers) Console.WriteLine(n);
```

上記例では、`n` の型は `int` で、`numbers` の要素型と推測されます。

26.2 拡張メソッド

拡張メソッドは、instance メソッド構文を使用して呼び出すことができる静的メソッドです。実際に拡張メソッドは、追加のメソッドを使用して既存の型および構築された型を拡張できます。

注意

拡張メソッドは検索されることが少なく、通常のインスタンスメソッドと比べて機能的により多くの制限を受けています。このため、拡張メソッドの使用はできるだけ控え、インスタンスメソッドが適していない場合や可能でない場合のみ使用するようにしてください。

プロパティ、イベント、および演算子など、他の拡張メンバについては検討中ですが、現在はサポートされていません。

26.2.1 拡張メソッドの宣言

拡張メソッドは、メソッドの最初のパラメータに修飾子としてキーワード `this` を指定することにより宣言されます。拡張メソッドは静的クラスでのみ宣言することができます。2 つの 拡張メソッドを宣言する静的クラスの例を次に示します。

```
namespace Acme.Utilities
{
   public static class Extensions
   {
      public static int ToInt32(this string s) {
         return Int32.Parse(s);
      }

      public static T[] Slice<T>(this T[] source, int index, int count) {
         if (index < 0 || count < 0 || source.Length – index < count)
            throw new ArgumentException();
         T[] result = new T[count];
         Array.Copy(source, index, result, 0, count);
         return result;
      }
   }
}
```

拡張メソッドは、正規の静的メソッドの能力をすべて保持しています。拡張メソッドは、いったんインポートされると、インスタンスメソッド構文を使用して呼び出すことができます。

26.2.2 拡張メソッドのインポート

拡張メソッドは、*using-namespace-directives* (§9.3.2) を通してインポートされます。名前空間に含まれる型をインポートする他に、*using-namespace-directive* は、すべての 拡張メソッドを名前空間のすべての静的クラスにインポートします。実際には、インポートされた 拡張メソッドは、最初のパラメータが指定し、正規のインスタンスメソッドより優先順位の低い型で、追加メソッドとして表示されます。たとえば、上記例の `Acme.Utilities` 名前空間が *using-namespace-directive* を使用してインポートされる場合、

```
using Acme.Utilities;
```

インスタンスメソッド構文を使用して静的クラス `Extensions` で 拡張メソッドを呼び出すことができるようになります。

```
string s = "1234";
int i = s.ToInt32();            // Extensions.ToInt32(s) と同じ

int[] digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int[] a = digits.Slice(4, 3);   // Extensions.Slice(digits, 4, 3) と同じ
```

26.2.3 拡張メソッド呼び出し

拡張メソッド呼び出しの詳細規則を次に示します。あるフォームの 1 つのメソッド呼び出し (§7.5.5.1) では、

expr `.` *identifier* `(` `)`

expr `.` *identifier* `(` *args* `)`

expr `.` *identifier* `<` *typeargs* `>` `(` `)`

expr `.` *identifier* `<` *typeargs* `>` `(` *args* `)`

通常の呼び出し処理により適用可能な インスタンスメソッドが見つからない場合 (特に呼び出しの候補メソッドのセットが空の場合)、拡張メソッド呼び出しとして構成子の処理が試みられます。メソッド呼び出しは、それぞれ次のいずれかに、最初に書き換えらます。

identifier (*expr*)

identifier (*expr* , *args*)

identifier < *typeargs* > (*expr*)

identifier < *typeargs* > (*expr* , *args*)

次に、書き換えられた形式が静的メソッド呼び出しとして処理されますが、*identifier* が解決される場合は除きます。最も近くを囲んでいる名前空間宣言から始め、続けてぞれぞれを囲んでいる名前空間宣言、最後に含んでいるコンパイル ユニットの処理を行います。継続的な試みで名前空間宣言の using-namespace-directives によりインポートされた識別子によって名付けられたすべてのアクセス可能な 拡張メソッドを含むメソッドのグループで、メソッド呼び出しを書き換える処理が行われます。空でない候補メソッドのセットを生じる最初のメソッド グループは、書き換えられたメソッド呼び出しに選択されたメソッド グループです。すべての試みを行って、空の候補メソッドのセットが生じる場合には、コンパイル時エラーが発生します。

前規則は、インスタンスメソッドが 拡張メソッドに優先し、内側の名前空間宣言にインポートされた 拡張メソッドが、外側の名前空間宣言にインポートされた 拡張メソッドに優先することを意味します。

```
using N1;

namespace N1
{
   public static class E
   {
      public static void F(this object obj, int i) { }

      public static void F(this object obj, string s) { }
   }
}

class A { }

class B
{
   public void F(int i) { }
}

class C
{
   public void F(object obj) { }
}

class X
{
   static void Test(A a, B b, C c) {
      a.F(1);           // E.F(object, int)
      a.F("hello");     // E.F(object, string)

      b.F(1);           // B.F(int)
      b.F("hello");     // E.F(object, string)

      c.F(1);           // C.F(object)
      c.F("hello");     // C.F(object)
   }
}
```

上記例では、B のメソッドが最初の 拡張メソッドに優先し、C のメソッドが両方の 拡張メソッドに優先します。

26.3 ラムダ式

C# 2.0 では、デリゲート値が見込まれる「インライン」にコード ブロックを記述できる、匿名メソッドを導入しています。匿名メソッドは、関数型プログラミング言語の表現力を十分に発揮しますが、匿名メソッド構文は、現実にかなり冗長ですが絶対に必要です。***ラムダ式***は、匿名メソッドの記述に簡潔で機能的な構文を提供しています。

ラムダ式は、パラメータ リストとして記述され、後に => トークン、その後に式またはステートメント ブロックが続きます。

expression:
    assignment
    non-assignment-expression

non-assignment-expression:
    conditional-expression
    lambda-expression
    query-expression

lambda-expression:
    (*lambda-parameter-list*$_{opt}$) => *lambda-expression-body*
    implicitly-typed-lambda-parameter => *lambda-expression-body*

lambda-parameter-list:
    explicitly-typed-lambda-parameter-list
    implicitly-typed-lambda-parameter-list

explicitly-typed-lambda-parameter-list
    explicitly-typed-lambda-parameter
    explicitly-typed-lambda-parameter-list , *explicitly-typed-lambda-parameter*

explicitly-typed-lambda-parameter:
    parameter-modifier$_{opt}$ *type identifier*

implicitly-typed-lambda-parameter-list
    implicitly-typed-lambda-parameter
    implicitly-typed-lambda-parameter-list , *implicitly-typed-lambda-parameter*

implicitly-typed-lambda-parameter:
    identifier

lambda-expression-body:
    expression
    block

ラムダ式のパラメータは、明示的または暗黙に型付けできます。明示的に型付けされたパラメータ リストでは、各パラメータの型が明示的に指定されます。暗黙に型付けされたパラメータ リストでは、パラメータの型は、ラムダ式が発生するコンテキストから推測されます。特に、ラムダ式が互換性のあるデリゲート型に変換される場合、そのデリゲート型はパラメータ型 (§26.3.1) を提供します。

1 つの暗黙に型付けされたパラメータを持つラムダ式では、パラメータ リストからかっこを省略できます。言い換えれば、以下の形式

(*param*) => *expr*

のラムダ式は、次のように省略できます。

param `=>` *expr*

ラムダ式の例は以下の通りです。

```
x => x + 1                      // 暗黙に型付けされた、式本体
x => { return x + 1; }          // 暗黙に型付けされた、文本体
(int x) => x + 1                // 明示的に型付けされた、式本体
(int x) => { return x + 1; }    // 明示的に型付けされた、文本体
(x, y) => x * y                 // 複数のパラメータ
() => Console.WriteLine()       // パラメータなし
```

一般に、C# 2.0 仕様の §21 に記載された匿名メソッドの仕様は、ラムダ式にも適用されます。ラムダ式は、匿名メソッドの関数型上位セットであり、次の追加機能があります。

- 匿名メソッドは、パラメータ型を明示的に指定する必要があるのに対し、ラムダ式はパラメータ型の省略や推測が可能です。
- 匿名メソッドの本体はステートメント ブロックにしかできませんが、ラムダ式の本体は、式またはステートメント ブロックにすることができます。
- 引数として渡されるラムダ式は、引数の型推論 (§26.3.2) およびメソッドのオーバーロード解決 (§26.3.3) に関係します。
- 式本体を持つラムダ式は、式階層 (§26.8) に変換できます。

注意

PDC 2005 Technology Preview コンパイラは、ステートメント ブロック本体でラムダ式をサポートしていません。ステートメント ブロック本体が必要な場合は、C# 2.0 匿名メソッドの構文を使用しなければなりません。

26.3.1 ラムダ式の変換

anonymous-method-expression と同様に、*lambda-expression* は、特別な変換規則を持つ値として分類されます。この値には型がありませんが、互換性のあるデリゲート型に暗黙に変換できます。特にデリゲート型 `D` は lambda-expression `L` と互換性があります。ただし、

- `D` と `L` のパラメータ数は同じです。
- `L` が明示的に型付けされたパラメータ リストを指定する場合、`D` の各パラメータの型および修飾子は、`L` の該当するパラメータと同じです。
- `L` が暗黙に型付けされたパラメータ リストを指定する場合、`D` には `ref` パラメータまたは `out` パラメータがありません。
- `D` が `void` 戻り型を指定し、`L` の本体が式の場合、`L` の各パラメータに `D` の該当するパラメータの型が指定してあれば、`L` の本体は、*statement-expression* (§8.6) として許可されている有効な式です。

- D が void 戻り型を指定し、L の本体がステートメント ブロックの場合、L の各パラメータに D の該当するパラメータの型が指定してあれば、L の本体は、return 文が式を指定していない有効なステートメント ブロックです。
- D が non-void 戻り型を指定し、L の本体が式の場合、L の各パラメータに D の該当するパラメータの型が指定してあれば、L の本体は、D の戻り型に暗黙に変換可能である有効な式です。
- D が non-void 戻り型を所有し、L の本体がステートメント ブロックの場合、L の各パラメータに D の該当するパラメータの型が指定してあれば、L の本体は、到達不能なエンド ポイントを持つ有効なステートメント ブロックになり、その中で各 return 文が、D の戻り型に暗黙に変換可能である式を指定します。

次の例は、型 A の引数を取り、型 R の値を戻す関数を表す、ジェネリック デリゲート型 Func<A,R> を使用しています。

```
delegate R Func<A,R>(A arg);
```

割り当てでは

```
Func<int,int> f1 = x => x + 1;          // Ok
Func<int,double> f2 = x => x + 1;       // Ok
Func<double,int> f3 = x => x + 1;       // エラー
```

各ラムダ式のパラメータおよび戻り型は、ラムダ式が割り当てられる変数の型から決定されます。最初の割り当ては、ラムダ式をデリゲート型 Func<int,int> に正常に変換します。x に型 int が指定されている場合、x + 1 は型 int へ暗黙に変換可能な有効な式であるためです。同様に、第 2 番目の割り当ては、ラムダ式をデリゲート型 Func<int,double> に正常に変換します。x + 1 (型 int) の結果が型 double へ暗黙に変換可能であるためです。しかし、第 3 の割り当ては、コンパイル時エラーになります。x に型 double が指定されている場合、x + 1 (型 double) の結果が型 int へ暗黙に変換できないためです。

26.3.2 型推論

ジェネリック メソッドを型引数を指定せずに呼び出す場合、型推論プロセスにより、呼び出しに対する型割り当ての推論が試みられます。引数としてジェネリック メソッドに渡されたラムダ式は、この型推論プロセスに関係します。

§20.6.4 に記述したように、まず、引数ごとに別個に型推論が発生します。この最初の段階では、ラムダ式である引数から何も推測されません。しかし、最初の段階の後に、反復プロセスを使用してラムダ式から追加の推論が行われます。特に、次に記載したすべてが真となるために、引数が 1 つまたは複数存在する限り、推論が行われます。

- 次の呼び出された L では、まだ推論が行われていない引数はラムダ式です。
- 次の呼び出された P では、該当するパラメータの型は、1 つまたは複数の型パラメータを含む戻り型を持つデリゲート型です。
- L に暗黙に型付けしたパラメータ リストがある場合、P および L のパラメータ数は同じであり、P の各パラメータは、L の該当するパラメータと同じ修飾子になるか、または修飾子がありません。

- P のパラメータ型には、メソッドの型パラメータが含まれていないか、または一貫した推論を既に行ったメソッドの型パラメータだけが含まれます。
- L に明示的に型付けしたパラメータ リストがある場合、推測された型を P のメソッドの型パラメータに代入すると、P の各パラメータは、L の該当するパラメータと同じ型になります。
- L に暗黙に型付けしたパラメータ リストがある場合、推測された型を P のメソッドの型パラメータに代入し、その結果生じるパラメータ型を L のパラメータに指定すると、L の本体は有効な式またはステートメント ブロックになります。
- 下記に示すように、L の戻り型を推測できます。

このような引数ごとに、P の戻り型と L の推測された戻り型を関連付けることにより、その引数から推論が行われ、新しい推論が蓄積された推論セットに追加されます。推論が行われなくなるまで、このプロセスが繰り返されます。

型推論とオーバーロード解決のために、ラムダ式 L の***推測された戻り型***は次のように決定されます。

- L の本体が式の場合、その式の型は L の推測された戻り型になります。
- L の本体がステートメント ブロックの場合、ブロックの `return` 文内の式の型により形成されるセットに 1 つの型が含まれ、セットの各型を暗黙に変換できる場合、およびその型が null 型でない場合は、型は L の推測した戻り型になります。
- それ以外の場合は、L の戻り型を推測できません。

ラムダ式に関係する型推論の例として、`System.Query.Sequence` クラスで宣言された `Select` 拡張メソッドを考えてみましょう。

```
namespace System.Query
{
   public static class Sequence
   {
      public static IEnumerable<S> Select<T,S>(
         this IEnumerable<T> source,
         Func<T,S> selector)
      {
         foreach (T element in source) yield return selector(element);
      }
   }
}
```

`System.Query` 名前空間を `using` 句を使用してインポートしたと想定し、クラス `Customer` に型 `string` の `Name` プロパティを指定した場合、顧客リスト名を選択するために `Select` メソッドを使用できます。

```
List<Customer> customers = GetCustomerList();
IEnumerable<string> names = customers.Select(c => c.Name);
```

`Select` の 拡張メソッド呼び出し (§26.2.3) は、呼び出しを静的メソッド呼び出しに書き換えることにより処理されます。

```
IEnumerable<string> names = Sequence.Select(customers, c => c.Name);
```

型引数が明示的に指定されていないので、型引数を推測するのに型推論を使用します。まず、`customers` 引数を `source` パラメータに関連付け、T が `Customer` になると推測します。次に、

上記の ラムダ式の型推論プロセスを使用し、`c` に型 `Customer` を指定し、式 `c.Name` を `selector` パラメータの戻り型に関連付け、`S` が `string` になると推測します。したがって、この呼び出しは以下と同じです。

```
Sequence.Select<Customer,string>(customers, (Customer c) => c.Name)
```

また、結果は型 `IEnumerable<string>` になります。

次の例は、ラムダ式の型推論により、ジェネリック メソッド呼び出しの引数間で型情報を「流す」方法を示しています。メソッドを指定した場合、

```
static Z F<X,Y,Z>(X value, Func<X,Y> f1, Func<Y,Z> f2) {
   return f2(f1(value));
}
```

呼び出しの型推論

```
double seconds = F("1:15:30", s => TimeSpan.Parse(s), t => t.TotalSeconds);
```

は以下のように行います。まず引数 `"1:15:30"` を `value` パラメータと関連付け、`X` が `string` になると推測します。次に、最初のラムダ式のパラメータ `s` に推測された型 `string` を指定し、式 `TimeSpan.Parse(s)` を `f1` の戻り型に関連付け、`Y` が `System.TimeSpan` になると推測します。最後に、第 2 のラムダ式のパラメータ `t` に推測された型 `System.TimeSpan` を指定し、式 `t.TotalSeconds` を `f2` の戻り型に関連付け、`Z` が `double` になると推測します。したがって、呼び出しの結果は型 `double` になります。

26.3.3 オーバーロード解決

一定の状況下で、引数リストのラムダ式は、オーバーロード解決に影響します。

次の規則は §7.4.2.3 を増強します。推測された戻り型 (§26.3.2) が存在するラムダ式 `L` を指定した場合、`L` のデリゲート型 D_1 への暗黙的変換は、`L` のデリゲート型 D_2 への暗黙的変換よりも適した変換です。ただし、D_1 および D_2 のパラメータ リストが同じで、`L` の推測された戻り型から D_1 の戻り型への暗黙的変換が、`L` の推測された戻り型から D_2 の戻り型への暗黙的変換より適した変換の場合です。この条件が真でなければ、いずれの変換も適していません。

次の例は、この規則の効果を示しています。

```
class ItemList<T>: List<T>
{
   public int Sum<T>(Func<T,int> selector) {
      int sum = 0;
      foreach (T item in this) sum += selector(item);
      return sum;
   }

   public double Sum<T>(Func<T,double> selector) {
      double sum = 0;
      foreach (T item in this) sum += selector(item);
      return sum;
   }
}
```

`ItemList<T>` クラスには 2 つの `Sum` メソッドがあります。それぞれ `selector` 引数を取りますが、これは、合計する値をリスト項目から抽出します。抽出された値は `int` または `double` いずれかになり、結果として生じる和は、`int` または `double` いずれかと同じです。

例の Sum メソッドは、詳細行リストから順に合計を計算するために使用できます。

```
class Detail
{
   public int UnitCount;
   public double UnitPrice;
   ...
}

void ComputeSums() {
   ItemList<Detail> orderDetails = GetOrderDetails(...);
   int totalUnits = orderDetails.Sum(d => d.UnitCount);
   double orderTotal = orderDetails.Sum(d => d.UnitPrice * d.UnitCount);
   ...
}
```

orderDetails.Sum の最初の呼び出しでは、ラムダ式 d => d.UnitCount は、Func<Detail,int> と Func<Detail,double> 両方に互換性があるので、両方の Sum メソッドが適用可能です。しかし、オーバーロード解決では、最初の Sum メソッドを選択します。Func<Detail,int> への変換の方が Func<Detail,double> への変換より適しているためです。

orderDetails.Sum の 2 番目の呼び出しでは、ラムダ式 d => d.UnitPrice * d.UnitCount は、型 double の値を生成するので、2 番目の Sum メソッドだけが適用可能です。したがって、オーバーロード解決は呼び出しに 2 番目の Sum メソッドを選択します。

26.4 オブジェクト初期化子および コレクション初期化子

オブジェクト作成式 (§7.5.10.1) には、新たに作成されたオブジェクトのメンバまたは新たに作成されたコレクションの要素を初期化する オブジェクト初期化子または コレクション初期化子を含めることができます。

object-creation-expression:
    new *type* (*argument-list*$_{opt}$) *object-or-collection-initializer*$_{opt}$
    new *type* *object-or-collection-initializer*

object-or-collection-initializer:
    object-initializer
    collection-initializer

オブジェクト作成式は、コンストラクタ引数リストおよび囲みかっこを省略できます。ただし、オブジェクト初期化子または コレクション初期化子が含まれている場合です。コンストラクタ引数リストおよび囲みかっこを省略することは、空の引数リストを指定することと同じです。

オブジェクト初期化子または コレクション初期化子が含まれるオブジェクト作成式を実行する場合、最初にインスタンス コンストラクタを呼び出し、次に オブジェクト初期化子または コレクション初期化子が指定したメンバ初期化あるいは要素初期化を実行して行います。

オブジェクト初期化子または コレクション初期化子では、初期化されるオブジェクト インスタンスを参照することは不可能です。

26.4.1 オブジェクト初期化子

オブジェクト初期化子は、オブジェクトの 1 つまたは複数のフィールドあるいはプロパティの値を指定します。

object-initializer:
{ *member-initializer-list*$_{opt}$ }
{ *member-initializer-list* , }

member-initializer-list:
member-initializer
member-initializer-list , *member-initializer*

member-initializer:
identifier = *initializer-value*

initializer-value:
expression
object-or-collection-initializer

オブジェクト初期化子は、一連の メンバ初期化子で構成され、{ および } トークンで囲み、コンマで分割します。各 メンバ初期化子は、初期化されるオブジェクトのアクセス可能なフィールドまたはプロパティに名前を付け、後に等号と式あるいは オブジェクト初期化子か コレクション初期化子を指定します。オブジェクト初期化子では、同じフィールドあるいは同じプロパティに複数の メンバ初期化子を含めるとエラーになります。

等号の後に式を指定する メンバ初期化子は、フィールドあるいはプロパティに対する引数 (§7.13.1) と同じ方法で処理されます。

等号の後に オブジェクト初期化子を指定する メンバ初期化子は、埋め込みオブジェクトの初期化です。オブジェクト初期化子の割り当ては、フィールドあるいはプロパティに新規値を割り当てる代わりに、フィールドあるいはプロパティのメンバへの割り当てとして処理されます。値型のプロパティは、この構成子を使用して初期化できません。

等号の後に コレクションの初期化を指定する メンバ初期化子は、埋め込みコレクションの初期化です。フィールドあるいはプロパティに新規コレクションを割り当てる代わりに、初期化子に指定された要素がフィールドあるいはプロパティが参照したコレクションに追加されます。フィールドあるいはプロパティは、**§エラー! 参照元が見つかりません。** で指定された要求を満たすコレクション型でなければなりません。

次のクラスは、 2 つの座標を持つポイントを表します。

```
public class Point
{
   int x, y;

   public int X { get { return x; } set { x = value; } }
   public int Y { get { return y; } set { y = value; } }
}
```

`Point` のインスタンスは、次のように作成され、初期化されます。

```
var a = new Point { X = 0, Y = 1 };
```

下記と同じ効果があります。

```
var a = new Point();
a.X = 0;
a.Y = 1;
```

次のクラスは、2 ポイントから作成された長方形を表します。


```
public class Rectangle
{
   Point p1, p2;

   public Point P1 { get { return p1; } set { p1 = value; } }
   public Point P2 { get { return p2; } set { p2 = value; } }
}
```

`Rectangle` のインスタンスは、次のように作成され、初期化されます。

```
var r = new Rectangle {
   P1 = new Point { X = 0, Y = 1 },
   P2 = new Point { X = 2, Y = 3 }
};
```

下記と同じ効果があります。

```
var r = new Rectangle();
var __p1 = new Point();
__p1.X = 0;
__p1.Y = 1;
r.P1 = __p1;
var __p2 = new Point();
__p2.X = 2;
__p2.Y = 3;
r.P2 = __p2;
```

ここで、`__p1` および `__p2` は一時的な変数であり、他では表示不可でアクセス不可になります。

`Rectangle` のコンストラクタが `2` つの埋め込み `Point` インスタンスを割り当てている場合、

```
public class Rectangle
{
   Point p1 = new Point();
   Point p2 = new Point();

   public Point P1 { get { return p1; } }
   public Point P2 { get { return p2; } }
}
```

次の構成子は、新しいインスタンスを割り当てる代わりに、埋め込み `Point` インスタンスを初期化することに使用できます。

```
var r = new Rectangle {
   P1 = { X = 0, Y = 1 },
   P2 = { X = 2, Y = 3 }
};
```

下記と同じ効果があります。

```
var r = new Rectangle();
r.P1.X = 0;
r.P1.Y = 1;
r.P2.X = 2;
r.P2.Y = 3;
```

26.4.2 コレクション初期化子

コレクション初期化子は、コレクションの要素を指定します。

collection-initializer:
{ *element-initializer-list*$_{opt}$ }
{ *element-initializer-list* , }

element-initializer-list:
element-initializer
element-initializer-list , *element-initializer*

element-initializer:
non-assignment-expression

コレクション初期化子は、一連のエレメント初期化子で構成され、{ および } トークンで囲み、コンマで分割します。各エレメント初期化子は、初期化されるコレクション オブジェクトに追加される要素を指定します。メンバ初期化子とのあいまいさを避けるために、エレメント初期化子を式に割り当てできません。*non-assignment-expression* プロダクションは、§26.3 に定義されています。

コレクション初期化子を含むオブジェクト作成式の例を次に示します。

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

コレクション初期化子が適用されるコレクション オブジェクトは、1 つの T に対し System.Collections.Generic.ICollection<T> を実装する型でなければなりません。さらに、暗黙的変換 (§6.1) が、各エレメント初期化子の型から T まで存在しなければなりません。この要求を満たしていない場合は、コンパイル時エラーが発生します。コレクション初期化子は、指定した要素ごとに順に ICollection<T>.Add(T) メソッドを呼び出します。

次のクラスは、氏名と電話番号リストを記載した名簿を表します。

```
public class Contact
{
   string name;
   List<string> phoneNumbers = new List<string>();

   public string Name { get { return name; } set { name = value; } }

   public List<string> PhoneNumbers { get { return phoneNumbers; } }
}
```

List<Contact> は、次のように作成され、初期化されます。

```
var contacts = new List<Contact> {
   new Contact {
      Name = "Chris Smith",
      PhoneNumbers = { "206-555-0101", "425-882-8080" }
   },
   new Contact {
      Name = "Bob Harris",
      PhoneNumbers = { "650-555-0199" }
   }
};
```

下記と同じ効果があります。


```
var contacts = new List<Contact>();
var __c1 = new Contact();
__c1.Name = "Chris Smith";
__c1.PhoneNumbers.Add("206-555-0101");
__c1.PhoneNumbers.Add("425-882-8080");
contacts.Add(__c1);
var __c2 = new Contact();
__c2.Name = "Bob Harris";
__c2.PhoneNumbers.Add("650-555-0199");
contacts.Add(__c2);
```

ここで、`__c1` and `__c2` は一時的な変数であり、他では表示不可でアクセス不可になります。

26.5 匿名型

C# 3.0 では、匿名オブジェクト初期化子を指定し匿名型のオブジェクトを作成するために、新しい演算子を使用できます。

primary-no-array-creation-expression:
    …
    anonymous-object-creation-expression

anonymous-object-creation-expression:
    `new` *anonymous-object-initializer*

anonymous-object-initializer:
    `{` *member-declarator-list*$_{opt}$ `}`
    `{` *member-declarator-list* `,` `}`

member-declarator-list:
    member-declarator
    member-declarator-list `,` *member-declarator*

member-declarator:
    simple-name
    member-access
    identifier `=` *expression*

匿名オブジェクト初期化子は匿名型を宣言し、その型のインスタンスを返します。匿名型は、`object` から直接継承している名前のないクラス型です。匿名型のメンバは、型のインスタンスの作成に使用された オブジェクト初期化子から推測される一連の読み取り/書き込みプロパティです。特に、次の形式の匿名オブジェクト初期化子

`new { `p_1 `=` e_1 `, `p_2 `=` e_2 `, …` p_n `=` e_n `}`

は、次の形式の匿名型を宣言します。

```
class __Anonymous1
{
   private T1 f1 ;
   private T2 f2 ;
   …
   private Tn fn ;
```

```
    public T1 p1 { get { return f1 ; } set { f1 = value ; } }
    public T2 p2 { get { return f2 ; } set { f2 = value ; } }
    …
    public Tn pn { get { return fn ; } set { fn = value ; } }
}
```

ここで、各 T_x は、該当する式 e_x の型です。匿名オブジェクト初期化子が null 型になる式では、コンパイル時エラーになります。

匿名型の名前は、コンパイラにより自動的に生成され、プログラム テキストでは参照できません。

同じプログラム内で、同じ名前の一連のプロパティと型を同じ順で指定する 2 つの匿名オブジェクト初期化子は、同じ匿名型のインスタンスを生成します。(プロパティは反射など特定の環境で観察可能であり、具体的なので、この定義にはプロパティの順序が含まれます。)

例では

```
var p1 = new { Name = "Lawnmower", Price = 495.00 };
var p2 = new { Name = "Shovel", Price = 26.95 };
p1 = p2;
```

`p1` および `p2` は同じ匿名型なので、最終行での割り当ては許可されています。

メンバ宣言子は、簡単名 (§7.5.2) またはメンバ アクセス (§7.5.4) に短縮することができます。これは、*プロジェクション初期化子*と呼ばれ、同じ名前を持つプロパティの割り当ておよび宣言の省略表現です。特に、形式のメンバ宣言子

identifier *expr* `.` *identifier*

は、それぞれ次に匹敵します。

identifer `=` *identifier* *identifier* `=` *expr* `.` *identifier*

したがって、プロジェクション初期化子では、*identifier* は、値およびフィールド両方、または値が割り当てられるプロパティを選択します。プロジェクション初期化子は、直観的に値だけでなく、値の名前も予測します。

26.6 暗黙に型付けされた配列

配列作成式の構文 (§7.5.10.2) は拡張され、暗黙に型付けされた配列作成式をサポートします。

array-creation-expression:
…
`new [ ]` *array-initializer*

暗黙に型付けされた配列作成式では、配列インスタンスの型は、配列初期化子で指定された要素から推測されます。特に、配列初期化子の式の型により形成されたセットには、セット内の各型を暗黙に変換できる 1 つの型が含まれなければなりません。その型が null 型でなければ、その型の配列が作成されます。1 つの型を正確に推測できない場合、または推測された型が null 型の場合、コンパイル時エラーが発生します。

暗黙に型付けされた 配列を作成する式の例を次に示します。

```
var a = new[] { 1, 10, 100, 1000 };          // int[]
```



```
var b = new[] { 1, 1.5, 2, 2.5 };           // double[]
var c = new[] { "hello", null, "world" };   // string[]
var d = new[] { 1, "one", 2, "two" };       // エラー
```

最後の式は、`int` も `string` も他の型に暗黙に変換できないため、コンパイル時エラーになります。たとえば、型が `object[]` になるよう指定する場合、明示的に型付けされた配列作成式を使用しなければなりません。あるいは、要素の 1 つを共通の基底型にキャストでき、この型が推測された要素型になります。

暗黙に型付けされた配列作成式を匿名オブジェクト初期化子と組み合わせ、匿名で型付けされたデータ構造を作成できます。たとえば次の通りです。

```
var contacts = new[] {
   new {
      Name = "Chris Smith",
      PhoneNumbers = new[] { "206-555-0101", "425-882-8080" }
   },
   new {
      Name = "Bob Harris",
      PhoneNumbers = new[] { "650-555-0199" }
   }
};
```

26.7 クエリ式

*クエリ式*は、SQL や XQuery などのリレーショナル クエリ言語および階層クエリ言語に似たクエリ用言語の統合された構文を提供します。

query-expression:
    from-clause *query-body*

from-clause:
    `from` *from-generators*

from-generators:
    from-generator
    from-generators `,` *from-generator*

from-generator:
    identifier `in` *expression*

query-body:
    from-or-where-clauses$_{opt}$ *orderby-clause*$_{opt}$ *select-or-group-clause* *into-clause*$_{opt}$

from-or-where-clauses:
    from-or-where-clause
    from-or-where-clauses *from-or-where-clause*

from-or-where-clause:
    from-clause
    where-clause

where-clause:
    `where` *boolean-expression*

orderby-clause:
`orderby` *ordering-clauses*

ordering-clauses:
ordering-clause
ordering-clauses `,` *ordering-clause*

ordering-clause:
expression *ordering-direction*$_{opt}$

ordering-direction:
`ascending`
`descending`

select-or-group-clause:
select-clause
group-clause

select-clause:
`select` *expression*

group-clause:
`group` *expression* `by` *expression*

into-clause:
`into` *identifier* *query-body*

query-expression は、*non-assignment-expression* として分類されます。その定義は §26.3 に記載してあります。

クエリ式は、`from` 句で始まり、`select` 句あるいは `group` 句いずれかで終わります。最初の `from` 句の後にゼロまたは `from` 句や `where` 句を続けることができます。各 `from` 句は、シーケンス全域にわたって反復変数を導入するジェネレータであり、各 `where` 句は、アイテムを結果から除外するフィルタです。最後の `select` 句または `group` 句は、反復変数に関する結果の形態を指定します。`select` 句あるいは `group` 句の前に、結果の順序を指定する `orderby` 句が置かれることがあります。最後に、`into` 句を使用して、あるクエリ結果を次のクエリのジェネレータとして処理することにより、クエリを「接合」できます。

クエリ式では、複数のジェネレータを持つ `from` 句は、単一のジェネレータを持つ複数の連続した `from` 句とまったく同じです。

26.7.1 クエリ式の変換

C# 3.0 言語は、クエリ式の適切な実行意味論を指定しません。むしろ、C# 3.0 はクエリ式を*クエリ式パターン*に従うメソッドの呼び出しに変換します。特にクエリ式は、**§エラー! 参照元が見つかりません。** に記載したように、特定のシグネチャ型や結果型を持つと思われる `Where`、`Select`、`SelectMany`、`OrderBy`、`OrderByDescending`、`ThenBy`、`ThenByDescending`、および `GroupBy` という名前のメソッドの呼び出しに変換されます。これらのメソッドは、問い合わされるオブジェクトのインスタンスメソッドまたはオブジェクト外部である 拡張メソッドにすることができ、クエリの実際の実行を具体化します。

クエリ式からメソッド呼び出しへの変換は、型バインディングまたはオーバーロード解決が実行される前に発生する構文マッピングです。変換では構文的に正しいことが保証されますが、意味的に正し

い C# コードを生成することは保証されていません。クエリ式の変換の後、結果として生じるメソッドの呼び出しは正規のメソッド呼び出しとして処理されます。これによりエラーが発覚することがあります。たとえば、メソッドが存在しない場合、引数の型が不正な場合、またはメソッドがジェネリックであり型推論が失敗する場合です。

クエリ式の変換は、次の一連の例により実証されます。変換規則の公式説明を後の節に記載します。

26.7.1.1 where 句

クエリ式の `where` 句

```
from c in customers
where c.City == "London"
select c
```

は、反復変数識別子と `where` 句の式を組み合わることにより作成された、合成ラムダ式を持つ `Where` メソッドの呼び出しに変換します。

```
customers.
Where(c => c.City == "London")
```

26.7.1.2 select 句

前節の例では、最も内側の反復変数を選択する `select` 句が、メソッド呼び出しへの変換によりどのように消去されるかを明確に示しました。

最も内側の反復変数以外の何かを選択する `select` 句

```
from c in customers
where c.City == "London"
select c.Name
```

は、合成ラムダ式を持つ `Select` メソッドの呼び出しに変換します。

```
customers.
Where(c => c.City == "London").
Select(c => c.Name)
```

26.7.1.3 group 句

`group` 句

```
from c in customers
group c.Name by c.Country
```

は GroupBy メソッドの呼び出しに変換します。

```
customers.
GroupBy(c => c.Country, c => c.Name)
```

26.7.1.4 orderby 句

`orderby` 句

```
from c in customers
orderby c.Name
select new { c.Name, c.Phone }
```

は、`OrderBy` メソッドの呼び出し、または降順方向が指定された場合は `OrderByDescending` メソッドの呼び出しに変換します。

```
customers.
OrderBy(c => c.Name).
Select(c => new { c.Name, c.Phone })
```

`orderby` 句の二次配列

```
from c in customers
orderby c.Country, c.Balance descending
select new { c.Name, c.Country, c.Balance }
```

は、`ThenBy` メソッドおよび `ThenByDescending` メソッドの呼び出しに変換します。

```
customers.
OrderBy(c => c.Country).
ThenByDescending(c => c.Balance).
Select(c => new { c.Name, c.Country, c.Balance })
```

26.7.1.5 複数ジェネレータ

複数ジェネレータ

```
from c in customers
where c.City == "London"
from o in c.Orders
where o.OrderDate.Year == 2005
select new { c.Name, o.OrderID, o.Total }
```

最も内側のジェネレータを除きすべてに対し `SelectMany` の呼び出しに変換します。

```
customers.
Where(c => c.City == "London").
SelectMany(c =>
   c.Orders.
   Where(o => o.OrderDate.Year == 2005).
   Select(o => new { c.Name, o.OrderID, o.Total })
)
```

複数ジェネレータが `orderby` 句と組み合わされる場合

```
from c in customers, o in c.Orders
where o.OrderDate.Year == 2005
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

追加の Select が投入され、一連のタプルの配列式および最終結果を収集します。OrderBy をシーケンス全体で動作できるようにこれが必要です。OrderBy の後に最終結果がタプルから抽出されます。

```
customers.
SelectMany(c =>
   c.Orders.
   Where(o => o.OrderDate.Year == 2005).
   Select(o => new { k1 = o.Total, v = new { c.Name, o.OrderID, o.Total } })
).
OrderByDescending(x => x.k1).
Select(x => x.v)
```

26.7.1.6 into 句

`into` 句


```
from c in customers
group c by c.Country into g
select new { Country = g.Key, CustCount = g.Group.Count() }
```

は、単に入れ子式クエリに便利な表記です。

```
from g in
   from c in customers
   group c by c.Country
select new { Country = g.Key, CustCount = g.Group.Count() }
```

次のように変換されます。

```
customers.
GroupBy(c => c.Country).
Select(g => new { Country = g.Key, CustCount = g.Group.Count() })
```

26.7.2 クエリ式パターン

*クエリ式パターン*は、クエリ式をサポートする型を実装できるメソッドのパターンを確立します。クエリ式は構文マッピングによりメソッド呼び出しに変換されるので、クエリ式の実装方法には柔軟性があります。たとえば、インスタンスメソッドも 拡張メソッドも呼び出し構文が同じなので、パターンのメソッドは、インスタンスメソッドとしても、または 拡張メソッドとしても実装できます。また、ラムダ式はデリゲート階層または式階層と互換性があるので、メソッドはデリゲート階層も、または式階層も要求できます。

クエリ式パターンをサポートしているジェネリック型 `C<T>` の推奨形式を下記に示しています。ジェネリック型は、パラメータと結果型間の関係が正しいことを明らかにするために使用されますが、非ジェネリック型のパターンも同様に実装可能です。

```
delegate R Func<A,R>(A arg);

class C<T>
{
   public C<T> Where(Func<T,bool> predicate);

   public C<S> Select<S>(Func<T,S> selector);

   public C<S> SelectMany<S>(Func<T,C<S>> selector);

   public O<T> OrderBy<K>(Func<T,K> keyExpr);

   public O<T> OrderByDescending<K>(Func<T,K> keyExpr);

   public C<G<K,T>> GroupBy<K>(Func<T,K> keyExpr);

   public C<G<K,E>> GroupBy<K,E>(Func<T,K> keyExpr, Func<T,E> elemExpr);
}

class O<T> : C<T>
{
   public O<T> ThenBy<K>(Func<T,K> keySelector);

   public O<T> ThenByDescending<K>(Func<T,K> keySelector);
}

class G<K,T>
{
   public K Key { get; }

   public C<T> Group { get; }
}
```

上記メソッドは、ジェネリック デリゲート型 `Func<A, R>` を使用していますが、パラメータと結果型の関係が同じ他のデリゲート型あるいは式階層型も同様に使用できます。

`ThenBy` メソッドおよび `ThenByDescending` メソッドは、`OrderBy` あるいは `OrderByDescending` の結果にしか使用できないという `C<T>` および `O<T>` の推奨関係に注意してください。また、それぞれが `Key` プロパティおよび `Group` プロパティを持つグループ分けのシーケンスである、`GroupBy` の結果の推奨形態にも注意してください。

標準クエリ演算子 (別の仕様で説明) を使用すると、`System.Collections.Generic.IEnumerable<T>` インターフェイスを実装する、すべての型にクエリ演算子パターンを実装することもできます。

26.7.3 公式変換規則

クエリ式は、次の変換を順に繰り返し適用すると処理されます。指定したパターンの発生がなくなるまで、各変換が適用されます。

`OrderBy` および `ThenBy` の呼び出しを生成する変換では、該当する ordering 句が `descending` 方向インジケータを指定すると、代わりに `OrderByDescending` または `ThenByDescending` の呼び出しが生成されることに注意してください。

- `into` 句が含まれるクエリ

  q_1 `into` x q_2

  は次のように変換されます。

  `from` x `in (` q_1 `)` q_2

- 複数のジェネレータを指定した `from` 句

  `from` g_1 `,` g_2 `,` … g_n

  は次のように変換されます。

  `from` g_1 `from` g_2 … `from` g_n

- `from` 句直後の `where` 句

  `from` x `in` e `where` f

  は次のように変換されます。

  `from` x `in (` e `) . Where (` x `=>` f `)`

- 複数の `from` 句、`orderby` 句、および `select` 句を指定したクエリ式

  `from` x_1 `in` e_1 `from` x_2 `in` e_2 … `orderby` k_1 `,` k_2 … `select` v

  は次のように変換されます。

  `(` `from` x_1 `in` e_1 `from` x_2 `in` e_2 …

  `select new {` `k1 =` k_1 `,` `k2 =` k_2 … `,` `v =` v `} )`

  `. OrderBy ( x => x . k1 ) . ThenBy ( x => x . k2 )` …


```
. Select ( x => x . v )
```

- 複数の from 句、orderby 句、および group 句を指定したクエリ式

  from x_1 in e_1 from x_2 in e_2 ... orderby k_1 , k_2 ... group v by g

  は次のように変換されます。

```
( from x1 in e1 from x2 in e2 ...
select new { k1 = k1 , k2 = k2 ... , v = v , g = g } )
. OrderBy ( x => x . k1 ) . ThenBy ( x => x . k2 ) ...
. GroupBy ( x => x . g , x => x . v )
```

- 複数の from 句および select 句を指定したクエリ式

  from x in e from x_1 in e_1 ... select v

  は次のように変換されます。

```
( e ) . SelectMany ( x => from x1 in e1 ... select v )
```

- 複数の from 句と group 句を指定したクエリ式

  from x in e from x_1 in e_1 ... group v by g

  は次のように変換されます。

```
( e ) . SelectMany ( x => from x1 in e1 ... group v by g )
```

- orderby 句は指定せず、1 つの from 句と select 句を指定したクエリ式

  from x in e select v

  は次のように変換されます。

```
( e ) . Select ( x => v )
```

  v が識別子 x の場合を除き、変換は単に次のようになります。

```
( e )
```

- orderby 句は指定せず、from 句と group 句を指定したクエリ式

  from x in e group v by g

  は次のように変換されます。

```
( e ) . GroupBy ( x => g , x => v )
```

  v が識別子 x の場合を除き、変換は次のようになります。

```
( e ) . GroupBy ( x => g )
```

- from 句、orderby 句、select 句を指定したクエリ式

  from x in e orderby k_1 , k_2 ... select v

は次のように変換されます。

```
( e ) . OrderBy ( x => k1 ) . ThenBy ( x => k2 ) ... . Select ( x => v )
```

v が識別子 *x* の場合を除き、変換は次のようになります。

```
( e ) . OrderBy ( x => k1 ) . ThenBy ( x => k2 ) ...
```

- `from` 句、`orderby` 句、`group` 句を指定したクエリ式

```
from x in e orderby k1 , k2 ... group v by g
```

は次のように変換されます。

```
( e ) . OrderBy ( x => k1 ) . ThenBy ( x => k2 ) ...
. GroupBy ( x => g , x => v )
```

v が識別子 *x* の場合を除き、変換は次のようになります。

```
( e ) . OrderBy ( x => k1 ) . ThenBy ( x => k2 ) ... . GroupBy ( x => g )
```

26.8 式階層

式階層では、ラムダ式を実行可能なコードの代わりにデータ構造として表すことができます。データ型 `D` に変換可能なラムダ式は、型 `System.Query.Expression<D>` の式階層にも変換可能です。ラムダ式をデリゲート型に変換すると、実行可能なコードが生成され、デリゲートにより参照されるのに対し、式階層型に変換すると、式階層インスタンスを作成するコードが送出されます。式階層は、ラムダ式の効果的なメモリ内データ表示であり、式の構造が透過的で明示的になります。

次の例は、ラムダ式を実行可能なコードと式階層の両方として表しています。`Func<int,int>` への変換が存在するので、`Expression<Func<int,int>>` への変換も存在します。

```
Func<int,int> f = x => x + 1;                  // コード
Expression<Func<int,int>> e = x => x + 1;      // データ
```

これらの引数に続き、デリゲート `f` は、`x + 1` を返すメソッドを参照し、式階層 `e` は、式 `x + 1` を記述するデータ構造を参照します。

注意

式階層の構造は別の仕様で説明されます。PDC 2005 Technology Preview はこの仕様を利用できません。