

# C#

言語仕様

Version 5.0

**注意**

*(C) 1999-2012 Microsoft Corporation. All Rights Reserved.*

*Microsoft、Windows、Visual Basic、Visual C#、およびVisual C++ は米国およびその他の国/地域の Microsoft Corporation の登録商標または商標です。*

*本書内のその他の製品名や会社名は、各社の商標である場合もあります。*

## 目次

<b>1. 概要.....</b>	<b>1</b>
1.1 Hello world.....	1
1.2 プログラム構造.....	2
1.3 型と変数.....	4
1.4 式 .....	8
1.5 ステートメント .....	10
1.6 クラスとオブジェクト .....	15
1.6.1 メンバー .....	16
1.6.2 アクセシビリティ .....	16
1.6.3 型パラメーター .....	17
1.6.4 基底クラス .....	17
1.6.5 フィールド .....	18
1.6.6 メソッド .....	18
1.6.6.1 パラメーター.....	19
1.6.6.2 メソッド本体とローカル変数.....	20
1.6.6.3 静的メソッドとインスタンスメソッド .....	20
1.6.6.4 仮想メソッド、オーバーライドメソッド、抽象メソッド .....	22
1.6.6.5 メソッドのオーバーロード .....	24
1.6.7 その他の関数メンバー.....	24
1.6.7.1 コンストラクター.....	26
1.6.7.2 プロパティ .....	27
1.6.7.3 インデクサー .....	27
1.6.7.4 イベント .....	28
1.6.7.5 演算子 .....	29
1.6.7.6 デストラクター .....	29
1.7 構造体 .....	29
1.8 配列 .....	31
1.9 インターフェイス .....	32
1.10 列挙型.....	33
1.11 デリゲート .....	35
1.12 属性 .....	36
<b>2. 構文の構造.....</b>	<b>39</b>
2.1 プログラム .....	39
2.2 文法 .....	39
2.2.1 文法表記 .....	39
2.2.2 字句文法 .....	40
2.2.3 構文文法 .....	40
2.3 字句解析 .....	41
2.3.1 行末記号 .....	41
2.3.2 コメント .....	42
2.3.3 空白 .....	43
2.4 トークン .....	43
2.4.1 Unicode 文字のエスケープ シーケンス .....	44

2.4.2 識別子 .....	44
2.4.3 キーワード .....	46
2.4.4 リテラル .....	47
2.4.4.1 ブール型リテラル .....	47
2.4.4.2 整数リテラル .....	47
2.4.4.3 実数リテラル .....	48
2.4.4.4 文字リテラル .....	49
2.4.4.5 リテラル文字列 .....	50
2.4.4.6 null リテラル <.....	52
2.4.5 演算子と区切り記号 .....	52
2.5 プリプロセッサディレクティブ .....	52
2.5.1 条件付きコンパイルシンボル .....	54
2.5.2 プリプロセス式 .....	54
2.5.3 宣言ディレクティブ .....	55
2.5.4 条件付きコンパイルディレクティブ .....	56
2.5.5 診断ディレクティブ .....	59
2.5.6 領域ディレクティブ .....	59
2.5.7 行ディレクティブ .....	60
2.5.8 pragma ディレクティブ .....	61
2.5.8.1 pragma warning .....	61
3. 基本概念 .....	63
3.1 アプリケーションの起動 .....	63
3.2 アプリケーションの終了 .....	64
3.3 宣言 .....	64
3.4 メンバー .....	67
3.4.1 名前空間のメンバー .....	67
3.4.2 構造体のメンバー .....	67
3.4.3 列挙型メンバー .....	68
3.4.4 クラスのメンバー .....	68
3.4.5 インターフェイスのメンバー .....	68
3.4.6 配列のメンバー .....	68
3.4.7 デリゲートのメンバー .....	68
3.5 メンバーアクセス .....	69
3.5.1 宣言されたアクセシビリティ .....	69
3.5.2 アクセシビリティドメイン .....	70
3.5.3 インスタンスマンバーへのプロテクトアクセス .....	73
3.5.4 アクセシビリティの制約 .....	74
3.6 シグネチャとオーバーロード .....	75
3.7 スコープ .....	76
3.7.1 名前の隠ぺい .....	79
3.7.1.1 入れ子による隠ぺい .....	79
3.7.1.2 継承による隠ぺい .....	80
3.8 名前空間と型の名前 .....	81
3.8.1 完全修飾名 .....	84

3.9 自動メモリ管理 .....	85
3.10 実行の順序 .....	87
<b>4. 型.....</b>	<b>89</b>
4.1 値型 .....	89
4.1.1 System.ValueType 型 .....	90
4.1.2 既定のコンストラクター .....	90
4.1.3 構造型体 .....	91
4.1.4 単純型 .....	91
4.1.5 整数型 .....	92
4.1.6 浮動小数点型 .....	94
4.1.7 Decimal 型 .....	95
4.1.8 bool 型 .....	95
4.1.9 列挙型 .....	96
4.1.10 null 許容型 .....	96
4.2 参照型 .....	96
4.2.1 クラス型 .....	97
4.2.2 オブジェクト型 .....	98
4.2.3 動的な型 .....	98
4.2.4 文字列型 .....	98
4.2.5 インターフェイス型 .....	98
4.2.6 配列型 .....	98
4.2.7 デリゲート型 .....	99
4.3 ボックス化とボックス化解除 .....	99
4.3.1 ボックス化変換 .....	99
4.3.2 ボックス化解除変換 .....	100
4.4 構築された型 .....	101
4.4.1 型引数 .....	102
4.4.2 オープン型とクローズ型 .....	102
4.4.3 バインド型と非バインド型 .....	103
4.4.4 制約の充足 .....	103
4.5 型パラメーター .....	104
4.6 式ツリー型 .....	105
4.7 動的な型 .....	106
<b>5. 変数.....</b>	<b>107</b>
5.1 変数のカテゴリ .....	107
5.1.1 静的変数 .....	107
5.1.2 インスタンス変数 .....	107
5.1.2.1 クラスのインスタンス変数 .....	108
5.1.2.2 構造体のインスタンス変数 .....	108
5.1.3 配列要素 .....	108
5.1.4 値パラメーター .....	108
5.1.5 参照パラメーター .....	108
5.1.6 出力パラメーター .....	109
5.1.7 ローカル変数 .....	109

5.2 既定値.....	110
5.3 確実な代入 .....	110
5.3.1 初期代入ありの変数 .....	111
5.3.2 初期代入なしの変数 .....	112
5.3.3 確実な代入を判断するための正確な規則.....	112
5.3.3.1 ステートメントの一般的な規則.....	112
5.3.3.2 ブロックステートメント、checkedステートメント、およびunchecked ステートメント.....	113
5.3.3.3 式ステートメント.....	113
5.3.3.4 宣言ステートメント .....	113
5.3.3.5 If ステートメント .....	113
5.3.3.6 switch ステートメント .....	114
5.3.3.7 While ステートメント .....	114
5.3.3.8 Do ステートメント .....	114
5.3.3.9 For ステートメント .....	114
5.3.3.10 Break ステートメント、Continue ステートメント、およびGoto ステートメント .....	115
5.3.3.11 Throw ステートメント .....	115
5.3.3.12 Return ステートメント .....	115
5.3.3.13 Try-catch ステートメント .....	115
5.3.3.14 Try-finally ステートメント .....	116
5.3.3.15 Try-catch-finally ステートメント .....	116
5.3.3.16 Foreach ステートメント .....	117
5.3.3.17 Using ステートメント .....	117
5.3.3.18 Lock ステートメント .....	117
5.3.3.19 yield ステートメント .....	118
5.3.3.20 単純な式の一般的な規則.....	118
5.3.3.21 埋め込まれた式を持つ式の一般的な規則 .....	118
5.3.3.22呼び出し式とオブジェクト作成式.....	118
5.3.3.23 簡単な代入式.....	119
5.3.3.24 && 式.....	119
5.3.3.25    式 .....	120
5.3.3.26 !式 .....	121
5.3.3.27 ?? 式 .....	121
5.3.3.28 ?: 式 .....	122
5.3.3.29 匿名関数 .....	122
5.4 変数参照 .....	123
5.5 変数参照の分割不能性 .....	123
<b>6. 変換.....</b>	<b>125</b>
6.1 暗黙の型変換 .....	125
6.1.1 恒等変換 .....	126
6.1.2 暗黙の数値変換 .....	126
6.1.3 暗黙の列挙値変換 .....	126
6.1.4 暗黙の null 許容変換 .....	127
6.1.5 null リテラル変換 .....	127

6.1.6 暗黙の参照変換 .....	127
6.1.7 ボックス化変換 .....	128
6.1.8 暗黙の動的変換 .....	128
6.1.9 暗黙の定数式変換 .....	129
6.1.10 型パラメーターを使用する暗黙の変換.....	129
6.1.11 ユーザ一定義の暗黙の変換.....	130
6.1.12 匿名関数の変換とメソッド グループの変換.....	130
6.2 明示的な変換 .....	130
6.2.1 明示的な数値変換の一覧表.....	130
6.2.2 明示的な列挙値変換 .....	132
6.2.3 明示的な null 許容変換 .....	132
6.2.4 明示的な参照変換 .....	133
6.2.5 ボックス化解除変換 .....	134
6.2.6 明示的な動的変換 .....	135
6.2.7 型パラメーターを使用する明示的な変換.....	135
6.2.8 ユーザ一定義の明示的な変換.....	136
6.3 標準変換 .....	136
6.3.1 標準暗黙変換 .....	136
6.3.2 標準明示変換 .....	137
6.4 ユーザ一定義変換 .....	137
6.4.1 使用できるユーザ一定義変換.....	137
6.4.2 リフト変換演算子 .....	137
6.4.3 ユーザ一定義変換の評価.....	137
6.4.4 ユーザ一定義の暗黙の変換.....	138
6.4.5 ユーザ一定義の明示的な変換.....	139
6.5 匿名関数の変換 .....	141
6.5.1 デリゲート型への匿名関数の変換の評価.....	142
6.5.2 式ツリー型への匿名関数の変換の評価.....	143
6.5.3 実装例 .....	143
6.6 メソッドのグループの変換 .....	146
<b>7. 式.....</b>	<b>149</b>
7.1 式の分類 .....	149
7.1.1 式の値 .....	150
7.2 静的バインディングと動的バインディング .....	150
7.2.1 バインディング時 .....	151
7.2.2 動的バインディング .....	151
7.2.3 構成する式の型 .....	152
7.3 演算子 .....	152
7.3.1 演算子の優先順位と結合規則.....	153
7.3.2 演算子のオーバーロード .....	154
7.3.3 単項演算子のオーバーロードの解決.....	155
7.3.4 二項演算子のオーバーロードの解決.....	155
7.3.5 ユーザ一定義演算子候補.....	156
7.3.6 数値の上位変換 .....	156

7.3.6.1 単数値上位変換.....	157
7.3.6.2 二項数値上位変換.....	157
7.3.7 リフト演算子 .....	158
7.4 メンバー検索 .....	159
7.4.1 基本型 .....	160
7.5 関数メンバー .....	160
7.5.1 引数リスト .....	163
7.5.1.1 対応するパラメーター .....	165
7.5.1.2 引数リストの実行時の評価 .....	166
7.5.2 型推論 .....	167
7.5.2.1 第1フェーズ .....	168
7.5.2.2 第2フェーズ .....	168
7.5.2.3 入力型 .....	169
7.5.2.4 出力型 .....	169
7.5.2.5 依存関係 .....	169
7.5.2.6 出力型推論 .....	169
7.5.2.7 明示的なパラメーター型推論 .....	169
7.5.2.8 正確な推論 .....	169
7.5.2.9 下限の推論 .....	170
7.5.2.10 上限の推論 .....	170
7.5.2.11 固定 .....	171
7.5.2.12 推論された戻り値型 .....	171
7.5.2.13 メソッドグループの変換での型推論 .....	173
7.5.2.14 式の集合の最適な共通の型の特定 .....	173
7.5.3 オーバーロードの解決法 .....	174
7.5.3.1 適用可能な関数メンバー .....	174
7.5.3.2 より適切な関数メンバー .....	175
7.5.3.3 式からの "より適切な変換" .....	176
7.5.3.4 型からの "より適切な変換" .....	177
7.5.3.5 より適切な変換対象 .....	177
7.5.3.6 ジェネリック クラスのオーバーロード .....	177
7.5.4 動的なオーバーロードの解決のコンパイル時チェック .....	178
7.5.5 関数メンバーの呼び出し .....	178
7.5.5.1 ボックス化されたインスタンスでの呼び出し .....	180
7.6 基本式 .....	180
7.6.1 リテラル .....	181
7.6.2 簡易名 .....	181
7.6.2.1 複数のブロック内での意味の一致 .....	182
7.6.3 かっこで囲まれた式 .....	183
7.6.4 メンバー アクセス .....	184
7.6.4.1 同一の簡易名と型名 .....	186
7.6.4.2 文法のあいまいさ .....	186
7.6.5 呼び出し式 .....	187
7.6.5.1 メソッドの呼び出し .....	188
7.6.5.2 拡張メソッド呼び出し .....	189

7.6.5.3 デリゲートの呼び出し .....	192
7.6.6 要素へのアクセス .....	192
7.6.6.1 配列アクセス .....	192
7.6.6.2 インデクサー アクセス .....	193
7.6.7 this-access .....	194
7.6.8 base-access .....	195
7.6.9 後置インクリメント演算子と後置デクリメント演算子 .....	195
7.6.10 new 演算子 .....	196
7.6.10.1 オブジェクト作成式 .....	197
7.6.10.2 オブジェクト初期化子 .....	198
7.6.10.3 コレクション初期化子 .....	200
7.6.10.4 配列作成式 .....	202
7.6.10.5 デリゲート作成式 .....	204
7.6.10.6 匿名オブジェクト作成式 .....	205
7.6.11 typeof 演算子 .....	207
7.6.12 checked 演算子と unchecked 演算子 .....	209
7.6.13 既定値の式 .....	211
7.6.14 匿名メソッドの式 .....	211
7.7 単項演算子 .....	211
7.7.1 単項プラス演算子 .....	212
7.7.2 単項マイナス演算子 .....	212
7.7.3 論理否定演算子 .....	213
7.7.4 ビットごとの補数演算子 .....	213
7.7.5 前置インクリメント演算子と前置デクリメント演算子 .....	213
7.7.6 キャスト式 .....	214
7.7.7 Await 式 .....	215
7.7.7.1 待機可能な式 .....	216
7.7.7.2 await 式の分類 .....	216
7.7.7.3 await 式の実行時評価 .....	216
7.8 算術演算子 .....	217
7.8.1 乗算演算子 .....	217
7.8.2 除算演算子 .....	218
7.8.3 剰余演算子 .....	219
7.8.4 加算演算子 .....	220
7.8.5 減算演算子 .....	222
7.9 シフト演算子 .....	224
7.10 関係演算子と型検査演算子 .....	225
7.10.1 整数比較演算子 .....	226
7.10.2 浮動小数点数比較演算子 .....	227
7.10.3 10進数比較演算子 .....	228
7.10.4 ブール等値演算子 .....	228
7.10.5 列挙比較演算子 .....	228
7.10.6 参照型等値演算子 .....	228
7.10.7 文字列等値演算子 .....	230
7.10.8 デリゲート等値演算子 .....	230

7.10.9 等値演算子と null .....	231
7.10.10 is 演算子 .....	231
7.10.11 as 演算子 .....	232
7.11 論理演算子 .....	233
7.11.1 整数論理演算子 .....	233
7.11.2 列挙論理演算子 .....	233
7.11.3 ブール論理演算子 .....	234
7.11.4 null 許容ブール論理演算子 .....	234
7.12 条件論理演算子 .....	235
7.12.1 ブール条件論理演算子 .....	235
7.12.2 ユーザー定義の条件論理演算子 .....	236
7.13 null 合体演算子 .....	236
7.14 条件演算子 .....	237
7.15 匿名関数の式 .....	238
7.15.1 匿名関数のシグネチャ .....	240
7.15.2 匿名関数の本体 .....	240
7.15.3 オーバーロードの解決法 .....	241
7.15.4 匿名関数と動的バインディング .....	242
7.15.5 外部変数 .....	242
7.15.5.1 キャプチャされた外部変数 .....	242
7.15.5.2 ローカル変数のインスタンス化 .....	243
7.15.6 匿名関数式の評価 .....	245
7.16 クエリ式 .....	245
7.16.1 クエリ式のあいまいさ .....	246
7.16.2 クエリ式の書き換え .....	247
7.16.2.1 連続した select 句および groupby 句 .....	247
7.16.2.2 明示的な範囲変数型 .....	248
7.16.2.3 変質クエリ式 .....	248
7.16.2.4 from 句、let 句、where 句、join 句、orderby 句 .....	249
7.16.2.5 Select 句 .....	252
7.16.2.6 Groupby 句 .....	252
7.16.2.7 透過的識別子 .....	253
7.16.3 クエリ式パターン .....	254
7.17 代入演算子 .....	255
7.17.1 単純代入 .....	256
7.17.2 複合代入 .....	258
7.17.3 イベント代入 .....	259
7.18 式 .....	260
7.19 定数式 .....	260
7.20 Boolean 式 .....	261
<b>8. ステートメント .....</b>	<b>263</b>
8.1 終了点と到達可能性 .....	263
8.2 ブロック .....	265
8.2.1 ステートメントリスト .....	266

8.3 空のステートメント .....	266
8.4 ラベル付きステートメント .....	267
8.5 宣言ステートメント .....	267
8.5.1 ローカル変数宣言 .....	267
8.5.2 ローカル定数宣言 .....	269
8.6 式ステートメント .....	269
8.7 選択ステートメント .....	270
8.7.1 if ステートメント .....	270
8.7.2 switch ステートメント .....	271
8.8 繰り返しステートメント .....	275
8.8.1 while ステートメント .....	275
8.8.2 do ステートメント .....	275
8.8.3 for ステートメント .....	276
8.8.4 foreach ステートメント .....	277
8.9 ジャンプステートメント .....	281
8.9.1 break ステートメント .....	282
8.9.2 continue ステートメント .....	282
8.9.3 goto ステートメント .....	283
8.9.4 return ステートメント .....	284
8.9.5 throw ステートメント .....	285
8.10 try ステートメント .....	286
8.11 checked ステートメントおよび unchecked ステートメント .....	289
8.12 lock ステートメント .....	290
8.13 using ステートメント .....	290
8.14 yield ステートメント .....	293
<b>9. 名前空間 .....</b>	<b>295</b>
9.1 コンパイル単位 .....	295
9.2 名前空間の宣言 .....	295
9.3 extern エイリアス .....	297
9.4 Using ディレクティブ .....	297
9.4.1 Using alias ディレクティブ .....	298
9.4.2 Using namespace ディレクティブ .....	300
9.5 名前空間のメンバー .....	302
9.6 型の宣言 .....	303
9.7 名前空間エイリアス修飾子 .....	303
9.7.1 エイリアスの一意性 .....	304
<b>10. クラス .....</b>	<b>307</b>
10.1 クラス宣言 .....	307
10.1.1 クラス修飾子 .....	307
10.1.1.1 抽象クラス .....	308
10.1.1.2 シールクラス .....	308
10.1.1.3 静的クラス .....	309
10.1.2 Partial 修飾子 .....	310
10.1.3 型パラメーター .....	310

10.1.4 Class base 指定.....	310
10.1.4.1 基底クラス .....	310
10.1.4.2 インターフェイスの実装.....	312
10.1.5 型パラメーターの制約.....	312
10.1.6 クラス本体 .....	317
10.2 部分型.....	317
10.2.1 属性 .....	318
10.2.2 修飾子 .....	318
10.2.3 型パラメーターと制約.....	318
10.2.4 基底クラス .....	319
10.2.5 基本インターフェイス.....	319
10.2.6 メンバー .....	320
10.2.7 部分メソッド .....	320
10.2.8 名前のバインディング .....	322
10.3 クラスのメンバー .....	323
10.3.1 インスタンス型 .....	324
10.3.2 構築された型のメンバー.....	325
10.3.3 継承 .....	326
10.3.4 new 修飾子.....	327
10.3.5 アクセス修飾子 .....	327
10.3.6 構成要素型 .....	327
10.3.7 静的メンバーとインスタンスメンバー .....	327
10.3.8 入れ子になった型.....	329
10.3.8.1 完全修飾名 .....	329
10.3.8.2 宣言されたアクセシビリティ .....	329
10.3.8.3 隠ぺい .....	330
10.3.8.4 this アクセス .....	330
10.3.8.5 包含する型の private メンバーおよび protected メンバーへのアクセス .....	331
10.3.8.6 ジェネリック クラスの入れ子になった型 .....	332
10.3.9 予約済みのメンバー名 .....	333
10.3.9.1 プロパティ用に予約済みのメンバー名 .....	333
10.3.9.2 イベント用に予約済みのメンバー名 .....	334
10.3.9.3 インデクサー用に予約済みのメンバー名 .....	334
10.3.9.4 デストラクター用に予約済みのメンバー名 .....	334
10.4 定数 .....	334
10.5 フィールド .....	336
10.5.1 静的フィールドとインスタンスフィールド .....	337
10.5.2 Readonly フィールド .....	338
10.5.2.1 定数への静的 readonly フィールドの使用 .....	338
10.5.2.2 定数と静的 readonly フィールドのバージョン管理 .....	339
10.5.3 Volatile フィールド .....	339
10.5.4 フィールドの初期化 .....	341
10.5.5 変数初期化子 .....	341
10.5.5.1 静的フィールドの初期化 .....	342
10.5.5.2 インスタンスフィールドの初期化 .....	343

10.6 メソッド .....	344
10.6.1 メソッドのパラメーター .....	346
10.6.1.1 値パラメーター .....	348
10.6.1.2 参照パラメーター .....	348
10.6.1.3 出力パラメーター .....	349
10.6.1.4 パラメーター配列 .....	350
10.6.2 静的メソッドとインスタンスマソッド .....	353
10.6.3 仮想メソッド .....	353
10.6.4 オーバーライドメソッド .....	355
10.6.5 シールメソッド .....	357
10.6.6 抽象メソッド .....	358
10.6.7 外部メソッド .....	359
10.6.8 部分メソッド .....	360
10.6.9 拡張メソッド .....	360
10.6.10 メソッド本体 .....	361
10.6.11 メソッドのオーバーロード .....	362
10.7 プロパティ .....	362
10.7.1 静的プロパティとインスタンスプロパティ .....	363
10.7.2 アクセサー .....	363
10.7.3 自動実装プロパティ .....	369
10.7.4 アクセシビリティ .....	370
10.7.5 仮想、シール、オーバーライド、抽象の各アクセサー .....	371
10.8 イベント .....	373
10.8.1 フィールドのように使用するイベント .....	375
10.8.2 イベントアクセサー .....	377
10.8.3 静的イベントとインスタンスイベント .....	378
10.8.4 仮想、シール、オーバーライド、抽象の各アクセサー .....	378
10.9 インデクサー .....	379
10.9.1 インデクサーのオーバーロード .....	382
10.10 演算子 .....	383
10.10.1 単項演算子 .....	384
10.10.2 二項演算子 .....	385
10.10.3 変換演算子 .....	386
10.11 インスタンスコンストラクター .....	388
10.11.1 コンストラクター初期化子 .....	389
10.11.2 インスタンス変数初期化子 .....	390
10.11.3 コンストラクターの実行 .....	390
10.11.4 既定のコンストラクター .....	392
10.11.5 プライベートコンストラクター .....	393
10.11.6 省略可能なインスタンスコンストラクターパラメーター .....	393
10.12 静的コンストラクター .....	394
10.13 デストラクター .....	396
10.14 反復子 .....	398
10.14.1 列挙子インターフェイス .....	398
10.14.2 列挙可能なインターフェイス .....	398

10.14.3 yield 型 .....	398
10.14.4 列挙子オブジェクト .....	398
10.14.4.1 MoveNext メソッド .....	399
10.14.4.2 Current プロパティ .....	400
10.14.4.3 Dispose メソッド .....	400
10.14.5 列挙可能なオブジェクト .....	401
10.14.5.1 GetEnumerator メソッド .....	401
10.14.6 実装例 .....	402
10.15 非同期関数 .....	408
10.15.1 Task を返す非同期関数の評価 .....	408
10.15.2 void を返す非同期関数の評価 .....	408
<b>11. 構造体.....</b>	<b>410</b>
11.1 構造体の宣言 .....	410
11.1.1 構造体修飾子 .....	410
11.1.2 Partial 修飾子 .....	411
11.1.3 構造体インターフェイス .....	411
11.1.4 構造体の本体 .....	411
11.2 構造体のメンバー .....	411
11.3 クラスと構造体の違い .....	411
11.3.1 値のセマンティクス .....	412
11.3.2 繙承 .....	413
11.3.3 代入 .....	413
11.3.4 既定値 .....	413
11.3.5 ボックス化とボックス化解除 .....	414
11.3.6 this の概念 .....	416
11.3.7 フィールド初期化子 .....	416
11.3.8 コンストラクター .....	416
11.3.9 デストラクター .....	417
11.3.10 静的コンストラクター .....	417
11.4 構造体の例 .....	418
11.4.1 データベースの整数型 .....	418
11.4.2 データベースのブール型 .....	419
<b>12. 配列.....</b>	<b>423</b>
12.1 配列型 .....	423
12.1.1 System.Array 型 .....	424
12.1.2 配列とジェネリック IList インターフェイス .....	424
12.2 配列の作成 .....	425
12.3 配列要素へのアクセス .....	425
12.4 配列のメンバー .....	425
12.5 配列の共変性 .....	425
12.6 配列初期化子 .....	426
<b>13. インターフェイス.....</b>	<b>429</b>
13.1 インターフェイスの宣言 .....	429

13.1.1 インターフェイス修飾子.....	429
13.1.2 Partial 修飾子 .....	430
13.1.3 バリアント型パラメーターリスト.....	430
13.1.3.1 変性の安全性.....	430
13.1.3.2 変性変換.....	431
13.1.4 基本インターフェイス.....	431
13.1.5 インターフェイスの本体.....	432
13.2 インターフェイスのメンバー .....	432
13.2.1 インターフェイスのメソッド.....	433
13.2.2 インターフェイスのプロパティ .....	434
13.2.3 インターフェイスのイベント.....	435
13.2.4 インターフェイスのインデクサー.....	435
13.2.5 インターフェイスのメンバーへのアクセス.....	435
13.3 インターフェイス メンバーの完全修飾名 .....	437
13.4 インターフェイスの実装 .....	438
13.4.1 インターフェイス メンバーの明示的実装.....	439
13.4.2 実装されるインターフェイスの一意性.....	441
13.4.3 ジェネリック メソッドの実装.....	442
13.4.4 インターフェイス マップ .....	443
13.4.5 インターフェイス実装の継承.....	446
13.4.6 インターフェイスの再実装.....	447
13.4.7 抽象クラスとインターフェイス .....	448
<b>14. 列挙型.....</b>	<b>451</b>
14.1 列挙型の宣言 .....	451
14.2 列挙修飾子 .....	451
14.3 列挙型のメンバー .....	452
14.4 System.Enum 型.....	454
14.5 列挙型の値と演算 .....	454
<b>15. デリゲート .....</b>	<b>455</b>
15.1 デリゲートの宣言 .....	455
15.2 デリゲートの互換性 .....	458
15.3 デリゲートのインスタンス化 .....	458
15.4 デリゲートの呼び出し .....	458
<b>16. 例外.....</b>	<b>461</b>
16.1 例外の原因 .....	461
16.2 System.Exception クラス .....	461
16.3 例外の処理方法 .....	461
16.4 共通の例外クラス .....	462
<b>17. 属性.....</b>	<b>465</b>
17.1 属性クラス .....	465
17.1.1 属性の使用法 .....	465
17.1.2 位置指定パラメーターと名前付きパラメーター .....	467

17.1.3 属性パラメータ一型.....	467
17.2 属性の指定 .....	468
17.3 属性インスタンス .....	473
17.3.1 属性のコンパイル.....	473
17.3.2 属性インスタンスの実行時取得.....	474
17.4 予約済み属性 .....	474
17.4.1 AttributeUsage 属性.....	474
17.4.2 Conditional 属性.....	475
17.4.2.1 条件付きメソッド.....	475
17.4.2.2 条件付き属性クラス.....	478
17.4.3 Obsolete 属性 .....	478
17.4.4 呼び出し元情報属性.....	479
17.4.4.1 CallerLineNumber 属性.....	480
17.4.4.2 CallerFilePath 属性.....	481
17.4.4.3 CallerMemberName 属性.....	481
17.5 相互運用の属性 .....	481
17.5.1 COM コンポーネントと Win32 コンポーネントとの相互運用.....	481
17.5.2 他の .NET 言語との相互運用.....	482
17.5.2.1 IndexerName 属性.....	482
<b>18. 安全でないコード.....</b>	<b>483</b>
18.1 Unsafe コンテキスト .....	483
18.2 ポインター型 .....	486
18.3 固定変数と移動可能変数 .....	489
18.4 ポインター変換 .....	489
18.4.1 ポインター配列 .....	490
18.5 式のポインター .....	491
18.5.1 ポインターの間接参照 .....	492
18.5.2 ポインター メンバー アクセス .....	492
18.5.3 ポインター要素アクセス .....	493
18.5.4 アドレス演算子 .....	494
18.5.5 ポインターのインクリメントとデクリメント .....	495
18.5.6 ポインターの算術演算 .....	495
18.5.7 ポインターの比較 .....	496
18.5.8 sizeof 演算子 .....	496
18.6 fixed ステートメント .....	497
18.7 固定サイズバッファー .....	501
18.7.1 固定サイズバッファーの宣言 .....	501
18.7.2 式の固定サイズバッファー .....	502
18.7.3 確実な代入のチェック .....	503
18.8 スタック割り当て .....	504
18.9 動的メモリ割り当て .....	505
<b>A. ドキュメントのコメント .....</b>	<b>507</b>
A.1 概要 .....	507
A.2 推奨されるタグ .....	508

A.2.1 <c> \t "See <c>" \b.....	509
A.2.2 <code> \t "See <code>" \b .....	510
A.2.3 <example> \t "See <example>" \b .....	510
A.2.4 <exception> \t "See <exception>" \b .....	510
A.2.5 <include> .....	511
A.2.6 <list> \t "See <list>" \b .....	512
A.2.7 <para> \t "See <para>" \b .....	512
A.2.8 <param> \t "See <param>" \b .....	513
A.2.9 <paramref> \t "See <paramref>" \b .....	513
A.2.10 <permission> \t "See <permission>" \b .....	514
A.2.11 <remark> \t "See <remarks>" \b .....	514
A.2.12 <returns> \t "See <returns>" \b .....	514
A.2.13 <see> \t "See <see>" \b .....	515
A.2.14 <seealso> \t "See <seealso>" \b .....	515
A.2.15 <summary> \t "See <summary>" \b.....	516
A.2.16 <value> \t "See <value>" \b .....	516
A.2.17 <typeparam> .....	516
A.2.18 <typeparamref>.....	517
A.3 ドキュメントファイルの処理 .....	517
A.3.1 ID 文字列形式 .....	517
A.3.2 ID 文字列の例 .....	519
A.4 例 .....	522
A.4.1 C# のソース コード .....	522
A.4.2 生成される XML.....	524
<b>B. 文法 .....</b>	<b>530</b>
B.1 字句文法 .....	530
B.1.1 行末記号 .....	530
B.1.2 コメント .....	530
B.1.3 空白 .....	531
B.1.4 トークン .....	531
B.1.5 Unicode 文字のエスケープ シーケンス .....	531
B.1.6 識別子 .....	531
B.1.7 キーワード .....	533
B.1.8 リテラル .....	533
B.1.9 演算子と区切り記号 .....	535
B.1.10 プリプロセッサ ディレクティブ .....	535
B.2 構文文法 .....	538
B.2.1 基本概念 .....	538
B.2.2 型 .....	538
B.2.3 変数 .....	539
B.2.4 式 .....	539
B.2.5 ステートメント .....	546
B.2.6 名前空間 .....	550
B.2.7 クラス .....	550
B.2.8 構造体 .....	558
B.2.9 配列 .....	558

## C# LANGUAGE SPECIFICATION

B.2.10 インターフェイス .....	559
B.2.11 列挙型 .....	560
B.2.12 デリゲート .....	561
B.2.13 属性 .....	561
B.3 安全でないコードのための文法 .....	562
<b>C. リファレンス .....</b>	<b>567</b>

# 1. 概要

C# ("シー シャープ" と読みます) は、オブジェクト指向とタイプセーフという特徴を備えた最新の簡単なプログラミング言語です。C# は C 系言語を起源とし、C、C++、および Java プログラマであればすぐに操作に慣れることができます。C# は、ECMA International では **ECMA-334** 標準として、ISO/IEC では **ISO/IEC 23270** 標準として標準規格化されています。Microsoft の .NET Framework 用 C# コンパイラは、この 2 つの標準規格の両方に準拠した実装です。

C# はオブジェクト指向言語ですが、C# ではさらにコンポーネント指向プログラミングもサポートしています。最近のソフトウェアデザインでは、機能が独立型および自己記述型パッケージの形になっているソフトウェアコンポーネントへの依存がますます高まっています。このようなコンポーネントで重要な点は、プログラミングモデルをプロパティ、メソッド、およびイベントを使って表すことです。これらの要素はコンポーネントに関する宣言情報を提供する属性を持ち、独自のドキュメントを組み込んでいます。C# はこれらの概念を直接サポートする言語構成要素を備えているため、ソフトウェアコンポーネントの作成と使用に非常に適しています。

C# には堅牢で永続性のあるアプリケーションの構築を支援する機能がいくつかあります。ガベージコレクションは、未使用のオブジェクトで占められているメモリを自動的にクリアします。例外処理は、エラーの検出と回復に対して構造化された拡張可能なアプローチを提供します。タイプセーフな言語のデザインは、初期化されていない変数からの読み取り、限界を超えた配列のインデックス、およびチェックなしの型キャストの実行を不可能にします。

C# では、**統一型システム (unified type system)** が採用されています。`int` や `double` などのプリミティブ型を含む C# のすべての型は、单一のルートである `object` 型だけから継承されます。したがって、一連の一般的な操作はすべての型に共通で、どの型の値も一貫した方法で格納、トランスポート、および操作できます。さらに、C# はユーザー一定義の参照型と値型の両方をサポートするため、ワンドウレス構造のインラインストレージ以外に、オブジェクトを動的に割り当てることができます。

時間の経過と共に C# のプログラムとライブラリが互換性を維持しながら進化できるように、C# のデザインではバージョン管理がきわめて重要視されてきました。この問題にあまり注意が払われていないプログラミング言語が多いため、新しいバージョンの依存ライブラリが導入されると、そのような言語で記述されたプログラムは必要以上に大きな影響を受けます。バージョン管理の問題について直接影響を受ける C# のデザインの要素には、`virtual` 修飾子と `override` 修飾子の独立、メソッドのオーバーロード解決の規則、明示的なインターフェイスメンバー宣言のサポートなどがあります。

この章では、C# 言語の基本機能について説明します。後の章では、規則と例外について詳細に、場合によっては数学的に説明しますが、この章では、完全さよりも明確さと簡潔さを優先します。早くプログラムを記述できるように、また、後の章の内容を理解しやすいように、この章では言語の概要について説明します。

## 1.1 Hello world

"Hello, World" プログラムは、従来からプログラミング言語の紹介に使用されています。このプログラムを C# で記述した例を次に示します。

```
using System;
```

```
class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

C# のソースファイルの拡張子には、通常は .cs が使用されます。"Hello, World" プログラムが hello.cs ファイルに格納されている場合、このプログラムを Microsoft C# コンパイラによってコマンドラインからコンパイルするには、次のように入力します。

```
csc hello.cs
```

この結果、hello.exe という名前の実行可能アセンブリが生成されます。このアプリケーションを実行すると、次の出力が表示されます。

```
Hello, world
```

"Hello, World" プログラムは、System 名前空間を参照する using ディレクティブから開始します。名前空間により、C# のプログラムとライブラリは階層的に編成されます。名前空間には、型と他の名前空間が含まれます。たとえば、System 名前空間には、プログラムで参照される console クラスなどのいくつかの型と、IO や Collections など、他のいくつかの名前空間が含まれています。特定の名前空間を参照する using ディレクティブを使用することで、その名前空間のメンバーである型は修飾なしで使用できます。using ディレクティブの使用により、プログラムは System.Console.WriteLine の省略である Console.WriteLine を使用できます。

"Hello, World" プログラムで宣言される Hello クラスには、単一のメンバーである Main という名前のメソッドがあります。Main メソッドは、static 修飾子で宣言されます。インスタンス メソッドはキーワード this を使用して外側にある特定のオブジェクトインスタンスを参照できますが、静的メソッドは特定のオブジェクトを参照せずに動作します。規約により、静的メソッド Main は、プログラムのエントリ ポイントとして機能します。

プログラムの出力は、System 名前空間にある console クラスの WriteLine メソッドによって生成されます。このクラスは、既定で Microsoft C# コンパイラによって自動的に参照される .NET Framework のクラス ライブラリから提供されます。C# 自体には独自のランタイム ライブラリがないことに注意してください。その代わり、.NET Framework が C# のランタイム ライブラリになります。

## 1.2 プログラム構造

C# の主な構造上の概念は、**プログラム**、**名前空間**、**型**、**メンバー**、および**アセンブリ**です。C# プログラムは、1つ以上のソースファイルで構成されます。プログラムは型を宣言します。型はメンバーを含み、名前空間として編成できます。型の例として、クラスとインターフェイスがあります。メンバーの例としては、フィールド、メソッド、プロパティ、イベントがあります。C# プログラムはコンパイルされると、アセンブリにパッケージ化されます。通常、アセンブリの拡張子には、"アプリケーション" と "ライブラリ" のどちらを実装するかに応じて、.exe または .dll が付けられます。

次の例を参照してください。

```
using System;
namespace Acme.Collections
{
    public class Stack
    {
        Entry top;
```

**Chapter エラー!** [ホーム] タブを使用して、ここに表示する文字列に Heading 1 を適用してください。 エラー! [ホーム] タブ

```
public void Push(object data) {
    top = new Entry(top, data);
}

public object Pop() {
    if (top == null) throw new InvalidOperationException();
    object result = top.data;
    top = top.next;
    return result;
}

class Entry {
    public Entry next;
    public object data;

    public Entry(Entry next, object data) {
        this.next = next;
        this.data = data;
    }
}
```

前の例は、名前空間 `Acme.Collections` でクラス `Stack` を宣言しています。このクラスの完全修飾名は `Acme.Collections.Stack` です。このクラスには、フィールド `top`、2つのメソッド `Push` と `Pop`、および入れ子になったクラス `Entry` が含まれています。この `Entry` クラスには、`next` と `data` の2つのフィールドとコンストラクターという3つのメンバーが含まれています。この例のソースコードがファイル `acme.cs` に格納されると仮定します。

```
csc /t:library acme.cs
```

上記のコマンドラインは、例をライブラリ (`Main` エントリ ポイントを含まないコード) としてコンパイルし、`acme.dll` という名前のアセンブリを生成します。

アセンブリには、**中間言語(IL: Intermediate Language)** 命令の形の実行可能コードと、**メタデータ**の形のシンボル情報が含まれます。アセンブリの IL コードは実行される前に、.NET 共通言語ランタイムの **Just-In-Time (JIT)** コンパイラにより、自動的にプロセッサ固有のコードに変換されます。

アセンブリはコードとメタデータの両方を含む自己記述型の機能単位であるため、C# では `#include` ディレクティブとヘッダーファイルは不要です。特定のアセンブリ内のパブリック型とパブリックメンバーは、プログラムのコンパイル時にそのアセンブリを参照するだけで C# プログラムで使用できるようになります。たとえば、次のプログラムは `acme.dll` アセンブリの `Acme.Collections.Stack` クラスを使用しています。

```
using System;
using Acme.Collections;

class Test
{
    static void Main() {
        Stack s = new Stack();
        s.Push(1);
        s.Push(10);
        s.Push(100);
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
    }
}
```

プログラムがファイル `test.cs` に格納されている場合、`test.cs` をコンパイルするときに、コンパイラの `/r` オプションを使用して `acme.dll` アセンブリを参照できます。

```
csc /r:acme.dll test.cs
```

これにより、実行可能なアセンブリ `test.exe` が作成され、このアセンブリを実行すると次のような出力が生成されます。

```
100  
10  
1
```

C# ではプログラムのソース テキストを複数のソース ファイルに格納できます。複数ファイルの C# プログラムがコンパイルされる場合、すべてのソース ファイルは同時に処理され、自由に相互参照できます。概念的には、すべてのソース ファイルが 1 つの大容量ファイルに結合されてから処理される場合と同じです。C# では、ごくわずかな例外を除いて宣言の順序は意味を持たないため、事前の宣言は不要です。C# ではソース ファイルでのパブリック型の宣言が 1 つに制限されず、またソース ファイル名とソース ファイルで宣言された型の一致も要求されません。

### 1.3 型と変数

C# では、**値型**と**参照型**という 2 種類の型をサポートしています。値型はデータを直接格納するのに対して、参照型の変数はオブジェクトであるデータへの参照を格納します。参照型では 2 つの変数から同じオブジェクトを参照できるため、ある変数を操作することによって、他の変数が参照しているオブジェクトにも影響する可能性があります。値型では、各変数が独自のデータ コピーを保持し、一方の変数に対する操作が他方の変数に影響を与えることはありません (`ref` パラメーターと `out` パラメーターの変数を除く)。

C# の値型はさらに**単純型**、**列挙型**、**構造型**、**許容型**に分かれています。C# の参照型はさらに**クラス型**、**インターフェイス型**、**配列型**、**デリゲート型**に分かれています。

次の表に C# の型システムの概要を示します。

カテゴリ		説明
値型	単純型	符号付き整数: <code>sbyte</code> 、 <code>short</code> 、 <code>int</code> 、 <code>long</code>
		符号なし整数: <code>byte</code> 、 <code>ushort</code> 、 <code>uint</code> 、 <code>ulong</code>
		Unicode 文字: <code>char</code>
		IEEE 浮動小数点数: <code>float</code> 、 <code>double</code>
		高精度小数: <code>decimal</code>
	ブール演算型	<code>bool</code>
	列挙型	ユーザ一定義の <code>enum E {...}</code> 形式の型
参照型	クラス型	ユーザ一定義の型 <code>struct S {...}</code> 形式
		<code>null</code> 値を持つ他のすべての値型の拡張
		他のすべての型に対する最終的な基底クラス: <code>object</code>
	インターフェイス型	Unicode 文字列: <code>string</code>
		ユーザ一定義の <code>class C {...}</code> 形式の型
	配列型	<code>int[]</code> や <code>int[,]</code> など、1 次元および多次元配列型
	デリゲート型	ユーザ一定義の型の形式、たとえば <code>delegate int D(...)</code>

8 つの整数型は符号付きまたは符号なしの形で 8 ビット、16 ビット、32 ビット、64 ビットの値をサポートします。

2 つの浮動小数点型、`float` と `double` は、32 ビット単精度および 64 ビット倍精度の IEEE 754 形式で表現されます。

`decimal` 型は、財務計算や通貨計算に適した 128 ビットデータ型です。

C# の `bool` 型は、`true` または `false` のブール値を表すために使用されます。

C# の文字および文字列処理は、Unicode エンコーディングを使用します。`char` 型は UTF-16 コード単位を表し、`string` 型は UTF-16 コード単位のシーケンスを表します。

次の表に C# の数値型を要約します。

カテゴリ	ビット	種類	範囲/精度
符号付き 整数	8	<code>sbyte</code>	-128...127
	16	<code>short</code>	-32,768...32,767
	32	<code>int</code>	-2,147,483,648...2,147,483,647
	64	<code>long</code>	-9,223,372,036,854,775,808...9,223,372,036,854,775,807
符号なし 整数	8	<code>byte</code>	0...255
	16	<code>ushort</code>	0...65,535
	32	<code>uint</code>	0...4,294,967,295
	64	<code>ulong</code>	0...18,446,744,073,709,551,615
浮動小数 点数	32	<code>float</code>	$1.5 \times 10^{-45} \sim 3.4 \times 10^{38}$ 、7桁の有効桁数
	64	<code>double</code>	$5.0 \times 10^{-324} \sim 1.7 \times 10^{308}$ 、15桁の有効桁数
10進数	128	<code>decimal</code>	$1.0 \times 10^{-28} \sim 7.9 \times 10^{28}$ 、28桁の有効桁数

C# プログラムでは、**型宣言**を使用して新しい型を作成します。型宣言では、新しい型の名前とメンバーが指定されます。C# の型カテゴリのうち、クラス型、構造体型、インターフェイス型、列挙型、デリゲート型の5つは、ユーザーが定義できます。

クラス型は、データ メンバー(フィールド)と関数メンバー(メソッド、プロパティ、その他)を格納したデータ構造を定義します。クラス型は、派生クラスが基底クラスを拡張および限定するための機構として、单一継承とポリモーフィズムをサポートします。

構造体型は、データ メンバーと関数メンバーで構造を表す点でクラス型と似ています。クラスと異なるのは、構造体は値型であり、ヒープ割り当てを必要としません。構造体型はユーザー指定の継承をサポートしません。すべての構造体型は暗黙的に `object` 型を継承します。

インターフェイス型は、名前付きのパブリック関数メンバーセットとしてコントラクトを定義します。インターフェイスを実装するクラスまたは構造体では、インターフェイスの関数メンバーが実装される必要があります。インターフェイスは複数の基本インターフェイスから継承できます。また、クラスや構造体は複数のインターフェイスを実装できます。

デリゲート型は、特定のパラメーターリストおよび戻り値の型を使用して、メソッドへの参照を表します。デリゲートにより、メソッドを変数に代入し、パラメーターとして渡すことのできるエンティティとして取り扱うことができます。デリゲートは他の一部の言語に見られる関数ポインターの概念に似ていますが、関数ポインターとは異なり、デリゲートはオブジェクト指向でタイプセーフです。

クラス、構造体、インターフェイス、デリゲートの各型はいずれもジェネリックをサポートするので、他の型でパラメーター化できます。

列挙型は、名前付き定数を持つ特別な型です。列挙型にはそれぞれ基になる型があり、これは 8 つの整数型のいずれかである必要があります。列挙型の値の集合は、その基になる型の値の集合と同じです。

C# は任意の型の 1 次元配列および複数次元配列をサポートします。上記の型とは異なり、配列型は使用する前に宣言しておく必要がありません。代わりに、配列型は型名の後に角かっこを続けて構築します。たとえば、`int[]` は `int` の 1 次元配列、`int[,]` は `int` の 2 次元配列、`int[][]` は `int` の 1 次元配列の 1 次元配列です。

`null` 許容型も使用前に宣言しておく必要はありません。`null` 非許容型の値型 `T` には、対応する `null` 許容型 `T?` があります。これには追加の値として `null` を保持することができます。たとえば、`int?` は 32 ビット整数または値 `null` を保持できる型です。

C# の型システムは、すべての型の値をオブジェクトとして扱うことができるように統一されています。C# の各型は、`object` クラス型から直接または間接に派生しており、`object` はすべての型の最終的な基底クラスです。参照型の値は、`object` 型として値を見ることで、簡単にオブジェクトとして扱うことができます。値型の値は、ボックス化とボックス化解除の操作を実行することで、オブジェクトとして扱うことができます。次の例では、値 `int` は `object` に変換され、再び `int` に変換されます。

```
using System;
class Test
{
    static void Main() {
        int i = 123;
        object o = i;           // Boxing
        int j = (int)o;         // Unboxing
    }
}
```

値型の値が `object` 型に変換されると、値を保持するための "ボックス" とも呼ばれるオブジェクトインスタンスが割り当てられ、値はそのボックスにコピーされます。逆に `object` 参照が値型にキャストされると、参照されたオブジェクトが正しい値型のボックスであるかどうかのチェックが行われ、チェックが成功するとボックス内の値がコピーされます。

C# の統一型システムでは、実質的に、値型を "必要に応じて" オブジェクトにすることができます。この統一により、`object` 型を使用する汎用ライブラリは、参照型と値型のどちらにも利用できます。

C# には、フィールド、配列要素、ローカル変数、パラメーターを含む数種類の "変数" があります。変数は格納場所を表し、各変数には型があります。この型により、その変数に格納できる値の種類が次の表に示すように決定されます。

変数の種類	内容
null 非許容値型	正確な型の値
null 許容値型	null 値またはその型の値
object	null 参照、参照型のオブジェクトへの参照、または値型のボックス化された値への参照
クラス型	null 参照、このクラス型のインスタンスへの参照、またはこのクラス型から派生したクラスのインスタンスへの参照
インターフェイス型	null 参照、このインターフェイス型を実装するクラス型のインスタンスの参照、またはこのインターフェイス型を実装する値型のボックス化された値への参照
配列型	null 参照、この配列型のインスタンスへの参照、または互換性のある配列型のインスタンスへの参照
デリゲート型	null 参照またはこのデリゲート型のインスタンスへの参照

## 1.4 式

式は、"オペランド"と"演算子"から構成されます。式の演算子は、オペランドに適用する演算を示します。+、-、\*、/、newなどが演算子の例です。オペランドには、リテラル、フィールド、ローカル変数、式などを指定します。

式に複数の演算子が含まれていると、演算子の"優先順位"によって、各演算子が評価される順序が制御されます。たとえば、\* 演算子は + 演算子よりも優先順位が高いので、 $x + y * z$  という式は  $x + (y * z)$  と評価されます。

ほとんどの演算子は、オーバーロードできます。演算子をオーバーロードすると、一方または両方のオペランドがユーザー定義のクラスまたは構造体型であるような操作に対して、ユーザー定義の演算子の実装を指定できます。

次の表は、C# の演算子をカテゴリの優先順位の高いものから順にまとめたものです。同じカテゴリの演算子の優先順位は同等です。

カテゴリ	式	説明
1 次式	<code>x.m</code>	メンバー アクセス
	<code>x(...)</code>	メソッドとデリゲートの呼び出し
	<code>x[...]</code>	配列とインデクサーへのアクセス
	<code>x++</code>	後置インクリメント
	<code>x--</code>	後置デクリメント
	<code>new T(...)</code>	オブジェクトとデリゲートの作成
	<code>new T(...){...}</code>	初期化子を使用したオブジェクトの作成
	<code>new {...}</code>	匿名オブジェクト初期化子
	<code>new T[...]</code>	配列の作成
	<code>typeof(T)</code>	<code>T</code> の <code>System.Type</code> オブジェクトを取得
	<code>checked(x)</code>	<code>checked</code> (チェックあり) コンテキストの式を評価
	<code>unchecked(x)</code>	<code>unchecked</code> (チェックなし) コンテキストの式を評価
単項式	<code>+x</code>	ID
	<code>-x</code>	否定
	<code>!x</code>	論理否定
	<code>~x</code>	ビットごとの否定
	<code>++x</code>	前置インクリメント
	<code>--x</code>	前置デクリメント
	<code>(T)x</code>	<code>x</code> を型 <code>T</code> に明示的に変換
	<code>await x</code>	<code>x</code> が完了するのを非同期的に待機
乗法演算	<code>x * y</code>	乗算
	<code>x / y</code>	除算
	<code>x % y</code>	剰余

加法演算	$x + y$	加算、文字列の連結、デリゲートの組み合わせ
	$x - y$	減算、デリゲートの削除
シフト	$x \ll y$	左シフト
	$x \gg y$	右シフト
関係式と型検査	$x < y$	より小さい
	$x > y$	より大きい
	$x \leq y$	以下
	$x \geq y$	以上
	$x \text{ is } T$	$x$ が $T$ の場合は <code>true</code> を返し、それ以外の場合は <code>false</code> を返す
	$x \text{ as } T$	$T$ として型指定された $x$ を返し、 $x$ が $T$ でない場合は <code>null</code> を返す
等価比較	$x == y$	等しい
	$x != y$	等しくない
論理 AND	$x \& y$	整数のビット単位の AND、ブール値の論理 AND
論理 XOR	$x ^ y$	整数のビットごとの XOR、ブール型の論理 XOR。
論理 OR	$x   y$	整数のビット単位の OR、ブール値の論理 OR
条件 AND	$x \&& y$	$x$ が <code>true</code> である場合のみ、 $y$ を評価
条件 OR	$x    y$	$x$ が <code>false</code> である場合のみ、 $y$ を評価します。
Null 合体	$x ?? y$	$x$ が <code>null</code> の場合は $y$ と評価し、それ以外の場合は $x$ と評価
条件	$x ? y : z$	$x$ が <code>true</code> の場合は $y$ を評価し、 $x$ が <code>false</code> の場合は $z$ を評価
代入または匿名 関数	$x = y$	譲渡
	$x op= y$	複合代入、サポートされる演算子は <code>*= /=%= += -= &lt;&lt;=%= &gt;&gt;=%= &amp;=%= ^=  =</code>
	$(T x) => y$	匿名関数(ラムダ式)

## 1.5 ステートメント

プログラムのアクションは、ステートメントを使用して表します。C# は複数の異なる種類のステートメントをサポートしており、その一部は埋め込みステートメントによって定義されます。

**block** を使うと、1つのステートメントが許容されるコンテキストに複数のステートメントを記述できます。1 ブロックは区切り記号 { と } の間に記述されたステートメントのリストから構成されます。

**宣言ステートメント**は、ローカルの変数と定数を宣言するために使用します。

**式ステートメント**は、式を評価するために使用します。ステートメントとして使用できる式には、メソッドの呼び出し、**new** 演算子を使用するオブジェクト割り当て、= と複合代入演算子を使用する代入、および ++ と -- 演算子を使用するインクリメント演算とデクリメント演算があります。

**選択ステートメント**は、式の値に基づいて、複数の実行可能なステートメントから、1つのステートメントを選択するために使用します。このグループには **if** ステートメントと **switch** ステートメントが含まれます。

**繰り返しステートメント**は、埋め込みステートメントを反復的に実行するために使用します。このグループには **while**、**do**、**for**、および **foreach** の各ステートメントが含まれます。

**ジャンプステートメント**は、制御を転送するために使用します。このグループには **break**、**continue**、**goto**、**throw**、**return** および **yield** の各ステートメントが含まれます。

**try...catch** ステートメントは、ブロックの実行中に発生する例外をキャッチするために使用し、**try...finally** ステートメントは、例外が発生するかどうかにかかわらず常に実行される終了コードを指定するために使用します。

**checked** ステートメントと **unchecked** ステートメントは、整数型の算術演算および変換に対する "オーバーフロー チェック コンテキスト" を制御するために使用します。

**lock** ステートメントは、指定のオブジェクトに対する相互排他ロックを取得し、ステートメントを実行してから、ロックを解放するために使用します。

**using** ステートメントは、リソースを取得し、ステートメントを実行してから、そのリソースを破棄するために使用します。

次の表に C# のステートメントとそれぞれの例を示します。

ステートメント	例
ローカル変数宣言	<pre>static void Main() {     int a;     int b = 2, c = 3;     a = 1;     Console.WriteLine(a + b + c); }</pre>
ローカル定数宣言	<pre>static void Main() {     const float pi = 3.1415927f;     const int r = 25;     Console.WriteLine(pi * r * r); }</pre>
式ステートメント	<pre>static void Main() {     int i;     i = 123;           // Expression statement     Console.WriteLine(i); // Expression statement     i++;              // Expression statement     Console.WriteLine(i); // Expression statement }</pre>
if ステートメント	<pre>static void Main(string[] args) {     if (args.Length == 0) {         Console.WriteLine("No arguments");     }     else {         Console.WriteLine("One or more arguments");     } }</pre>

switch フラグメント	<pre>static void Main(string[] args) {     int n = args.Length;     switch (n) {         case 0:             Console.WriteLine("No arguments");             break;         case 1:             Console.WriteLine("One argument");             break;         default:             Console.WriteLine("{0} arguments", n);             break;     } }</pre>
while フラグメント	<pre>static void Main(string[] args) {     int i = 0;     while (i &lt; args.Length) {         Console.WriteLine(args[i]);         i++;     } }</pre>
do フラグメント	<pre>static void Main() {     string s;     do {         s = Console.ReadLine();         if (s != null) Console.WriteLine(s);     } while (s != null); }</pre>
for フラグメント	<pre>static void Main(string[] args) {     for (int i = 0; i &lt; args.Length; i++) {         Console.WriteLine(args[i]);     } }</pre>
foreach フラグメント	<pre>static void Main(string[] args) {     foreach (string s in args) {         Console.WriteLine(s);     } }</pre>
break フラグメント	<pre>static void Main() {     while (true) {         string s = Console.ReadLine();         if (s == null) break;         Console.WriteLine(s);     } }</pre>
continue フラグメント	<pre>static void Main(string[] args) {     for (int i = 0; i &lt; args.Length; i++) {         if (args[i].StartsWith("/")) continue;         Console.WriteLine(args[i]);     } }</pre>

goto ステートメント	<pre>static void Main(string[] args) {     int i = 0;     goto check; loop:     Console.WriteLine(args[i++]); check:     if (i &lt; args.Length) goto loop; }</pre>
return ステートメント	<pre>static int Add(int a, int b) {     return a + b; }  static void Main() {     Console.WriteLine(Add(1, 2));     return; }</pre>
yield ステートメント	<pre>static IEnumerable&lt;int&gt; Range(int from, int to) {     for (int i = from; i &lt; to; i++) {         yield return i;     }     yield break; }  static void Main() {     foreach (int x in Range(-10,10)) {         Console.WriteLine(x);     } }</pre>
throw ステートメントと try ブロック	<pre>static double Divide(double x, double y) {     if (y == 0) throw new DivideByZeroException();     return x / y; }  static void Main(string[] args) {     try {         if (args.Length != 2) {             throw new Exception("Two numbers required");         }         double x = double.Parse(args[0]);         double y = double.Parse(args[1]);         Console.WriteLine(Divide(x, y));     }     catch (Exception e) {         Console.WriteLine(e.Message);     }     finally {         Console.WriteLine("Good bye!");     } }</pre>
checked ブロックと unchecked ブロック	<pre>static void Main() {     int i = int.MaxValue;     checked {         Console.WriteLine(i + 1);      // Exception     }     unchecked {         Console.WriteLine(i + 1);      // overflow     } }</pre>

lock ステートメント	<pre>class Account {     decimal balance;     public void Withdraw(decimal amount) {         lock (this) {             if (amount &gt; balance) {                 throw new Exception("Insufficient funds");             }             balance -= amount;         }     } }</pre>
using ステートメント	<pre>static void Main() {     using (TextWriter w = File.CreateText("test.txt")) {         w.WriteLine("Line one");         w.WriteLine("Line two");         w.WriteLine("Line three");     } }</pre>

## 1.6 クラスとオブジェクト

クラスはC#の最も基本的な型です。クラスは、1つの単位内の状態(フィールド)とアクション(メソッドとその他の関数メンバー)を結合するデータ構造です。クラスは、クラスから動的に作成されるインスタンスの定義を提供します。インスタンスはオブジェクトとも呼びます。クラスは、派生クラスが基底クラスを拡張および限定するための機構である、継承とポリモーフィズムをサポートします。

新しいクラスは、クラス宣言を使用して作成します。クラス宣言はヘッダーから開始します。ヘッダーでは、クラスの属性と修飾子、クラスの名称、基底クラス(存在する場合)、およびクラスで実装されるインターフェイスを指定します。ヘッダーに続いてクラス本体を宣言します。クラス本体は、区切り記号{と}の間に記述されたメンバー宣言のリストから構成されます。

次に示すのは、Pointという名前の簡単なクラスの宣言です。

```
public class Point
{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

クラスのインスタンスはnew演算子を使用して作成します。この演算子は新しいインスタンスのメモリを割り当て、インスタンスを初期化するためのコンストラクターを呼び出し、インスタンスの参照を返します。次のステートメントは、2つのPointオブジェクトを作成し、2つの変数にオブジェクトの参照を格納します。

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

オブジェクトによって占められるメモリは、オブジェクトが使用されなくなると自動的に解放されます。C#ではオブジェクトを明示的に解放する必要はありません。また、オブジェクトを明示的に解放することはできません。

### 1.6.1 メンバー

クラスのメンバーは、**静的メンバー**か**インスタンス メンバー**のどちらかになります。静的メンバーはクラスに属し、インスタンスマンバーは（クラスのインスタンスである）オブジェクトに属します。

次の表に、クラスに含まれる各種のメンバーの概要を説明します。

メンバー	説明
定数	クラスに関連付けられた定数值
フィールド	クラスの変数
メソッド	クラスで実行できる演算とアクション
プロパティ	クラスの名前付きプロパティの読み取りと書き込みに関連付けられるアクション
インデクサー	配列などクラスのインスタンスのインデックス付けに関するアクション
イベント	クラスによって生成される通知
演算子	クラスでサポートされる変換演算子と式演算子
コンストラクター	クラスのインスタンスまたはクラス自体の初期化に必要なアクション
デストラクター	クラスのインスタンスが完全に破棄される前に実行されるアクション
型	クラスにより宣言される、入れ子にされた型

### 1.6.2 アクセシビリティ

クラスの各メンバーにはアクセシビリティが関連付けられており、これにより、メンバーにアクセスできるプログラム テキストの範囲が制御されます。5種類のアクセシビリティがあります。次の表は、アクセシビリティをまとめたものです。

アクセシビリティ	説明
<code>public</code>	アクセスは制限されません。
<code>protected</code>	アクセスはそのクラスおよびそのクラスから派生したクラスに制限されます。
<code>internal</code>	アクセスはそのプログラムに制限されます。
<code>protected internal</code>	アクセスはそのプログラムまたはそのクラスから派生したクラスに制限されます。
<code>private</code>	アクセスはそのクラスに制限されます。

### 1.6.3 型パラメーター

クラス定義で型パラメーターの集合を指定するには、クラス名の後に山かっこで囲んだ型パラメータ名のリストを付けます。型パラメーターは、クラスのメンバーを定義するためにクラス宣言の本体で使用できます。次の例では、`Pair` の型パラメーターは `TFirst` と `TSecond` です。

```
public class Pair<TFirst,TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

型パラメーターを取るように宣言されたクラス型はジェネリック クラス型と呼ばれます。構造体、インターフェイス、デリゲートの各型もジェネリックにすることができます。

ジェネリック クラスの使用時には、各型パラメーターに型引数を指定する必要があります。

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two" };
int i = pair.First; // TFirst is int
string s = pair.Second; // TSecond is string
```

上記の `Pair<int,string>` のように、型引数が指定されたジェネリック型は構築された型と呼ばれます。

### 1.6.4 基底クラス

クラス宣言では、クラス名と型パラメーターの後にコロンと基底クラス名を続けることによって、基底クラスを指定できます。基底クラスの指定を省略することは、`object` 型からの派生と同じ意味を持ちます。次の例では、`Point3D` の基底クラスは `Point` であり、`Point` の基底クラスは `object` です。

```
public class Point
{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
public class Point3D: Point
{
    public int z;
    public Point3D(int x, int y, int z): base(x, y) {
        this.z = z;
    }
}
```

クラスはその基底クラスのメンバーを継承します。継承とは、基底クラスのインスタンス コンストラクター、静的コンストラクター、およびデストラクターを除いて、クラスが暗黙的に基底クラスのすべてのメンバーを格納することを意味します。派生クラスは、継承するメンバーに新しいメンバーを追加できますが、継承したメンバーの定義は削除できません。前の例では、`Point3D` は `Point` から `x` フィールドと `y` フィールドを継承し、`Point3D` の各インスタンスには 3 つのフィールド `x`、`y`、`z` が格納されます。

あるクラス型からその基底クラスの任意の型への暗黙的な変換が存在します。したがって、あるクラス型の変数は、そのクラスのインスタンス、または任意の派生クラスの変数を参照できます。たとえば、上記のクラス宣言では、型 `Point` の変数は `Point` か `Point3D` のいずれかを参照できます。

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

### 1.6.5 フィールド

フィールドとは、クラスまたはクラスのインスタンスに関連付けられた変数です。

**static** 修飾子を指定して宣言されたフィールドは、**静的フィールド**を定義します。静的フィールドは正確に1つの格納場所を表します。クラスのインスタンスがいくつ作成されても、静的フィールドのコピーは1つだけです。

**static** 修飾子を使用せずに宣言されたフィールドは、**インスタンスフィールド**を定義します。クラスのどのインスタンスにも、そのクラスのすべてのインスタンスフィールドのコピーが個別に格納されます。

次の例では、**Color** クラスの各インスタンスに **r**、**g**、および **b** インスタンスフィールドの各コピーがありますが、**Black**、**White**、**Red**、**Green** および **Blue** の各静的フィールドのコピーは1つだけです。

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);

    private byte r, g, b;

    public Color(byte r, byte g, byte b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

上記の例に示すように、"読み取り専用のフィールド" は **readonly** 修飾子を使用して宣言できます。**readonly** フィールドへの代入は、フィールドの宣言の一部としてだけ、または同じクラスのコンストラクターの中だけで行うことができます。

### 1.6.6 メソッド

"メソッド" は、オブジェクトまたはクラスによって実行される演算またはアクションを実装するメンバーです。**静的メソッド**には、クラスを通じてアクセスします。**インスタンスマソッド**には、クラスのインスタンスを通じてアクセスします。

メソッドには、メソッドに渡される値または変数参照を表すパラメーターと、メソッドによって計算され返される値の型を指定する戻り値の型から構成されるリスト(空にすることもできる)が含まれます。メソッドが値を返さない場合、その戻り値の型は **void** です。

型と同様に、メソッドも型パラメーターの集合を持つことができます。その場合、メソッドが呼び出されたときに型引数が指定されている必要があります。型とは異なり、型引数はメソッド呼び出しの引数から推測できる場合が多く、明示的に指定する必要はありません。

メソッドのシグネチャは、メソッドが宣言されるクラス内で一意である必要があります。メソッドのシグネチャは、メソッドの名前、型パラメーターの数、そのパラメーターの数、修飾子、型で構成されています。メソッドのシグネチャには、戻り値の型は含まれません。

### 1.6.6.1 パラメーター

パラメーターは、メソッドに値または変数参照を渡すために使用します。メソッドのパラメーターは、メソッドが呼び出されるときに指定される引数から実際の値を取得します。パラメーターには、値パラメーター、参照パラメーター、出力パラメーター、およびパラメーター配列の4種類があります。

**値パラメーター**は、入力パラメーターを渡す場合に使用します。値パラメーターは、パラメーターに渡された引数からその初期値を取得するローカル変数に対応します。値パラメーターの変更は、パラメーターに渡された引数に影響しません。

対応する引数を省略できるように既定値を指定すると、値パラメーターを省略可能にすることができます。

**参照パラメーター**は、入出力パラメーターを渡す場合に使用します。参照パラメーターには引数として変数を渡す必要があり、メソッドの実行中、参照パラメーターは引数の変数と同じ格納場所を示します。参照パラメーターは、`ref` 修飾子を使って宣言します。次の例は `ref` パラメーターの使い方を示しています。

```
using System;
class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("{0} {1}", i, j);           // Outputs "2 1"
    }
}
```

**出力パラメーター**は、出力パラメーターを渡す場合に使用します。出力パラメーターは参照パラメーターと似ていますが、呼び出し側が提供する引数の初期値が重要ではない点で異なります。出力パラメーターは、`out` 修飾子を指定して宣言します。次の例は `out` パラメーターの使い方を示しています。

```
using System;
class Test
{
    static void Divide(int x, int y, out int result, out int remainder) {
        result = x / y;
        remainder = x % y;
    }

    static void Main() {
        int res, rem;
        Divide(10, 3, out res, out rem);
        Console.WriteLine("{0} {1}", res, rem);    // Outputs "3 1"
    }
}
```

**パラメーター配列**を使用すると、可変数の引数をメソッドに渡すことができます。パラメーター配列は、`params` 修飾子を使って宣言します。メソッドの最後のパラメーターのみがパラメーター配列になり、パラメーター配列の型は必ず1次元配列型になります。パラメーター配列のわかりやすい使用例として、`System.Console` クラスの `Write` メソッドと `WriteLine` メソッドがあります。これらのメソッドは次のように宣言されています。

```
public class Console
{
    public static void Write(string fmt, params object[] args) {...}
    public static void WriteLine(string fmt, params object[] args) {...}

    ...
}
```

パラメーター配列を使用するメソッド内で、パラメーター配列は配列型の標準パラメーターとまったく同じように動作します。ただし、パラメーター配列を使用するメソッドの呼び出しでは、パラメーター配列型の1つの引数、またはそのパラメーター配列の要素型の任意の数の引数のどちらかを渡すことができます。後者の場合、指定された引数で配列インスタンスが自動的に作成され、初期化されます。次の例を参照してください。

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

これは次と同じです。

```
string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

### 1.6.6.2 メソッド本体とローカル変数

メソッドの本体は、メソッドを呼び出すと実行されるステートメントを指定します。

メソッドの本体で、メソッドの呼び出しに固有の変数を宣言できます。このような変数は**ローカル変数**と呼ばれます。ローカル変数宣言は、型名、変数名、場合によっては初期値を指定します。次の例では、ローカル変数 *i* を初期値ゼロで、ローカル変数 *j* を初期値なしで宣言しています。

```
using System;
class Squares
{
    static void Main() {
        int i = 0;
        int j;
        while (i < 10) {
            j = i * i;
            Console.WriteLine("{0} x {0} = {1}", i, j);
            i = i + 1;
        }
    }
}
```

C# では、ローカル変数の値を取得するためには、あらかじめ明示的に代入する必要があります。たとえば、上記の例の *i* の宣言に初期値を指定していない場合、その後で *i* が使用されるとコンパイラはエラーを報告します。これはプログラム内の *i* が使用される各時点で確実に代入されないためです。

メソッドは **return** ステートメントを使用して、呼び出し元に制御を返すことができます。**void** を返すメソッドでは、**return** ステートメントで式を指定できません。**void** 以外を返すメソッドでは、戻り値を計算する式を **return** ステートメントに指定する必要があります。

### 1.6.6.3 静的メソッドとインスタンス メソッド

**static** 修飾子を使って宣言されたメソッドは、**静的メソッド**です。静的メソッドは特定のインスタンスで動作せず、静的メンバーにのみ直接アクセスできます。

**static** 修飾子を使用せずに宣言されたメソッドは、**インスタンスメソッド**です。インスタンス メソッドは特定のインスタンスで動作し、静的メンバーとインスタンス メンバーのいずれにもアクセスできます。インスタンス メソッドが呼び出されたインスタンスには、**this** と同様に明示的にアクセスできます。静的メソッドで **this** を参照するとエラーになります。

次の **Entity** クラスには、静的メンバーとインスタンス メンバーの両方が格納されています。

```
class Entity
{
    static int nextSerialNo;
    int serialNo;
    public Entity() {
        serialNo = nextSerialNo++;
    }
    public int GetSerialNo() {
        return serialNo;
    }
    public static int GetNextSerialNo() {
        return nextSerialNo;
    }
    public static void SetNextSerialNo(int value) {
        nextSerialNo = value;
    }
}
```

それぞれの **Entity** インスタンスには、連番(およびここでは示されないその他の情報)が格納されています。**Entity** コンストラクター(インスタンス メソッドに似ている)は、次に使用できる連番によって新しいインスタンスを初期化します。コンストラクターはインスタンス メンバーであるため、**serialNo** インスタンス フィールドと **nextSerialNo** 静的フィールドのいずれにもアクセスが許可されます。

**GetNextSerialNo** と **SetNextSerialNo** の静的メソッドは、**nextSerialNo** 静的フィールドにアクセスできますが、**serialNo** インスタンス フィールドに直接アクセスするとエラーになります。

**Entity** クラスの例を次に示します。

```
using System;
class Test
{
    static void Main() {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();
        Console.WriteLine(e1.GetSerialNo());           // Outputs "1000"
        Console.WriteLine(e2.GetSerialNo());           // Outputs "1001"
        Console.WriteLine(Entity.GetNextSerialNo());   // Outputs "1002"
    }
}
```

**SetNextSerialNo** と **GetNextSerialNo** の静的メソッドはクラスで呼び出されるのに対して、**GetSerialNo** インスタンス メソッドはクラスのインスタンスで呼び出されます。

#### 1.6.6.4 仮想メソッド、オーバーライドメソッド、抽象メソッド

インスタンスメソッドの宣言に `virtual` 修飾子が含まれる場合、そのメソッドは "仮想メソッド" と呼ばれます。`virtual` 修飾子が存在しない場合、そのメソッドは非仮想メソッドと呼ばれます。

仮想メソッドの呼び出しでは、呼び出しが行われるインスタンスの "実行時の型" により、実際に呼び出されるメソッド実装が決定されます。非仮想メソッドの呼び出しでは、インスタンスのコンパイル時の型が決定要因です。

仮想メソッドは派生クラスではオーバーライドできません。インスタンスメソッド宣言に `override` 修飾子が含まれる場合、メソッドは継承された仮想メソッドと同じシグネチャでオーバーライドします。仮想メソッドの宣言は新しいメソッドを "導入" しますが、オーバーライドメソッドの宣言は、そのメソッドの新しい実装を提供して、既存の継承仮想メソッドを "特化" します。

**抽象メソッド**は、実装のない仮想メソッドです。抽象メソッドは `abstract` 修飾子を使用して宣言され、`abstract` が宣言されているクラスでのみ宣言が許可されます。抽象メソッドは、派生した非抽象クラスごとにオーバーライドする必要があります。

次の例では抽象クラス `Expression` を宣言しています。このクラスは式のツリー ノードおよび3つの派生クラス、`Constant`、`VariableReference`、`Operation` を表しています。この3つの派生クラスは、定数、変数参照、および算術演算の式のツリー ノードを実装します(これは、4.6で紹介されている式ツリー型と似ているので、混同しないように気を付けてください)。

```
using System;
using System.Collections;
public abstract class Expression
{
    public abstract double Evaluate(Hashtable vars);
}
public class Constant: Expression
{
    double value;
    public Constant(double value) {
        this.value = value;
    }
    public override double Evaluate(Hashtable vars) {
        return value;
    }
}
public class VariableReference: Expression
{
    string name;
    public VariableReference(string name) {
        this.name = name;
    }
    public override double Evaluate(Hashtable vars) {
        object value = vars[name];
        if (value == null) {
            throw new Exception("Unknown variable: " + name);
        }
        return Convert.ToDouble(value);
    }
}
```

```

public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;

    public Operation(Expression left, char op, Expression right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }

    public override double Evaluate(Hashtable vars) {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Unknown operator");
    }
}

```

上述の 4 つのクラスを使用して、算術式をモデル化できます。たとえば、これらのクラスのインスタンスを使用した場合、式  $x + 3$  は次のように表されます。

```

Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));

```

**Expression** インスタンスの **Evaluate** メソッドを呼び出して、指定された式を評価し、**double** 値を生成します。このメソッドは、変数名(エントリのキー)と値(エントリの値)を格納する **Hashtable** を引数に使用します。**Evaluate** メソッドは、仮想抽象メソッドです。つまり、実際に実装するためには、派生した非抽象クラスでオーバーライドする必要があります。

**Evaluate** で **Constant** を実装すると、格納された定数のみが返されます。**VariableReference** の実装では、ハッシュテーブルの変数名が検索され、結果値が返されます。**Operation** の実装では、まず(**Evaluate** メソッドを再帰的に呼び出して)左右のオペランドが評価され、次に指定された算術演算が実行されます。

次のプログラムでは、**Expression** クラスを使用して異なる値の  $x$  と  $y$  について式  $x * (y + 2)$  を評価しています。

```

using System;
using System.Collections;
class Test
{
    static void Main() {
        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
        );
        Hashtable vars = new Hashtable();

```

```

vars["x"] = 3;
vars["y"] = 5;
Console.WriteLine(e.Evaluate(vars));      // Outputs "21"
vars["x"] = 1.5;
vars["y"] = 9;
Console.WriteLine(e.Evaluate(vars));      // Outputs "16.5"
}
}

```

### 1.6.6.5 メソッドのオーバーロード

メソッドをオーバーロードした場合、同じクラスのメソッドであってもシグネチャが異なれば、複数のメソッドで同じ名前を使用できます。コンパイラは、オーバーロードされたメソッドの呼び出しをコンパイルする場合、[オーバーロードの解決](#)によって個々の呼び出すメソッドを決定します。オーバーロードの解決は、引数に最適なメソッドを1つ見つけます。最適なメソッドが見つからない場合はエラーを報告します。次の例では、実際のオーバーロードの解決を示しています。`Main` メソッドの各呼び出しに対するコメントは、実際に呼び出されるメソッドを示しています。

```

class Test
{
    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object x) {
        Console.WriteLine("F(object)");
    }

    static void F(int x) {
        Console.WriteLine("F(int)");
    }

    static void F(double x) {
        Console.WriteLine("F(double)");
    }

    static void F<T>(T x) {
        Console.WriteLine("F<T>(T)");
    }

    static void F(double x, double y) {
        Console.WriteLine("F(double, double)");
    }

    static void Main() {
        F();                                // Invokes F()
        F(1);                               // Invokes F(int)
        F(1.0);                             // Invokes F(double)
        F("abc");                           // Invokes F(object)
        F((double)1);                      // Invokes F(double)
        F((object)1);                      // Invokes F(object)
        F<int>(1);                         // Invokes F<T>(T)
        F(1, 1);                            // Invokes F(double, double)
    }
}

```

例で示されているように、一致するパラメーターの型に引数を明示的にキャストするか、型引数を明示的に指定することによって、常に特定のメソッドを選択できます。

### 1.6.7 その他の関数メンバー

実行可能なコードを含むメンバーは、クラスの[関数メンバー](#)と総称されます。前のセクションでは、関数メンバーの基本的な種類であるメソッドを説明しました。このセクションでは、C# でサポート

されるその他の種類の関数メンバーであるコンストラクター、プロパティ、インデクサー、イベント、演算子、およびデストラクターについて説明します。

次の表に、拡張可能なオブジェクトのリストを実装するジェネリック クラス `List<T>` を示します。このクラスには、最も一般的な種類の関数メンバーの例がいくつか含まれています。

<code>public class List&lt;T&gt;</code>	
<code>{</code>	
<code>    const int defaultCapacity = 4;</code>	定数
<code>    T[] items;</code>	フィールド
<code>    int count;</code>	
<code>    public List(int capacity = defaultCapacity) {</code>	コンストラクタ
<code>        items = new T[capacity];</code>	
<code>    }</code>	
<code>    public int Count {</code>	プロパティ
<code>        get { return count; }</code>	
<code>    }</code>	
<code>    public int Capacity {</code>	
<code>        get {</code>	
<code>            return items.Length;</code>	
<code>        }</code>	
<code>        set {</code>	
<code>            if (value &lt; count) value = count;</code>	
<code>            if (value != items.Length) {</code>	
<code>                T[] newItems = new T[value];</code>	
<code>                Array.Copy(items, 0, newItems, 0, count);</code>	
<code>                items = newItems;</code>	
<code>            }</code>	
<code>        }</code>	
<code>}</code>	

<pre> public T this[int index] {     get {         return items[index];     }     set {         items[index] = value;         OnChanged();     } } </pre>	インデクサー
<pre> public void Add(T item) {     if (count == Capacity) Capacity = count * 2;     items[count] = item;     count++;     OnChanged(); } protected virtual void OnChanged() {     if (Changed != null) Changed(this, EventArgs.Empty); } public override bool Equals(object other) {     return Equals(this, other as List&lt;T&gt;); } static bool Equals(List&lt;T&gt; a, List&lt;T&gt; b) {     if (a == null) return b == null;     if (b == null    a.Count != b.Count) return false;     for (int i = 0; i &lt; a.Count; i++) {         if (!object.Equals(a.Items[i], b.Items[i])) {             return false;         }     }     return true; } </pre>	メソッド
<pre> public event EventHandler Changed; </pre>	イベント
<pre> public static bool operator ==(List&lt;T&gt; a, List&lt;T&gt; b) {     return Equals(a, b); } public static bool operator !=(List&lt;T&gt; a, List&lt;T&gt; b) {     return !Equals(a, b); } </pre>	演算子

### 1.6.7.1 コンストラクター

C#はインスタンスコンストラクターと静的コンストラクターの両方をサポートします。“**インスタンスコンストラクター**”は、クラスのインスタンスを初期化するために必要なアクションを実装するメンバーです。**静的コンストラクター**は、クラスが最初に読み込まれたときに、それ自体を初期化するために必要なアクションを実装するメンバーです。

コンストラクターは、メソッドと同様、戻り値の型を指定せず、包含するクラスと同じ名前で宣言されます。**static**修飾子を含むコントラクタ宣言は、静的コンストラクターを宣言します。それ以外の場合はインスタンスコンストラクターを宣言します。

インスタンスコンストラクターはオーバーロードできます。たとえば、**List<T>**クラスは2つのインスタンスコンストラクターを宣言しますが、一方はパラメーターがなく、もう一方は**int**パラメーターを使用します。インスタンスコンストラクターは、**new**演算子を使用して呼び出します。次

のステートメントは、`List` クラスの各コンストラクターを使用して、2 つの `List<string>` インスタンスを割り当てています。

```
List<string> list1 = new List<string>();  
List<string> list2 = new List<string>(10);
```

他のメンバーと異なり、インスタンス コンストラクターは継承されません。クラスには、クラスで実際に宣言されたものを除いて、インスタンス コンストラクターはありません。クラスに対してインスタンス コンストラクターを定義しない場合は、パラメーターのない空のインスタンス コンストラクターが自動的に生成されます。

### 1.6.7.2 プロパティ

プロパティは、フィールドを延長したものと考えることができます。どちらも関連する型を持つ名前付きのメンバーで、アクセスするための構文も同じです。ただし、フィールドとは異なり、プロパティには記憶場所の指定はありません。代わりに、プロパティには、値を読み取ったり書き込んだりするときに実行されるステートメントを指定する "アクセサー" があります。

プロパティはフィールドと同様に宣言しますが、宣言をセミコロンで終了せずに、区切り記号 { と } の間に記述された `get` アクセサーと `set` アクセサーまたはそのいずれかで終了する点が異なります。`get` と `set` の両方のアクセサーで終了するプロパティは "読み取り/書き込み" プロパティ、`get` アクセサーのみで終了するプロパティは "読み取り専用" プロパティ、`set` アクセサーのみで終了するプロパティは "書き込み専用" プロパティと呼ばれます。

`get` アクセサーは、プロパティ型の戻り値を持つパラメーターなしのメソッドに対応します。プロパティが代入のターゲットとしてではなく、式で参照される場合は、プロパティの `get` アクセサーが呼び出され、プロパティの値が計算されます。

`set` アクセサーは、`value` という名前のパラメーターを 1 つ使用し、戻り値の型のないメソッドに対応します。プロパティが代入のターゲットとして、または ++ または -- のオペランドとして参照される場合は、新しい値を提供する引数を使用して `set` アクセサーが呼び出されます。

`List<T>` クラスは、読み取り専用の `Count` プロパティと読み取り/書き込み可能な `Capacity` プロパティを宣言します。次に、この 2 つのプロパティの使用例を示します。

```
List<string> names = new List<string>();  
names.Capacity = 100;           // Invokes set accessor  
int i = names.Count;          // Invokes get accessor  
int j = names.Capacity;        // Invokes get accessor
```

フィールドおよびメソッドの場合と同様に、C# はインスタンス プロパティと静的 プロパティの両方をサポートします。静的 プロパティは `static` 修飾子を使用して宣言し、インスタンス プロパティはこの修飾子を使用せずに宣言します。

プロパティのアクセサーは仮想にすることができます。プロパティ宣言に `virtual`、`abstract`、`override` のいずれかの修飾子が含まれている場合、この宣言はプロパティのアクセサーに適用されます。

### 1.6.7.3 インデクサー

インデクサーは、配列と同じ方法でオブジェクトにインデックスを作成できるようにするためのメンバーです。インデクサーはプロパティと同じ方法で宣言しますが、メンバーの名前が `this` で、この後に区切り記号 [ と ] の間に記述したパラメーター リストがある点が異なります。パラメーターはインデクサーのアクセサーで使用できます。プロパティと同様に、インデクサーは読み取り/書き込

み、読み取り専用、書き込み専用のすべての種類が可能で、インデクサーのアクセサーは仮想にすることができます。

`List` クラスは、`int` パラメーターを使用する单一の読み取り/書き込みインデクサーを宣言します。インデクサーにより、`int` 値を使用して `List` インスタンスのインデックスを作成できます。次に例を示します。

```
List<string> names = new List<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++) {
    string s = names[i];
    names[i] = s.ToUpper();
}
```

インデクサーはオーバーロードできます。したがって、パラメーターの数または型が異なる限り、1 つのクラスで複数のインデクサーを宣言できます。

#### 1.6.7.4 イベント

イベントは、クラスやオブジェクトが通知を発行できるようにするためのメンバーです。宣言に `event` キーワードが含まれている点と、型をデリゲート型にする必要があることを除いて、イベントはフィールドと同じように宣言します。

イベントメンバーを宣言するクラス内では、イベントが抽象イベントではなく、アクセサーを宣言しない場合、イベントはデリゲート型のフィールドとまったく同じように動作します。フィールドは、イベントに追加されたイベントハンドラーを表すデリゲートへの参照を格納します。イベントハンドラーが存在しない場合、フィールドは `null` です。

`List<T>` クラスでは、唯一のイベントメンバー `Changed` を宣言します。このイベントメンバーは、新しい項目がリストに追加されたことを示します。`Changed` イベントは、イベントが `null` (ハンドラーが格納されていない) かどうかを最初にチェックする `OnChanged` 仮想メソッドによって発生します。イベント発生の表記は、イベントによって表されるデリゲートの呼び出しとまったく同じです。このため、イベント発生用の特別な言語構成要素はありません。

クライアントは、イベントハンドラーを通じてイベントに対応します。イベントハンドラーは `+=` 演算子を使用して追加し、`-=` 演算子を使用して削除します。次の例では、イベントハンドラーを `List<string>` の `Changed` イベントに追加しています。

```
using System;
class Test
{
    static int changeCount;
    static void ListChanged(object sender, EventArgs e) {
        changeCount++;
    }
    static void Main() {
        List<string> names = new List<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount);      // Outputs "3"
    }
}
```

イベントの基になるストレージを制御する必要がある高度なシナリオでは、プロパティの `set` アクセサーに似た `add` アクセサーと `remove` アクセサーをイベント宣言で明示的に指定できます。

#### 1.6.7.5 演算子

**演算子**は、特定の式の演算子をクラスのインスタンスに適用する意味を定義するメンバーです。単項演算子、二項演算子、および変換演算子という 3 種類の演算子を定義できます。すべての演算子は、`public` および `static` として宣言する必要があります。

`List<T>` クラスは、2 つの演算子 `operator ==` と `operator !=` を宣言することで、これらの演算子を `List` インスタンスに適用する式に新しい意味を与えます。具体的には、これらの演算子は `List<T>` の 2 つのインスタンスが同等であるかどうかを、`Equals` メソッドを使用してインスタンス内の各オブジェクトを比較することとして定義します。次の例では、`==` 演算子を使用して 2 つの `List<int>` インスタンスを比較しています。

```
using System;
class Test
{
    static void Main() {
        List<int> a = new List<int>();
        a.Add(1);
        a.Add(2);
        List<int> b = new List<int>();
        b.Add(1);
        b.Add(2);
        Console.WriteLine(a == b);      // Outputs "True"
        b.Add(3);
        Console.WriteLine(a == b);      // Outputs "False"
    }
}
```

最初の `Console.WriteLine` は、2 つのリストに同じ値を持つ同じ数のオブジェクトが同じ順序で格納されているので `True` を出力します。`List<T>` で `operator ==` が定義されていない場合、最初の `Console.WriteLine` は、`a` と `b` が異なる `List<int>` インスタンスを参照するため `False` を出力します。

#### 1.6.7.6 デストラクター

"デストラクター" は、クラスのインスタンスを消滅させるために必要なアクションを実装するメンバーです。デストラクターは、パラメーターを持つことも、アクセシビリティ修飾子を指定することもできません。また、明示的に呼び出すこともできません。インスタンスに対するデストラクターは、ガーベージコレクションの間に自動的に呼び出されます。

ガーベージコレクターでは、オブジェクトのコレクション時期やデストラクターの実行時期を、かなり自由に決定できます。特に、デストラクターの呼び出しのタイミングは確定的ではなく、デストラクターは任意のスレッドで実行できます。このことを含むいくつかの理由から、他に適切なソリューションが見つからない場合にのみ、クラスにデストラクターを実装してください。

`using` ステートメントを使用すると、より適切にオブジェクトを破壊できます。

### 1.7 構造体

クラスと同様に、**構造体**はデータ メンバーと関数メンバーを格納できるデータ構造体ですが、クラスと異なり、構造体は値型でありヒープ割り当てを必要としません。構造体型の変数は構造体のデータを直接格納します。クラス型の変数は動的に割り当てられたオブジェクトへの参照を格納します。

構造体型はユーザー指定の継承をサポートしません。すべての構造体型は暗黙的に `object` 型を継承します。

構造体は、値セマンティクスを持つ小規模なデータ構造に特に役立ちます。構造体に適した例としては、複雑な数値、座標システム内の点、またはディクショナリ内のキー値のペアがあります。小さなデータ構造にはクラスではなく構造体を使用することで、アプリケーションで実行されるメモリ割り当ての回数を大幅に減らすことができます。たとえば、次のプログラムでは、100 ポイントの配列を作成し、初期化しています。クラスとして実装された `Point` では、101 個の独立したオブジェクト(配列に対して 1 つと、100 個の要素に 1 つずつ)がインスタンス化されます。

```
class Point
{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
class Test
{
    static void Main() {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++) points[i] = new Point(i, i);
    }
}
```

または、`Point` を構造体にします。

```
struct Point
{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

これで 1 つのオブジェクト、つまり配列のオブジェクトのみがインスタンス化されました。`Point` のインスタンスは配列にインラインで格納されます。

構造体コンストラクターは `new` 演算子を使用して呼び出しますが、この呼び出しによりメモリが割り当てられるわけではありません。オブジェクトを動的に割り当て、そのオブジェクトの参照を返す代わりに、構造体インストラクタは構造体の値そのもの(通常はスタックの一時的な場所にある)を返します。この値はその後、必要に応じてコピーされます。

クラスでは、2 つの変数が同じオブジェクトを参照できるため、一方の変数に対する演算が他方の変数によって参照されるオブジェクトに影響を与えることがあります。構造体の場合、各変数はデータの独自のコピーを保持しているため、ある変数を操作することによって他の変数にも影響することはありません。たとえば、次のコードから生成される出力は、`Point` がクラスか構造体のどちらであるかによって異なります。

```
Point a = new Point(10, 10);
Point b = a;
a.x = 20;
Console.WriteLine(b.x);
```

`Point` がクラスであれば、`a` と `b` は同じオブジェクトを参照するため、`20` が出力されます。`Point` が構造体の場合、`a` を `b` に代入することで値のコピーが作成され、このコピーは、その後の `a.x` の影響を受けないため、`10` が出力されます。

上記の例は、構造体の制限のうちの 2 つを示しています。まず、通常は構造体全体のコピーはオブジェクト参照のコピーよりも効率が低いため、参照型よりも構造体を使用する方が、代入と値パラメーターの受け渡しの負荷が大きくなります。第 2 に、`ref` パラメーターと `out` パラメーターを除き、構造体の参照を作成することは不可能であるため、状況によっては構造体を使用できません。

## 1.8 配列

配列とは、算出されたインデックスを使用してアクセスされる複数の変数を含むデータ構造です。配列に含まれる変数は、配列の要素とも呼ばれ、すべて同じ型です。この型を配列の要素型と呼びます。

配列型は参照型なので、配列変数を宣言すると確保されるのは、配列のインスタンスに対する参照のための領域だけです。実際の配列のインスタンスは、`new` 演算子を使用して実行時に動的に作成されます。`new` 演算は新しい配列インスタンスの長さを指定します。この長さはインスタンスの有効期間に合わせて固定されます。配列の要素のインデックスの範囲は、`0` から `Length - 1` までです。`new` 演算子は配列の要素を自動的に既定値に初期化します。既定値は、たとえばすべての数値型ではゼロ、すべての参照型では `null` です。

次の例では、`int` 要素の配列を作成し、この配列を初期化し、配列の内容を出力します。

```
using System;
class Test
{
    static void Main()
    {
        int[] a = new int[10];
        for (int i = 0; i < a.Length; i++) {
            a[i] = i * i;
        }
        for (int i = 0; i < a.Length; i++) {
            Console.WriteLine("a[{0}] = {1}", i, a[i]);
        }
    }
}
```

この例は 1 次元配列で作成および演算が行われています。C# は複数次元配列もサポートしています。配列型の次元数は、配列型のランクとも呼ばれ、配列型の角かっこ内に記述されたコンマの数に 1 を加えた数です。次の例では、1 次元、2 次元、および 3 次元配列を割り当てています。

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

`a1` 配列には 10 の要素が、`a2` 配列には  $10 \times 5$  の要素が、`a3` 配列には  $10 \times 5 \times 2$  の要素が格納されています。

配列の要素型は、配列型を含めて、任意の型に設定できます。配列型の要素から成る配列は、要素の配列の長さが等しくなる必要がないため、ジャグ配列と呼ばれる場合もあります。次の例では、`int` の配列から構成される配列を割り当てています。

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

1行目では3つの要素から成る配列が作成され、各要素の型は `int[]`、初期値はそれぞれ `null` とし  
ています。後の行では、長さの異なる個々の配列インスタンスを参照して3つの要素を初期化してい  
ます。

`new` 演算子を使用した場合、区切り記号 `{ と }` の内側に記述された式のリストである "配列初期化子"  
を使用して、配列要素の初期値を指定できます。次の例では、3つの要素を使用して `int[]` を割り  
当て、初期化しています。

```
int[] a = new int[] {1, 2, 3};
```

配列の長さは、`{ と }` の間の式の数から導かれます。ローカル変数とフィールドの宣言をさらに短縮  
して、配列型を再度宣言する必要がない形式にできます。

```
int[] a = {1, 2, 3};
```

上記の例は、いずれも次の式と同等です。

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

## 1.9 インターフェイス

インターフェイスは、クラスと構造体で実装できるコントラクトを定義します。インターフェイスに  
は、メソッド、プロパティ、イベント、およびインデクサーを含めることができます。インターフェ  
イスは、そのインターフェイスで定義されるメンバーを実装しません。そのインターフェイスを実装  
するクラスまたは構造体が提供する必要のあるメンバーを指定するだけです。

インターフェイスでは、**多重継承**を使用できます。次の例では、インターフェイス `IComboBox` は  
`ITextBox` と `IListBox` の両方を継承しています。

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

クラスおよび構造体は、複数のインターフェイスを実装できます。次の例では、クラス `EditBox` は  
`IControl` と `IDataBound` の両方を実装しています。

```
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox: IControl, IDataBound
{
    public void Paint() {...}
```

```
    public void Bind(Binder b) {...}  
}
```

クラスまたは構造体が特定のインターフェイスを実装する場合、そのクラスまたは構造体のインスタンスは、実装されたインターフェイスの型に暗黙的に変換できます。次に例を示します。

```
EditBox editBox = new EditBox();  
IControl control = editBox;  
IDataBound dataBound = editBox;
```

インスタンスが静的に認識されず、特定のインターフェイスを実装できない場合、動的な型のキャストを使用できます。たとえば、次のステートメントは動的な型のキャストを使用して、オブジェクトの **IControl** および **IDataBound** インターフェイス実装を取得しています。オブジェクトの実際の型は **EditBox** であるため、キャストは成功します。

```
object obj = new EditBox();  
IControl control = (IControl)obj;  
IDataBound dataBound = (IDataBound)obj;
```

前の **EditBox** クラスでは、**IControl** インターフェイスの **Paint** メソッドと **IDataBound** インターフェイスの **Bind** メソッドは、**public** メンバーを使用して実装されています。また C# では "インターフェイスメンバーの明示的実装" をサポートしており、クラスまたは構造体でメンバーが **public** になるのを避けるために使用できます。インターフェイスメンバーの明示的実装は、完全修飾インターフェイス メンバー名を使用して記述します。たとえば、次のように **EditBox** クラスはインターフェイス メンバーの明示的実装を使用して、**IControl.Paint** メソッドと **IDataBound.Bind** メソッドを実装することができます。

```
public class EditBox: IControl, IDataBound  
{  
    void IControl.Paint() {...}  
    void IDataBound.Bind(Binder b) {...}  
}
```

明示的インターフェイス メンバーには、インターフェイス型を使用してのみアクセスできます。たとえば、前の **EditBox** クラスが提供する **IControl.Paint** の実装を呼び出すには、まず **EditBox** 参照を **IControl** インターフェイス型に変換する必要があります。

```
EditBox editBox = new EditBox();  
editBox.Paint(); // Error, no such method  
IControl control = editBox;  
control.Paint(); // ok
```

## 1.10 列挙型

**列挙型**は、一連の名前付き定数を持つ特別な型です。次の例では、3 つの定数値 **Red**、**Green**、**Blue**を持つ列挙型 **Color** を宣言して使用しています。

```
using System;  
enum Color  
{  
    Red,  
    Green,  
    Blue  
}
```

```

class Test
{
    static void PrintColor(Color color) {
        switch (color) {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Unknown color");
                break;
        }
    }

    static void Main() {
        Color c = Color.Red;
        PrintColor(c);
        PrintColor(Color.Blue);
    }
}

```

各列挙型には、列挙型の**基になる型**と呼ばれる、対応する整数型があります。列挙型の宣言で基になる型を明示的に宣言しない場合は、基になる型が `int` になります。列挙型のストレージ形式と使用できる値の範囲は、基になる型によって決まります。列挙型が使用できる値のセットが、その列挙型のメンバーによって制限されることはありません。特に、列挙型の基になる型の値は、その列挙型にキャストでき、その列挙型の固有の有効値となります。

次の例は、基になる型が `sbyte` の列挙型 `Alignment` を宣言しています。

```

enum Alignment: sbyte
{
    Left = -1,
    Center = 0,
    Right = 1
}

```

この例から示されるように、列挙型のメンバー宣言にメンバーの値を指定する定数式を含めることができます。列挙型の各メンバーの定数値は、その列挙型の基になる型の範囲内にある必要があります。列挙型のメンバーの宣言で値を明示的に指定しない場合、値には値ゼロ(列挙型の最初のメンバーの場合)、または直前の列挙型メンバーの値に 1 を加えた値が割り当てられます。

型キャストを使用して、列挙値を整数値に、または整数値を列挙値に変換できます。次に例を示します。

```

int i = (int)Color.Blue;      // int i = 2;
Color c = (Color)2;          // Color c = Color.Blue;

```

列挙型の既定値は、列挙型に変換された整数値ゼロです。変数が自動的に既定値に初期化される場合は、この既定値が列挙型の変数に割り当てられます。列挙型の既定値を簡単に利用できるように、リテラル 0 は暗黙的に任意の列挙型に変換されます。この場合、次の式を使用できます。

```
Color c = 0;
```

## 1.11 デリゲート

デリゲート型は、特定のパラメーターリストおよび戻り値の型を使用して、メソッドへの参照を表します。デリゲートにより、メソッドを変数に代入し、パラメーターとして渡すことのできるエンティティとして取り扱うことができます。デリゲートは他の一部の言語に見られる関数ポインターの概念に似ていますが、関数ポインターとは異なり、デリゲートはオブジェクト指向でタイプセーフです。

次の例では、デリゲート型 `Function` を宣言し使用しています。

```
using System;
delegate double Function(double x);
class Multiplier
{
    double factor;
    public Multiplier(double factor) {
        this.factor = factor;
    }
    public double Multiply(double x) {
        return x * factor;
    }
}
class Test
{
    static double Square(double x) {
        return x * x;
    }
    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }
    static void Main() {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, Square);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles = Apply(a, m.Multiply);
    }
}
```

`Function` デリゲート型のインスタンスは、引数 `double` を使用し、`double` 値を返す任意のメソッドを参照できます。`Apply` メソッドは、指定された `Function` を `double[]` の要素に適用し、結果を含む `double[]` を返します。`Main` メソッドでは、`Apply` を使用して 3 つの異なる関数を `double[]` に適用します。

デリゲートは、静的メソッド(前の例の `Square`、`Math.Sin` など)またはインスタンスメソッド(前の例の `m.Multiply` など)を参照できます。インスタンスマソッドを参照するデリゲートは、特定のオブジェクトも参照し、インスタンスマソッドがデリゲートを通じて呼び出されると、そのオブジェクトは呼び出しの中で `this` になります。

デリゲートも匿名関数を使用して作成できます。匿名関数とは、その場で作成される "インラインメソッド" です。匿名関数は、外側のメソッドのローカル変数を参照できます。したがって、上記の乗算の例は、`Multiplier` クラスを使用せずに、より簡単に記述できます。

```
double[] doubles = Apply(a, (double x) => x * 2.0);
```

デリゲートの興味深く有用な性質として、参照するメソッドのクラスについて関知も関与もしないということがあります。考慮されるのは、参照されるメソッドと、デリゲートのパラメーターおよび戻り値の型が同じであるということだけです。

## 1.12 属性

C# プログラムの型、メンバー、その他のエンティティは、それぞれの動作の特定の側面を制御する修飾子をサポートします。たとえば、メソッドのアクセシビリティは `public`、`protected`、`internal` および `private` の各修飾子を使用して制御されます。C# ではこの機能が汎用化されており、ユーザー定義の宣言情報の型をプログラムのエンティティに追加し、実行時に取得できます。プログラムでは、**属性**を定義して使用することにより、この追加の宣言情報を指定します。

次の例では、プログラムのエンティティに配置して関連するドキュメントへのリンクを設定できる `HelpAttribute` 属性を宣言しています。

```
using System;
public class HelpAttribute: Attribute
{
    string url;
    string topic;
    public HelpAttribute(string url) {
        this.url = url;
    }
    public string Url {
        get { return url; }
    }
    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}
```

すべての属性クラスは、.NET Framework で提供される `System.Attribute` 基底クラスから派生しています。属性は、その名前と引数を角かっこで囲んで指定することによって適用できます。これは、関連する宣言の直前に置きます。属性の名前が `Attribute` で終わる場合、属性を参照するときに名前のその部分を省略できます。たとえば、`HelpAttribute` 属性は次のように使用できます。

```
[Help("http://msdn.microsoft.com/.../MyClass.htm")]
public class Widget
{
    [Help("http://msdn.microsoft.com/.../MyClass.htm", Topic = "Display")]
    public void Display(string text) {}
}
```

この例では `HelpAttribute` を `Widget` クラスに追加し、もう 1 つの `HelpAttribute` をこのクラスの `Display` メソッドに追加しています。属性クラスのパブリック コンストラクターは、属性をプログラムエンティティに追加するときに提供する必要のある情報を制御します。属性クラスのパブリック読み取り/書き込み可能なプロパティを参照することで、上記の `Topic` プロパティへの参照などの情報を追加できます。

次の例では、指定したプログラム エンティティの属性情報を、リフレクションを使用して実行時に取得する方法を示しています。

```
using System;
using System.Reflection;
class Test
{
    static void ShowHelp(MemberInfo member) {
        HelpAttribute a = Attribute.GetCustomAttribute(member,
            typeof(HelpAttribute)) as HelpAttribute;
        if (a == null) {
            Console.WriteLine("No help for {0}", member);
        }
        else {
            Console.WriteLine("Help for {0}:", member);
            Console.WriteLine(" Url={0}, Topic={1}", a.Url, a.Topic);
        }
    }
    static void Main() {
        ShowHelp(typeof(widget));
        ShowHelp(typeof(widget).GetMethod("Display"));
    }
}
```

リフレクションを通じて特定の属性を要求した場合は、プログラム ソースで提供される情報によって属性クラスのコンストラクターが呼び出され、最終的な属性インスタンスが返されます。プロパティを通じて情報が追加された場合は、それらのプロパティが指定の値に設定されてから、属性インスタンスが返されます。



## 2. 構文の構造

### 2.1 プログラム

C# の "プログラム" は、正式には "コンパイル単位" と呼ばれる 1 つ以上の "ソースファイル" で構成されています (9.1 を参照)。ソースファイルは、規則正しく並んだ一連の Unicode 文字です。ソースファイルは、普通、ファイルシステムのファイルと一対一に対応していますが、この対応は必須ではありません。最大限の移植性を得るには、ファイルシステムのファイルを UTF-8 エンコーディングでエンコードすることをお勧めします。

概念的には、プログラムのコンパイルは 3 つの手順で行われます。

1. 変換。特定の文字範囲とエンコーディングスキームを使用したファイルを、一連の Unicode 文字を使用したファイルに変換します。
2. 字句解析。Unicode 入力文字のストリームをトークンのストリームに変換します。
3. 構文解析。トークンのストリームを実行可能なコードに変換します。

### 2.2 文法

この仕様では、C# プログラミング言語の構文を 2 つの文法を使って表します。"字句文法" (2.2.2 を参照) では、行末記号、空白、コメント、トークン、およびプリプロセッサディレクティブを構成するための Unicode 文字の組み合わせ方法を定義します。"構文文法" (2.2.3 を参照) では、字句文法で形成されるトークンを組み合わせて C# プログラムを構成する方法を定義します。

#### 2.2.1 文法表記

字句文法と構文文法は、文法生成規則を使って表します。それぞれの文法生成規則では、非終端記号と、その非終端記号を拡張することで可能な非終端記号または終端記号のシーケンスの生成規則を定義します。文法生成規則では、*non-terminal* (非終端) 記号は斜体で示し、*terminal* (終端) 記号は固定幅のフォントで示します。

文法生成規則の 1 行目は、定義されている非終端記号の名前の後にコロンを付けたものです。インデントされた 2 行目以降の各行は、非終端記号の可能な拡張を非終端記号または終端記号のシーケンスで示します。たとえば、次のようなコードを記述するとします。

```
while-statement:  
    while ( boolean-expression ) embedded-statement
```

この規則は、*while-statement* (while ステートメント) が、順に、*while* トークン、 "(" トークン、*boolean-expression* (ブール式)、 ")" トークン、*embedded-statement* (埋め込みステートメント) で構成されることを定義しています。

非終端記号の展開表記が複数ある場合は、それを別の行で示します。たとえば、次のようなコードを記述するとします。

```
statement-list:  
    statement  
    statement-list statement
```

この生成規則では、*statement-list* が *statement* で構成されるか、または *statement-list* と *statement* で構成されることを定義しています。つまり、この定義は再帰的で、ステートメントリストが 1 つ以上のステートメントで構成されることを示しています。

下付のサフィックス "*opt*" は、省略可能な記号を示すために使われます。次に生成規則の例を示します。

```
block:  
  { statement-listopt }
```

この定義は、以下の定義を省略して表したものです。

```
block:  
  { }  
  { statement-list }
```

この生成規則は、 "{" トークンと "}" トークンで囲まれた省略可能な *statement-list* によって *block* が構成されていることを定義しています。

普通、代替規則はそれぞれ別の行に記述されますが、代替規則が多い場合は、1 行の展開表記の中に列記した代替規則の前に "次のいずれか" という表記を付けて示すことがあります。これは、代替規則をそれぞれ別の行に記述する代わりに省略して表しただけです。たとえば、次のようなコードを記述するとします。

```
real-type-suffix: one of  
  F  f  D  d  M  m
```

この定義は、以下の定義を省略して表したものです。

```
real-type-suffix:  
  F  
  f  
  D  
  d  
  M  
  m
```

### 2.2.2 字句文法

C# の字句文法は 2.3、2.4、2.5 で説明します。字句文法の終端記号は、Unicode 文字セットの文字です。字句文法では、トークン(2.4 を参照)、空白(2.3.3 を参照)、コメント(2.3.2 を参照)、およびプリプロセッサディレクティブ(2.5 を参照)を形成するために文字を組み合わせる方法を指定します。

C# プログラムのすべてのソースファイルは、字句文法の *input* 生成規則(2.3 を参照)に従う必要があります。

### 2.2.3 構文文法

C# の構文文法については、後の章と付録で説明します。構文文法の終端記号は、字句文法で定義されているトークンです。構文文法では、トークンを組み合わせて C# プログラムを構成する方法が指定されています。

C# プログラムのすべてのソースファイルは、構文文法の *compilation-unit* 生成規則(9.1 を参照)に従う必要があります。

## 2.3 字句解析

*input* 生成規則では、C# ソース ファイルの字句構造を定義しています。C# プログラムの各ソース ファイルは、この字句文法生成規則に従う必要があります。

```
input:  
    input-sectionopt  
  
input-section:  
    input-section-part  
    input-section input-section-part  
  
input-section-part:  
    input-elementsopt new-line  
    pp-directive  
  
input-elements:  
    input-element  
    input-elements input-element  
  
input-element:  
    whitespace  
    comment  
    token
```

行末記号(2.3.1 を参照)、空白(2.3.3 を参照)、コメント(2.3.2 を参照)、トークン(2.4 を参照)、プリプロセッサディレクティブ(2.5 を参照)の 5 つの基本要素から、C# ソース ファイルの字句構造が構成されます。これらの基本要素のうち、C# プログラムの構文文法(2.2.3 を参照)で重要なのはトークンだけです。

C# ソース ファイルの字句処理は、ファイルからトークンのシーケンスへの変換で構成されており、このシーケンスは構文解析への入力になります。行末記号、空白、およびコメントを使うとトークンを区切ることができ、プリプロセッサディレクティブを使うとソース ファイルのセクションをスキップできますが、その他の点では、これらの字句要素は C# プログラムの構文構造にどのような影響も及ぼしません。

複数の字句文法生成規則がソース ファイルの文字シーケンスと一致する場合、字句処理は、常に、可能性のある最も長い字句要素を生成します。たとえば、// という文字シーケンスの場合は、單一行コメント開始の字句要素の方が単独の / トークンより長いため、單一行コメントの開始として処理されます。

### 2.3.1 行末記号

行末記号は、C# ソース ファイルの文字列を行に分割します。

```
new-line:  
    Carriage return character (U+000D)  
    Line feed character (U+000A)  
    Carriage return character (U+000D) followed by line feed character (U+000A)  
    Next line character (U+0085)  
    Line separator character (U+2028)  
    Paragraph separator character (U+2029)
```

EOF マーカーを追加するソース コード編集ツールとの互換性を維持するため、および適切に終端処理された行のシーケンスとしてソース ファイルを表示できるようにするために、C# プログラムのすべてのソース ファイルに対して次の変換が適用されます。

- ソース ファイルの最後の文字が Ctrl-Z 文字 (U+001A) の場合は、この文字を削除します。
- ソース ファイルが空ではなく、ソース ファイルの最後の文字がキャリッジ リターン (U+000D)、ライン フィード (U+000A)、行区切り記号 (U+2028)、または段落区切り記号 (U+2029) のいずれでもない場合は、ソース ファイルの最後にキャリッジ リターン文字 (U+000D) を追加します。

### 2.3.2 コメント

コメントの形式としては、單一行コメントと区切り記号付きコメントの 2 種類がサポートされています。"單一行コメント" は、// で始まり、そのソース行の行末まで続きます。"区切り記号付きコメント" は、/\* で始まり、\*/ で終わります。区切り記号付きコメントは、複数の行にわたることができます。

```

comment:
  single-line-comment
  delimited-comment

single-line-comment:
  // input-charactersopt

input-characters:
  input-character
  input-characters input-character

input-character:
  Any Unicode character except a new-line-character

new-line-character:
  Carriage return character (U+000D)
  Line feed character (U+000A)
  Next line character (U+0085)
  Line separator character (U+2028)
  Paragraph separator character (U+2029)

delimited-comment:
  /* delimited-comment-textopt asterisks */

delimited-comment-text:
  delimited-comment-section
  delimited-comment-text delimited-comment-section

delimited-comment-section:
  /
  asterisksopt not-slash-or-asterisk

asterisks:
  *
  asterisks *

not-slash-or-asterisk:
  Any Unicode character except / or *

```

コメントは入れ子にはなりません。//で始まるコメントの内部では、/\*と\*/という文字シーケンスに特別な意味はありません。また、区切り記号付きコメントの内部では、//と/\*という文字シーケンスに特別な意味はありません。

文字リテラルおよびリテラル文字列の中では、コメントは処理されません。

次の例を参照してください。

```
/* Hello, world program
   This program writes "hello, world" to the console
*/
class Hello
{
    static void Main() {
        System.Console.WriteLine("hello, world");
    }
}
```

この例では、区切り記号付きコメントが使用されています。

次の例を参照してください。

```
// Hello, world program
// This program writes "hello, world" to the console
//
class Hello // any name will do for this class
{
    static void Main() { // this method must be named "Main"
        System.Console.WriteLine("hello, world");
    }
}
```

この例では、複数の單一行コメントが示されています。

### 2.3.3 空白

空白文字は、Unicode のクラス Zs の文字(空白文字を含む)および水平タブ文字、垂直タブ文字、フォーム フィード文字と定義されています。

*whitespace:*

- Any character with Unicode class Zs
- Horizontal tab character (U+0009)
- Vertical tab character (U+000B)
- Form feed character (U+000C)

## 2.4 トークン

トークンには、識別子、キーワード、リテラル、演算子、区切り記号など、いくつかの種類があります。空白記号とコメントはトークンではありませんが、トークンの区切り記号として使用できます。

*token:*

- identifier*
- keyword*
- integer-literal*
- real-literal*
- character-literal*
- string-literal*
- operator-or-punctuator*

## 2.4.1 Unicode 文字のエスケープ シーケンス

Unicode 文字のエスケープ シーケンスは Unicode 文字を表します。Unicode 文字エスケープ シーケンスは、識別子 (2.4.2 を参照)、文字リテラル (2.4.4.4 を参照)、通常のリテラル文字列 (2.4.4.5 を参照) 内では処理されます。これ以外の場所 (演算子、区切り記号、キーワードなどを構成する場所など) では、Unicode 文字エスケープは処理されません。

*unicode-escape-sequence:*

```
\u hex-digit hex-digit hex-digit hex-digit
\u hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit
```

Unicode エスケープ シーケンスは、"\u" または "\u" という文字とその後に続く 16 進数で構成され、1 つの Unicode 文字を表します。C# では、文字および文字列の中では Unicode コード ポイントの 16 ビット エンコーディングが使われるため、U+10000 ~ U+10FFFF の範囲の Unicode 文字は、文字リテラルの中では使用できず、リテラル文字列の中では Unicode サロゲートペアを使って表されます。コード ポイントが 0x10FFFF より大きい Unicode 文字は、サポートされていません。

2 段階以上の変換は行われません。たとえば、リテラル文字列 "\u005Cu005C" は、"\\" ではなく "\u005C" に相当します。Unicode 値の \u005C は文字 "\\" です。

次の例を参照してください。

```
class class1
{
    static void Test(bool \u0066) {
        char c = '\u0066';
        if (\u0066)
            System.Console.WriteLine(c.ToString());
    }
}
```

この例には、\u0066 の使い方がいくつか示されています。 \u0066 は、"f" という文字に対するエスケープ シーケンスです。上のプログラムは、次のプログラムと同等です。

```
class class1
{
    static void Test(bool f) {
        char c = 'f';
        if (f)
            System.Console.WriteLine(c.ToString());
    }
}
```

## 2.4.2 識別子

ここで示す識別子の規則は、先頭文字としてアンダースコアを使用できること (C プログラミング言語の従来の規則と同じ)、識別子の中で Unicode エスケープ シーケンスを使用できること、および "@" 文字をプレフィックスとしてすることでキーワードを識別子として使用できることの 3 点を除くと、『Unicode Standard Annex #31』で推奨されている規則に厳密に対応しています。

*identifier:*

```
available-identifier
@ identifier-or-keyword
```

*available-identifier:*

An *identifier-or-keyword* that is not a *keyword*

*identifier-or-keyword:*

```
identifier-start-character identifier-part-charactersopt
```

*identifier-start-character:*

*letter-character*

\_ (the underscore character U+005F)

*identifier-part-characters:*

*identifier-part-character*

*identifier-part-characters identifier-part-character*

*identifier-part-character:*

*letter-character*

*decimal-digit-character*

*connecting-character*

*combining-character*

*formatting-character*

*letter-character:*

A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl

A *unicode-escape-sequence* representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl

*combining-character:*

A Unicode character of classes Mn or Mc

A *unicode-escape-sequence* representing a character of classes Mn or Mc

*decimal-digit-character:*

A Unicode character of the class Nd

A *unicode-escape-sequence* representing a character of the class Nd

*connecting-character:*

A Unicode character of the class Pc

A *unicode-escape-sequence* representing a character of the class Pc

*formatting-character:*

A Unicode character of the class Cf

A *unicode-escape-sequence* representing a character of the class Cf

これらの Unicode 文字クラスについては、『*The Unicode Standard, Version 3.0, section 4.5*』を参照してください。

有効な識別子の例には "identifier1"、"\_identifier2"、"@if" などがあります。

準拠したプログラムでは、『*Unicode Standard Annex #15*』で定義されている Unicode 正規形 C が規定する標準形式の識別子を使用する必要があります。正規形 C を使用していない識別子が検出された場合の動作は、実装で定義されます。ただし、診断は不要です。

プレフィックスとして "@" を付加すると、キーワードを識別子として使うことができ、他のプログラミング言語と整合させる場合に立ちます。文字 @ は実際には識別子の一部ではなく、他の言語ではプレフィックスのない通常の識別子として認識されます。@ プレフィックスを持つ識別子は、逐語的識別子と呼ばれます。キーワードではない識別子に @ プレフィックスを使用してもかまいませんが、スタイルの問題としてできる限り使用しないことをお勧めします。

次に例を示します。

```

class @class
{
    public static void @static(bool @bool) {
        if (@bool)
            System.Console.WriteLine("true");
        else
            System.Console.WriteLine("false");
    }
}

class class1
{
    static void M() {
        c1\u0061ss.st\u0061tic(true);
    }
}

```

この例で定義されている `class` クラスには、"bool" パラメーターを受け取る "static" 静的メソッドがあります。キーワードの中では Unicode エスケープは使用できないため、トークン "c1\u0061ss" は識別子であり、"@class" と同じ識別子であるという点に注意してください。

次の変換がこの順序で適用された結果が等しい場合、2つの識別子は同じものと見なされます。

- プレフィックス "@" が使われている場合は削除します。
- *unicode-escape-sequence* を、対応する Unicode 文字に変換します。
- *formatting-character* を削除します。

連続する 2 つのアンダースコア文字 (U+005F) を含む識別子は、実装で使用するために予約されています。たとえば、実装では、2 つのアンダースコアで始まる拡張キーワードを使用できます。

### 2.4.3 キーワード

"キーワード" は識別子と似た文字シーケンスですが、予約されており、@ 文字を前に付けない限り識別子としては使用できません。

keyword: one of				
abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			

文法内の一場所では、特定の識別子が特別な意味を持ちますが、キーワードではありません。このような識別子は "コンテキスト キーワード" と呼ばれることがあります。たとえば、プロパティ宣言の中では、"get" 識別子と "set" 識別子は特別な意味を持ちます (10.7.2 を参照)。get または set

以外の識別子はこれらの場所では使用できないため、これらの単語を識別子として使用しても衝突しません。他の場合には、暗黙に型指定されたローカル変数宣言 (8.5.1 を参照) の識別子 "var" など、コンテキストキーワードが宣言名と衝突する場合があります。このような場合は、コンテキストキーワードとして使用される識別子よりも宣言名が優先されます。

#### 2.4.4 リテラル

リテラルは、値をソースコードで表したものです。

*literal:*

- boolean-literal*
- integer-literal*
- real-literal*
- character-literal*
- string-literal*
- null-literal*

##### 2.4.4.1 ブール型リテラル

ブール型リテラル値には `true` と `false` の 2 種類があります。

*boolean-literal:*

- `true`
- `false`

*boolean-literal* の型は `bool` です。

##### 2.4.4.2 整数リテラル

整数リテラルは、`int`、`uint`、`long`、`ulong` の各型の値を記述するために使用します。整数リテラルは、10 進数と 16 進数という 2 つの形式で記述できます。

*integer-literal:*

- decimal-integer-literal*
- hexadecimal-integer-literal*

*decimal-integer-literal:*

- decimal-digits integer-type-suffix<sub>opt</sub>*

*decimal-digits:*

- decimal-digit*
- decimal-digits decimal-digit*

*decimal-digit:* one of

- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

*integer-type-suffix:* one of

- `U`
- `u`
- `L`
- `l`
- `UL`
- `uL`
- `u1`
- `LU`
- `Lu`
- `lU`
- `lu`

*hexadecimal-integer-literal:*

- `0x hex-digits integer-type-suffixopt`
- `0X hex-digits integer-type-suffixopt`

*hex-digits:*

- hex-digit*
- hex-digits hex-digit*

*hex-digit:* one of  
 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

整数リテラルの型は、次のようにして決まります。

- サフィックスのないリテラルの場合は、`int`、`uint`、`long`、`ulong` のうち、その値を表すことができる最初の型になります。
- リテラルのサフィックスが `U` または `u` の場合は、`uint` または `ulong` のうち、その値を表すことができる最初の型になります。
- リテラルのサフィックスが `L` または `l` の場合は、`long` または `ulong` がその値を表すことができる最初の型になります。
- リテラルのサフィックスが `UL`、`U1`、`uL`、`u1`、`LU`、`Lu`、`lU`、または `lu` の場合は、`ulong` 型になります。

整数リテラルの値が `ulong` 型の範囲を超えると、コンパイル エラーになります。

スタイルの問題として、文字の "1" は数字の "1" と間違いやさいため、`long` 型のリテラルを記述するときは、"1" ではなく "L" を使うことを推奨します。

`int` 型および `long` 型で表現できる最も小さい値を 10 進整数リテラルとして記述できるようにするために、次の 2 つの規則が適用されます。

- 単項マイナス演算子トークン (7.7.2 を参照) の直後にトークンとして、値が  $2147483648 (2^{31})$  で *integer-type-suffix* を持たない *decimal-integer-literal* が現れた場合、結果は値が  $-2147483648 (-2^{31})$  の `int` 型の定数になります。それ以外のあらゆる状況では、このような *decimal-integer-literal* は `uint` 型になります。
- 単項マイナス演算子トークン (7.7.2 を参照) の直後のトークンとして、値が  $9223372036854775808 (2^{63})$  で、*integer-type-suffix* がないか *integer-type-suffix* が `L` または `l` の *decimal-integer-literal* が現れた場合、結果は値が  $-9223372036854775808 (-2^{63})$  の `long` 型の定数になります。それ以外のあらゆる状況では、このような *decimal-integer-literal* は `ulong` 型になります。

#### 2.4.4.3 実数リテラル

実数リテラルは、`float`、`double`、`decimal` の各型の値を記述するために使用します。

*real-literal:*  
*decimal-digits* . *decimal-digits* *exponent-part*<sub>opt</sub> *real-type-suffix*<sub>opt</sub>  
 . *decimal-digits* *exponent-part*<sub>opt</sub> *real-type-suffix*<sub>opt</sub>  
*decimal-digits* *exponent-part* *real-type-suffix*<sub>opt</sub>  
*decimal-digits* *real-type-suffix*

*exponent-part:*  
 e *sign*<sub>opt</sub> *decimal-digits*  
 E *sign*<sub>opt</sub> *decimal-digits*

*sign:* one of  
 + -

*real-type-suffix:* one of  
 F f D d M m

*real-type-suffix* が指定されていない場合、実数リテラルの型は `double` になります。それ以外の場合は、実数型のサフィックスによって、実数リテラルの型は次のように決まります。

- 実数リテラルのサフィックスが F または f の場合は float 型です。たとえば、1f、1.5f、1e10f、123.456F などのリテラルはすべて float 型です。
- 実数リテラルのサフィックスが D または d の場合は double 型です。たとえば、リテラル 1d、1.5d、1e10d、および 123.456D はすべて double 型です。
- 実数リテラルのサフィックスが M または m の場合は decimal 型です。たとえば、リテラル 1m、1.5m、1e10m、および 123.456M はすべて decimal 型です。このリテラルは、正確な値を受け取ることによって decimal 値に変換されます。必要な場合は、銀行型丸め方式で最も近い表現可能な値に丸められます (4.1.7 を参照)。値が丸められている場合、または値がゼロの場合を除き、リテラルのスケールは保持されます。ゼロの場合は、符号とスケールもゼロになります。したがって、リテラル 2.900m は、符号 0、係数 2900、スケール 3 の 10 進数を形成するとして解析されます。

指定されているリテラル値が示されている型で表現できない場合は、コンパイルエラーが発生します。

float 型または double 型の実数リテラルの値は、IEEE の "四捨五入" モードを使って決定されます。

実数リテラルでは、小数点以下は常に 10 進数字である必要があります。たとえば、1.3F は実数リテラルですが 1.F は実数リテラルではありません。

#### 2.4.4.4 文字リテラル

文字リテラルは 1 文字を表し、普通、'a' のように文字を引用符で囲んで示します。

*character-literal:*

' character '

*character:*

*single-character*

*simple-escape-sequence*

*hexadecimal-escape-sequence*

*unicode-escape-sequence*

*single-character:*

Any character except ' (U+0027), \ (U+005C), and new-line-character

*simple-escape-sequence:* one of

\' \" \\ \0 \a \b \f \n \r \t \v

*hexadecimal-escape-sequence:*

\x hex-digit hex-digit<sub>opt</sub> hex-digit<sub>opt</sub> hex-digit<sub>opt</sub>

character 内の円記号 (\) の後には、'、"、\、0、a、b、f、n、r、t、u、U、x、v のいずれかの文字を指定する必要があります。それ以外の場合は、コンパイル時のエラーが発生します。

16 進エスケープシーケンスは、"\x" の後に続く 16 進数値で構成された値で、1 つの Unicode 文字を表しています。

文字リテラルで表された値が U+FFFF を超える場合は、コンパイルエラーになります。

文字リテラル内の Unicode 文字エスケープシーケンス (2.4.1 を参照) は、U+0000 ~ U+FFFF の範囲内である必要があります。

次の表に示すように、単純なエスケープシーケンスは、Unicode 文字のエンコーディングを表します。

エスケープ シーケンス	文字名	Unicode エン コーディング
\'	单一引用符	0x0027
\\"	二重引用符	0x0022
\\"\\	円記号	0x005C
\0	Null	0x0000
\a	ビープ音	0x0007
\b	バックスペー ス	0x0008
\f	フォーム フィード	0x000C
\n	改行	0x000A
\r	キャリッジリ ターン	0x000D
\t	水平タブ	0x0009
\v	垂直タブ	0x000B

*character-literal* の型は `char` です。

#### 2.4.4.5 リテラル文字列

C# では、"標準リテラル文字列" と "逐語的リテラル文字列" という 2 種類のリテラル文字列がサポートされています。

標準リテラル文字列は、"hello" のように 0 個以上の文字を二重引用符で囲んで指定し、単純なエスケープシーケンス(タブ文字に対する \t など)も、16 進数および Unicode のエスケープシーケンスも含むことができます。

逐語的リテラル文字列は、@ 文字、二重引用符、0 個以上の文字、および閉じる二重引用符で構成されています。@"hello" は逐語的リテラル文字列の簡単な例です。逐語的リテラル文字列では、区切り記号の間の文字は、逐語的に解釈されます。ただし、*quote-escape-sequence* だけは例外です。特に、単純なエスケープシーケンス、16 進数および Unicode のエスケープシーケンスは、逐語的リテラル文字列では処理されません。逐語的リテラル文字列は、複数の行にわたることができます。

*string-literal:*

regular-string-literal  
verbatim-string-literal

*regular-string-literal:*

" regular-string-literal-characters<sub>opt</sub> "

*regular-string-literal-characters:*

regular-string-literal-character  
regular-string-literal-characters regular-string-literal-character

```

regular-string-literal-character:
  single-regular-string-literal-character
  simple-escape-sequence
  hexadecimal-escape-sequence
  unicode-escape-sequence

single-regular-string-literal-character:
  Any character except " (U+0022), \ (U+005C), and new-line-character

verbatim-string-literal:
  @" verbatim-string-literal-charactersopt "

verbatim-string-literal-characters:
  verbatim-string-literal-character
  verbatim-string-literal-characters verbatim-string-literal-character

verbatim-string-literal-character:
  single-verbatim-string-literal-character
  quote-escape-sequence

single-verbatim-string-literal-character:
  Any character except "

quote-escape-sequence:
  """

```

*regular-string-literal-character* 内の円記号 (\) の後には、'、"、\、0、a、b、f、n、r、t、u、U、x、v のいずれかの文字を指定する必要があります。それ以外の場合は、コンパイル時のエラーが発生します。

次の例を参照してください。

```

string a = "hello, world";           // hello, world
string b = @"hello, world";          // hello, world
string c = "hello \t world";         // hello      world
string d = @"hello \t world";        // hello \t world
string e = "Joe said \"Hello\" to me"; // Joe said "Hello" to me
string f = @"Joe said ""Hello"" to me"; // Joe said "Hello" to me
string g = @"\\\server\\share\\file.txt"; // \\server\\share\\file.txt
string h = @"\\server\\share\\file.txt"; // \\server\\share\\file.txt
string i = "one\r\n\two\r\nthree";
string j = @"one
two
three";

```

これらは、さまざまなりテラル文字列の例です。最後のリテラル文字列 j は、複数の行にまたがる逐語的リテラル文字列の例です。引用符の間にある文字は、改行文字などの空白記号を含めて、そのまま維持されます。

16進値のエスケープ シーケンスで指定できる 16進数字の個数は可変であるため、リテラル文字列 "\x123" には、123 という 16進値の 1 個の文字が含まれます。12 という 16進値の文字に続けて 3 という文字を含む文字列を作成するには、"\x00123" または "\x12" + "3" と記述します。

*string-literal* の型は **string** です。

リテラル文字列ごとに新しい文字列インスタンスが作成されるとは限りません。同じプログラム内の複数のリテラル文字列が文字列等値演算子 (7.10.7 を参照) によって等しくなっている場合、これらの

リテラル文字列は同じ文字列インスタンスを参照します。たとえば、次のようなプログラムを考えます。

```
class Test
{
    static void Main()
    {
        object a = "hello";
        object b = "hello";
        System.Console.WriteLine(a == b);
    }
}
```

この例の 2 つのリテラルは同じ文字列インスタンスを参照しているため、出力は `True` になります。

#### 2.4.4.6 null リテラル<

*null-literal:*  
null

*null-literal* は、参照型や `null` 許容型に暗黙的に変換できます。

#### 2.4.5 演算子と区切り記号

C# には、複数の演算子と区切り記号があります。演算子は式の中で使われて、オペランドが関係する演算を示します。たとえば、`a + b` という式では、演算子 `+` を使って、オペランド `a` と `b` が加算されます。区切り記号は、グループ化と分離に使用します。

*operator-or-punctuator:* one of  
{      }      [      ]      (      )      .      ,      :      ;  
+      -      \*      /      %      &      |      ^      !      ~  
=      <      >      ?      ??      ::      ++      --      &&      ||  
->      ==      !=      <=      >=      +=      -=      \*=      /=      %=  
&=      |=      ^=      <<      <<=      =>

*right-shift:*  
  >/>

*right-shift-assignment:*  
  >/>=

*right-shift* 生成および *right-shift-assignment* 生成における縦棒は、構文文法の他の生成とは異なり、トークンの間にはどのような種類の文字(空白も含む)も使用できないことを示しています。これらの生成規則は、*type-parameter-list* を正しく処理できるように特別に扱われます(10.1.3 を参照)。

#### 2.5 プリプロセッサ ディレクティブ

プリプロセッサ ディレクティブには、ソースファイルの特定部分の条件付き省略、エラー状態や警告状態の通知、ソースコードの異なる領域の線引きなどの機能があります。“プリプロセッサ ディレクティブ”という用語は、C 言語および C++ 言語との整合性のためにのみ使われています。C# には独立したプリプロセス ステップはなく、プリプロセッサ ディレクティブは、字句解析フェーズの中で処理されます。

*pp-directive:*  
  *pp-declaration*  
  *pp-conditional*  
  *pp-line*  
  *pp-diagnostic*  
  *pp-region*  
  *pp-pragma*

以下のプリプロセッサディレクティブを使用できます。

- **#define** と **#undef** は、それぞれ、条件付きコンパイルシンボルを定義したり未定義にしたりするするために使用します (2.5.3 を参照)。
- **#if**、**#elif**、**#else**、および **#endif** はソースコードの一部分を条件付きでスキップするために使用します (2.5.4 を参照)。
- **#line** は、エラーや警告に対する行番号の表示を制御するために使用します (2.5.7 を参照)。
- **#error** と **#warning** は、それぞれ、エラーや警告を出力するために使用します (2.5.5 を参照)。
- **#region** と **#endregion** は、ソースコードのセクションを明示的に示すために使用します (2.5.6 を参照)。
- **#pragma** は、コンパイラに対してオプションのコンテキスト情報を指定するために使用します (2.5.8 を参照)。

プリプロセッサディレクティブは、ソースコードでは常に、独立した行に記述し、#を前に付けたプリプロセッサディレクティブ名で始めます。#の前、および#とディレクティブ名の間には、空白を挿入できます。

**#define**、**#undef**、**#if**、**#elif**、**#else**、**#endif**、**#line**、または **#endregion** ディレクティブを含むソース行の最後には、單一行コメントを付けてもかまいません。プリプロセッサディレクティブを含むソース行では、区切り記号付きコメント (`/* */` の形式のコメント) は使用できません。

プリプロセッサディレクティブは、トークンではなく、C#の構文文法の一部でもありません。ただし、プリプロセッサディレクティブを使うことで、一連のトークンを有効にしたり無効にしたりでき、それによってC#プログラムに影響を与えることができます。次に示すプログラムは、この場合の例です。

```
#define A
#undef B
class C
{
#if A
    void F() {}
#else
    void G() {}
#endif
#if B
    void H() {}
#else
    void I() {}
#endif
}
```

このプログラムをコンパイルすると、次のプログラムとまったく同じ一連のトークンが生成されます。

```
class C
{
    void F() {}
    void I() {}
}
```

つまり、2つのプログラムは字句的にはまったく異なっていますが、構文的には同じものです。

### 2.5.1 条件付きコンパイル シンボル

`#if`、`#elif`、`#else`、および`#endif`の各ディレクティブによって実現される条件付きコンパイル機能は、プリプロセス式(2.5.2を参照)と条件付きコンパイルシンボルで制御されます。

*conditional-symbol:*

Any identifier-or-keyword except `true` or `false`

条件付きコンパイルシンボルには、**定義済み**と**未定義**の2つの状態があります。ソースファイルの字句処理が始まるときの条件付きコンパイルシンボルの状態は、外部機構(コマンドラインコンパイラオプションなど)によって明示的に定義されているのでない限り、未定義になっています。

`#define` ディレクティブが処理されると、ディレクティブで指定されている条件付きコンパイルシンボルが、そのソースファイルで定義済み状態になります。同じシンボルに対する`#undef` ディレクティブが処理されるまで、またはソースファイルが終わるまで、シンボルは定義済みのままになります。あるソースファイルで指定されている`#define` ディレクティブと`#undef` ディレクティブは、同じプログラムの他のソースファイルには影響を与えません。

プリプロセス式の中で参照される場合、定義済みの条件付きコンパイルシンボルはブール値の`true`になります、未定義の条件付きコンパイルシンボルは`false`になります。プリプロセス式の中で参照する前に、条件付きコンパイルシンボルを明示的に宣言しておく必要はありません。宣言されていないシンボルは、未定義として扱われて、ブール値`false`になるだけです。

条件付きコンパイルシンボルに対する名前空間は、C#プログラムの他のあらゆる名前付きエンティティとは別に独立しています。条件付きコンパイルシンボルは、`#define` ディレクティブと`#undef` ディレクティブおよびプリプロセス式の中だけで参照できます。

### 2.5.2 プリプロセス式

プリプロセス式は、`#if` ディレクティブと`#elif` ディレクティブの中で使用できます。プリプロセス式では、`!`、`==`、`!=`、`&&`、および`||` の各演算子、およびグループ化のためのかっこを使用できます。

*pp-expression:*

*whitespace<sub>opt</sub>* *pp-or-expression* *whitespace<sub>opt</sub>*

*pp-or-expression:*

*pp-and-expression*

*pp-or-expression* *whitespace<sub>opt</sub>* `||` *whitespace<sub>opt</sub>* *pp-and-expression*

*pp-and-expression:*

*pp-equality-expression*

*pp-and-expression* *whitespace<sub>opt</sub>* `&&` *whitespace<sub>opt</sub>* *pp-equality-expression*

*pp-equality-expression:*

*pp-unary-expression*

*pp-equality-expression* *whitespace<sub>opt</sub>* `==` *whitespace<sub>opt</sub>* *pp-unary-expression*

*pp-equality-expression* *whitespace<sub>opt</sub>* `!=` *whitespace<sub>opt</sub>* *pp-unary-expression*

```
pp-unary-expression:
  pp-primary-expression
    ! whitespaceopt pp-unary-expression
```

```
pp-primary-expression:
  true
  false
  conditional-symbol
  ( whitespaceopt pp-expression whitespaceopt )
```

プリプロセス式の中で参照される場合、定義済みの条件付きコンパイルシンボルはブール値の `true` になり、未定義の条件付きコンパイルシンボルは `false` になります。

プリプロセス式を評価すると、結果は常にブール値になります。プリプロセス式の評価規則は、参照可能なユーザー定義のエンティティだけが条件付きコンパイルシンボルであることを除けば、定数式に対する評価規則(7.19を参照)と同じです。

### 2.5.3 宣言ディレクティブ

宣言ディレクティブは、条件付きコンパイルシンボルを定義済みまたは未定義にするために使用します。

```
pp-declaration:
  whitespaceopt # whitespaceopt define whitespace conditional-symbol pp-new-line
  whitespaceopt # whitespaceopt undef whitespace conditional-symbol pp-new-line

pp-new-line:
  whitespaceopt single-line-commentopt new-line
```

`#define` ディレクティブが処理されると、指定されている条件付きコンパイルシンボルが、ディレクティブの次のソース行以降において、定義済みの状態になります。`#undef` ディレクティブが処理されると、指定されている条件付きコンパイルシンボルが、ディレクティブの次のソース行以降において、未定義の状態になります。

ソースファイル中のすべての `#define` ディレクティブと `#undef` ディレクティブは、ソースファイルの最初のトークン(2.4を参照)よりも前に指定する必要があります。そうでない場合、コンパイルエラーが発生します。つまり、`#define` ディレクティブと `#undef` ディレクティブは、ソースファイルの"実際のコード"よりも前で指定する必要があります。

次に例を示します。

```
#define Enterprise
#if Professional || Enterprise
#define Advanced
#endif
namespace Megacorp.Data
{
  #if Advanced
  class PivotTable {...}
  #endif
}
```

ソースファイルの最初のトークン(`namespace`キーワード)よりも前に `#define` ディレクティブがあるため、この例は有効です。

次のプログラムでは、`#define` ディレクティブの前に実際のコードがあるため、コンパイルエラーになります。

```
#define A
namespace N
{
    #define B
    #if B
    class Class1 {}
    #endif
}
```

`#define` ディレクティブは、既に定義されている条件付きコンパイルシンボルを、`#undef` で未定義にせずに定義できます。次の例では、条件付きコンパイルシンボル `A` を 1 回定義した後で、再度定義しています。

```
#define A
#define A
```

`#undef` ディレクティブでは、定義されていない条件付きコンパイルシンボルを未定義にできます。次の例では、条件付きコンパイルシンボル `A` を定義してから、2 回未定義にしています。2 回目の `#undef` は何も行いませんが、有効です。

```
#define A
#undef A
#undef A
```

#### 2.5.4 条件付きコンパイル ディレクティブ

条件付きコンパイルディレクティブは、ソースファイルの一部分を条件付きで有効または無効にするために使用します。

```
pp-conditional:
    pp-if-section pp-elif-sectionsopt pp-else-sectionopt pp-endif

pp-if-section:
    whitespaceopt # whitespaceopt if whitespace pp-expression pp-new-line conditional-
    sectionopt

pp-elif-sections:
    pp-elif-section
    pp-elif-sections pp-elif-section

pp-elif-section:
    whitespaceopt # whitespaceopt elif whitespace pp-expression pp-new-line conditional-
    sectionopt

pp-else-section:
    whitespaceopt # whitespaceopt else pp-new-line conditional-sectionopt

pp-endif:
    whitespaceopt # whitespaceopt endif pp-new-line

conditional-section:
    input-section
    skipped-section

skipped-section:
    skipped-section-part
    skipped-section skipped-section-part
```

```

skipped-section-part:
  skipped-charactersopt new-line
  pp-directive

skipped-characters:
  whitespaceopt not-number-sign input-charactersopt

not-number-sign:
  Any input-character except #

```

構文で示しているように、条件付きコンパイルディレクティブは、`#if` ディレクティブ、0 個以上の`#elif` ディレクティブ、0 個以上の`#else` ディレクティブ、および`#endif` ディレクティブをこの順序で指定したもので構成される組として記述する必要があります。ディレクティブの間は、ソースコードの条件セクションです。各セクションは、すぐ前のディレクティブによって制御されます。条件セクション自体でも、入れ子にした条件付きコンパイルディレクティブを指定できます。ただし、このようなディレクティブは、上で示した完全な組を構成している必要があります。

*pp-conditional* では、含まれている *conditional-section* の中の多くても 1 つが、通常の字句処理のために選択されます。

- `#if` ディレクティブおよび`#elif` ディレクティブの *pp-expression* は、結果が `true` になるまで、記述されているとおりの順序で評価されます。式が `true` と評価された場合は、対応するディレクティブの *conditional-section* が選択されます。
- *pp-expression* がすべて `false` になり、`#else` ディレクティブがある場合は、`#else` ディレクティブの *conditional-section* が選択されます。
- それ以外の場合、*conditional-section* は選択されません。

*conditional-section* が選択された場合、そのセクションは通常の *input-section* として処理されます。セクションに含まれるソースコードは、字句文法に準拠している必要があります。セクション内のソースコードからトークンが生成されて、セクション内のプリプロセッサディレクティブは既定どおりに機能します。

選択されない *conditional-section* がある場合、そのセクションは *skipped-section* として処理されます。プリプロセッサディレクティブを除き、セクション内のソースコードは字句文法に準拠している必要はありません。セクション内のソースコードからトークンは生成されません。セクション内のプリプロセッサディレクティブは、字句文法に準拠している必要がありますが、それ以外に関しては処理されません。*skipped-section* として処理される *conditional-section* 内では、入れ子の (`#if...#endif` 構造と `#region...#endregion` 構造内に入れ子になっている) *conditional-section* はすべて、*skipped-section* として処理されます。

次に示すのは、条件付きコンパイルディレクティブを入れ子にする方法の例です。

```

#define Debug      // Debugging on
#undef Trace     // Tracing off

```

```

class PurchaseTransaction
{
    void Commit() {
        #if Debug
            CheckConsistency();
        #if Trace
            WriteToLog(this.ToString());
        #endif
        #endif
        CommitHelper();
    }
}

```

プリプロセッサディレクティブを除き、スキップされるソースコードは字句解析の対象にはなりません。たとえば、次に示す例は、`#else`セクションに終了していないコメントがありますが、それでも有効です。

```

#define Debug      // Debugging on
class PurchaseTransaction
{
    void Commit() {
        #if Debug
            CheckConsistency();
        #else
            /* Do something else
        #endif
    }
}

```

ただし、プリプロセッサディレクティブは、ソースコードのスキップされるセクションであっても字句文法的に正しく記述されている必要があることに注意してください。

複数行入力要素の内部にあるプリプロセッサディレクティブは、処理されません。次に示すプログラムは、この場合の例です。

```

class Hello
{
    static void Main() {
        System.Console.WriteLine(@"hello,
#if Debug
    world
#else
    Nebraska
#endif
    ");
}

```

出力は次のようになります。

```

hello,
#if Debug
    world
#else
    Nebraska
#endif

```

特別なケースとして、処理されるプリプロセッサディレクティブの集合が、*pp-expression* の評価に依存する場合があります。次に例を示します。

```
#if X
/*
#else
/* */ class Q { }
#endif
```

この例では、`X` が定義されているかどうかにかかわらず、常に同じトークンストリーム (`class Q { }`) が生成されます。`X` が定義されている場合は、複数行コメントになるために、`#if` ディレクティブと `#endif` ディレクティブだけが処理されます。`X` が定義されていない場合は、3つのディレクティブ (`#if`、`#else`、および `#endif`) が、ディレクティブの組の一部になります。

## 2.5.5 診断ディレクティブ

診断ディレクティブは、エラーメッセージや警告メッセージを明示的に生成するために使用されます。このようなメッセージは、コンパイル時の他のエラーや警告と同じ方法で報告されます。

```
pp-diagnostic:
    whitespaceopt # whitespaceopt error pp-message
    whitespaceopt # whitespaceopt warning pp-message

pp-message:
    new-line
    whitespace input-charactersopt new-line
```

次に例を示します。

```
#warning Code review needed before check-in
#if Debug && Retail
    #error A build can't be both debug and retail
#endif

class Test {...}
```

この例では、"Code review needed before check" という警告が常に生成されます。さらに条件シンボルの `Debug` と `Retail` が両方とも定義されていると、"A build can't be both debug and retail" というコンパイルエラーも生成されます。`pp-message` には、任意のテキストを含めることができます。つまり、単語 `can't` の單一引用符のような、整形式のトークンを含める必要はありません。

## 2.5.6 領域ディレクティブ

領域ディレクティブは、ソースコードの領域を明示的に示すために使用されます。

```
pp-region:
    pp-start-region conditional-sectionopt pp-end-region

pp-start-region:
    whitespaceopt # whitespaceopt region pp-message

pp-end-region:
    whitespaceopt # whitespaceopt endregion pp-message
```

領域に特別な意味が付加されることはありません。領域は、プログラマや自動化ツールがソースコードのセクションを作成するときに使用するためのものです。同様に、`#region` ディレクティブや `#endregion` ディレクティブで指定されるメッセージにも特別な意味はありません。領域を識別するためだけに使用されます。対応する `#region` ディレクティブと `#endregion` ディレクティブで、`pp-message` が異なっていてもかまいません。

領域の字句処理は次のように行われます。

```
#region
...
#endregion
```

この領域ディレクティブは、次の形式の条件付きコンパイルディレクティブの字句処理とまったく同じです。

```
#if true
...
#endif
```

## 2.5.7 行ディレクティブ

行ディレクティブを使うと、警告やエラーなどのコンパイラ出力で報告され、呼び出し元情報属性で使用される行番号とソースファイル名を変更できます(17.4.4を参照)。

行ディレクティブは、他のテキスト入力からC#のソースコードを生成するメタプログラミングツールで最もよく使われます。

```
pp-line:
    whitespaceopt # whitespaceopt line whitespace line-indicator pp-new-line

line-indicator:
    decimal-digits whitespace file-name
    decimal-digits
    default
    hidden

file-name:
    " file-name-characters "

file-name-characters:
    file-name-character
    file-name-characters file-name-character

file-name-character:
    Any input-character except "
```

`#line`ディレクティブがないと、コンパイラの出力では、実際の行番号とソースファイル名が報告されます。`default`以外の`line-indicator`を含む`#line`ディレクティブを処理するときに、コンパイラは、ディレクティブの"後"にある行を、指定された行番号として(ファイル名が指定されている場合は、そのファイル名で)扱います。

`#line default`ディレクティブは、それより前にあるすべての`#line`ディレクティブの効果を元に戻します。コンパイラは、それ以降の行については、`#line`ディレクティブがそれまでにまったく処理されていない場合と同じように、本当の行情報を報告します。

`#line hidden`ディレクティブは、エラーメッセージで報告されるファイルや行番号への効果はありませんが、ソースレベルのデバッグで効果があります。デバッグ時に、`#line hidden`ディレクティブから次の`#line`ディレクティブ(`#line hidden`ではない)までのすべての行には、行番号情報がありません。デバッガーでコードをステップ実行する場合、これらの行はすべてスキップされます。

`file-name`は通常のリテラル文字列と異なり、エスケープ文字が処理されません。つまり、`file-name`では、'\'文字は単に通常の円記号を示します。

## 2.5.8 pragma ディレクティブ

プリプロセッサディレクティブ `#pragma` を使用して、コンパイラに対してオプションのコンテキスト情報を指定できます。`#pragma` ディレクティブで指定された情報がプログラムのセマンティクスを変更することはありません。

```
pp-pragma:  
    whitespaceopt # whitespaceopt pragma whitespace pragma-body pp-new-line  
  
pragma-body:  
    pragma-warning-body
```

C# には、コンパイラの警告を制御するための `#pragma` ディレクティブがあります。言語の将来のバージョンでは、他の `#pragma` ディレクティブも追加される可能性があります。他の C# コンパイラとの相互運用性を保証するために、Microsoft C# コンパイラは未知の `#pragma` ディレクティブがあつてもコンパイルエラーは発行しません。ただし、警告は生成します。

### 2.5.8.1 pragma warning

`#pragma warning` ディレクティブを使用して、その後のプログラム テキストのコンパイル時に、警告メッセージのすべてまたは一部を無効にしたり有効に戻したりできます。

```
pragma-warning-body:  
    warning whitespace warning-action  
    warning whitespace warning-action whitespace warning-list  
  
warning-action:  
    disable  
    restore  
  
warning-list:  
    decimal-digits  
    warning-list whitespaceopt , whitespaceopt decimal-digits
```

警告リストを指定しない `#pragma warning` ディレクティブは、すべての警告に影響します。警告リストを含む `#pragma warning` ディレクティブは、そのリストに指定されている警告のみに影響します。

`#pragma warning disable` ディレクティブは、警告のすべてまたは指定されたセットを無効化します。

`#pragma warning restore` ディレクティブは、警告のすべてまたは指定されたセットを、コンパイル単位の先頭で有効だった状態に復元します。外部から無効にした警告は、警告の全部か一部かを問わず、`#pragma warning restore` を使用して有効に戻すことはできないので注意してください。

次の例は、旧式のメンバーが参照されたときの警告を一時的に無効にするための `#pragma warning` の使い方を示します。警告番号は Microsoft C# コンパイラのものです。

```
using System;  
class Program  
{  
    [Obsolete]  
    static void Foo() {}
```

## C# LANGUAGE SPECIFICATION

```
static void Main() {
    #pragma warning disable 612
    Foo();
    #pragma warning restore 612
}
```

## 3. 基本概念

### 3.1 アプリケーションの起動

エントリ ポイントを持つアセンブリは、アプリケーションと呼ばれます。アプリケーションを実行すると、新しいアプリケーション ドメインが作成されます。1つのアプリケーションの異なるインスタンスが、同じコンピューターに同時に存在できます。このとき、各インスタンスが独自のアプリケーション ドメインを持ちます。

アプリケーション ドメインは、アプリケーション状態のコンテナーとして機能することにより、アプリケーションの分離を可能にします。アプリケーション ドメインは、アプリケーションで定義した型と、使用するライブラリのコンテナーおよび境界として機能します。あるアプリケーション ドメインに読み込まれた型は、別のアプリケーション ドメインに読み込まれた型と区別されます。オブジェクトのインスタンスは、アプリケーション ドメインの間で直接共有されません。たとえば、各アプリケーション ドメインは、これらの型の静的変数のコピーを持ちます。型の静的コンストラクターは、アプリケーション ドメインごとに多くても1回しか実行されません。実装では、アプリケーション ドメインの構築と破棄に関する、実装固有のポリシーや機構を自由に提供できます。

アプリケーションの起動は、指定されたメソッドを実行環境が呼び出すことで行われます。このメソッドは、アプリケーションのエントリ ポイントと呼ばれます。エントリ ポイントメソッドの名前は常に `Main` で、シグネチャは次のいずれかです。

```
static void Main() {...}
static void Main(string[] args) {...}
static int Main() {...}
static int Main(string[] args) {...}
```

この例のように、エントリ ポイントは、`int` 型の値を返してもかまいません。この戻り値は、アプリケーションの終了時に使用されます(3.2を参照)。

エントリ ポイントは、オプションとして、仮パラメーターを1つ持つことができます。パラメータ名は任意ですが、パラメーターの型は `string[]` であることが必要です。仮パラメーターが存在する場合、実行環境は、アプリケーションの起動時に指定されたコマンド ライン引数が含まれている `string[]` 引数を作成して渡します。`string[]` 引数は `null` ではありませんが、コマンド ライン引数が指定されていない場合は長さが 0 の場合があります。

C# はメソッドのオーバーロードをサポートしているため、シグネチャが異なってさえいれば、クラスや構造体はメソッドの複数の定義を保持できます。ただし、1つのプログラム内では、クラスや構造体はアプリケーションのエントリ ポイントとして使用されるように定義された `Main` という名前のメソッドを2つ以上含むことはできません。ただし、`Main` をオーバーロードしたメソッドがあってもかまわるのは、パラメーターが複数ある場合、またはパラメーターが1つだけでも `string[]` 以外の型である場合です。

アプリケーションは、複数のクラスまたは構造体で構成できます。これらの複数のクラスまたは構造体に、`Main` という名前を持ち、アプリケーションのエントリ ポイントとして使用するための条件を満たすように定義されたメソッドが含まれていてもかまいません。その場合は、コマンド ラインコ

ンパイラ オプションなどの外部機構を使用して、いずれかの `Main` メソッドをエントリ ポイントとして選択する必要があります。

C# では、すべてのメソッドは、クラスまたは構造体のメンバーとして定義されている必要があります。普通、メソッドのアクセシビリティ (3.5.1 を参照) は、メソッドの宣言で指定されているアクセス修飾子 (10.3.5 を参照) によって決まり、型のアクセシビリティは、型の宣言で指定されているアクセス修飾子によって決まります。ある型のあるメソッドが呼び出し可能であるためには、型とメソッドの両方がアクセス可能であることが必要です。ただし、アプリケーションのエントリ ポイントは特殊なケースです。エントリ ポイントの宣言されたアクセシビリティにも、エントリ ポイントを囲む型宣言の宣言されたアクセシビリティにも関係なく、実行環境はアプリケーションのエントリ ポイントにアクセスできます。

アプリケーションのエントリ ポイント メソッドは、ジェネリック クラス宣言内では使用できません。それ以外の点に関しては、エントリ ポイント メソッドは、エントリ ポイントではないメソッドと同じように動作します。

### 3.2 アプリケーションの終了

アプリケーションの終了時には、制御が実行環境に戻ります。

アプリケーションの "エントリ ポイント" メソッドの戻り値の型が `int` の場合、返される値はアプリケーションの "終了ステータス コード" になります。このコードを利用することで、プログラムの成功または失敗を実行環境に通知できます。

エントリ ポイント メソッドの戻り値の型が `void` である場合、そのメソッドを終了する右中かっこ `()` に到達するか、式を持たない `return` ステートメントが実行され、終了ステータス コードが `0` になります。

アプリケーションの終了前に、ガベージコレクションが行われていないすべてのオブジェクトに対してデストラクターが呼び出されます。ただし、ライブラリ メソッド `GC.SuppressFinalize` などの呼び出しによってアプリケーションのクリーンアップが抑止されていない場合に限ります。

### 3.3 宣言

C# プログラムにおける宣言とは、プログラムの構成要素を定義することです。C# のプログラムは、名前空間 (9 を参照) を使って構成されており、名前空間は、型の宣言および入れ子になった名前空間の宣言を含むことができます。型の宣言 (9.6 を参照) を使用して、クラス (10 を参照)、構造体 (10.14 を参照)、インターフェイス (13 を参照)、列挙型 (14 を参照)、およびデリゲート (15 を参照) を定義します。型宣言で指定できるメンバーの種類は、型宣言の形式によって異なります。たとえば、クラスの宣言により、定数 (10.4 を参照)、フィールド (10.5 を参照)、メソッド (10.6 を参照)、プロパティ (10.7 を参照)、イベント (10.8 を参照)、インデクサー (10.9 を参照)、演算子 (10.10 を参照)、インスタンス コンストラクター (10.11 を参照)、静的コンストラクター (10.12 を参照)、デストラクター (10.13 を参照)、および入れ子の型 (10.3.8 を参照) を宣言できます。

宣言では、宣言が属している宣言空間内の名前を定義します。オーバーロードされたメンバー (3.6 を参照) の場合を除き、複数の宣言で、宣言空間に同じ名前のメンバーを導入するとコンパイルエラーになります。同じ名前で異なる種類のメンバーを 1 つの宣言空間に収めることはできません。たとえば、1 つの宣言空間に、同じ名前のフィールドとメソッドを入れることはできません。

宣言空間には、以下のようにいくつかの種類があります。

- プログラムのすべてのソースファイルにおいて、*namespace-declaration* の外にある *namespace-member-declaration* は、**グローバル宣言空間**と呼ばれる单一の結合された宣言空間のメンバーです。
- プログラムのすべてのソースファイルにおいて、同一の完全修飾名前空間名を持つ *namespace-declaration* 内の *namespace-member-declaration* は、单一の結合された宣言空間のメンバーです。
- クラス、構造体、インターフェイスの各宣言は、新しい宣言空間を作成します。名前は、*class-member-declaration*、*struct-member-declaration*、*interface-member-declaration*、または *type-parameter* を通じてこの宣言空間に導入されます。オーバーロードされたインスタンス コンストラクター宣言および静的コンストラクター宣言を除き、クラスまたは構造体には、そのクラスまたは構造体と同じ名前を持つメンバー宣言を含めることはできません。クラス、構造体、またはインターフェイスでは、オーバーロードされたメソッドおよびインデクサーを宣言できます。さらに、クラスまたは構造体では、オーバーロードされたインスタンス コンストラクターおよび演算子を宣言できます。たとえば、クラス、構造体、またはインターフェイスでは、メソッド宣言のシグネチャが異なる限り、同じ名前で複数のメソッドを宣言できます(3.6 を参照)。基底クラスはクラスの宣言空間には関係なく、基本インターフェイスはインターフェイスの宣言空間には関係ありません。したがって、派生クラスまたは派生インターフェイスでは、継承されているメンバーと同じ名前でメンバーを宣言できます。このようなメンバーは、継承されているメンバーを隠ぺいしています。
- 各デリゲート宣言では、新しい宣言空間が作成されます。名前は、仮パラメーター (*fixed-parameter* と *parameter-array*) および *type-parameter* を使用してこの宣言空間に導入されます。
- 各列挙型宣言では、新しい宣言空間が作成されます。名前は、*enum-member-declaration* を通じてこの宣言空間に導入されます。
- メソッド、インデクサー、演算子、インスタンス コンストラクター、匿名関数がそれぞれ宣言されるたびに、**ローカル変数宣言空間**と呼ばれる新しい宣言空間が作成されます。名前は、仮パラメーター (*fixed-parameter* と *parameter-array*) および *type-parameter* を通じて、この宣言空間に導入されます。関数メンバーまたは匿名関数が存在する場合、その本体はローカル変数宣言空間の中で入れ子になっていると考えられます。ローカル変数宣言空間および入れ子になっているローカル変数宣言空間に同じ名前の要素が含まれるとエラーになります。したがって、入れ子になっている宣言空間の中では、外側の宣言空間のローカル変数または定数と同じ名前のローカル変数または定数を宣言することはできません。宣言空間が入れ子になっていない場合は、2つの宣言空間に同じ名前の要素を含むことができます。
- 各 *block* \b または *switch-block* \b \t "See declaration space, block and" \b \t "See declaration space, block and" \b は、*for*、*foreach*、*using* の各ステートメントと同様に、ローカル変数とローカル定数に対してローカル変数宣言空間を作成します \b \t "See declaration space, block and" \b \t "See declaration space, switch block"。名前は、*local-variable-declaration* と *local-constant-declaration* を通じてこの宣言空間に導入されます。関数メンバーまたは匿名関数の本体として発生するブロック、またはその本体の中に発生するブロックは、パラメーターに対応する関数により宣言されたローカル変数宣言空間の内部で入れ子になります。そのため、たとえば同じ名前のローカル変数とパラメーターが1つのメソッドで使用されると、エラーになります。
- 各 *block* または *switch-block* は、各ラベルに対して、個別に宣言空間を作成します。名前は、*labeled-statement* を通じてこの宣言空間に導入され、*goto-statement* を通じて参照されます。ブロックの**ラベル宣言空間**には入れ子になったブロックが含まれます。したがって、入れ子になっ

ているブロックの中では、外側のブロックのラベルと同じ名前のラベルを宣言することはできません。

一般に、名前を宣言する記述上の順序は重要ではありません。特に、名前空間、定数、メソッド、プロパティ、イベント、インデクサー、演算子、インスタンス コンストラクター、デストラクター、静的コンストラクター、および型を宣言したり使用したりする場合、記述の順序は重要ではありません。宣言の順序は、以下の場合に重要です。

- フィールドおよびローカル変数の場合は、宣言の順序により、初期化子(ある場合)が実行される順序が決まります。
- ローカル変数は、使用する前に定義しておく必要があります(3.7を参照)。
- *constant-expression* の値を省略する場合、列挙メンバーの宣言順序(14.3を参照)は重要です。

名前空間の宣言空間は "閉じてはいない" ため、完全修飾名が同じ 2 つの名前空間の宣言は、同じ宣言空間に含まれます。次に例を示します。

```
namespace Megacorp.Data
{
    class Customer
    {
        ...
    }
}

namespace Megacorp.Data
{
    class Order
    {
        ...
    }
}
```

上の例では、2 つの名前空間宣言は同じ宣言空間に含まれ、完全修飾名が `Megacorp.Data.Customer` と `Megacorp.Data.Order` である 2 つのクラスが宣言されています。2 つの宣言は同じ宣言空間に対するものなので、それぞれに同じ名前のクラスの宣言が含まれているとコンパイル エラーになります。

上記で指定されているように、ブロックの宣言空間には、入れ子になっているすべてのブロックが含まれます。したがって、次の例では、外側のブロックで宣言されている `i` という名前は、内側のブロックで再度宣言できないため、`F` メソッドと `G` メソッドはコンパイル エラーになります。ただし、`H` メソッドと `I` メソッドの `i` は入れ子ではない独立したブロックで宣言されているため、この 2 つのメソッドは有効です。

```
class A
{
    void F()
    {
        int i = 0;
        if (true)
        {
            int i = 1;
        }
    }

    void G()
    {
        if (true)
        {
            int i = 0;
        }
        int i = 1;
    }
}
```

```

void H()
{
    if (true) {
        int i = 0;
    }
    if (true) {
        int i = 1;
    }
}

void I()
{
    for (int i = 0; i < 10; i++)
        H();
    for (int i = 0; i < 10; i++)
        H();
}
}

```

## 3.4 メンバー

名前空間と型にはメンバーがあります。エンティティのメンバーは、普通、エンティティに対する参照、".." トークン、およびメンバーの名前で構成される修飾名を使用することで利用できます。

型のメンバーは、その型の中で宣言するか、または型の基底クラスから継承します。基底クラスから型を継承すると、インスタンスコンストラクター、デストラクター、および静的コンストラクターを除く基底クラスのすべてのメンバーは、派生型のメンバーになります。基底クラスのメンバーに宣言されたアクセシビリティが、メンバーの継承の可否を左右することはありません。インスタンスコンストラクター、静的コンストラクター、デストラクター以外のすべてのメンバーが、継承の対象になります。ただし、継承元のメンバーに宣言されたアクセシビリティのため(3.5.1を参照)、または継承型自体の宣言によって隠ぺいされるため(3.7.1.2を参照)、継承元のメンバーに派生型からアクセスできない場合があります。

### 3.4.1 名前空間のメンバー

外側に名前空間を持たない名前空間と型は、グローバル名前空間のメンバーです。このようなメンバーは、グローバル宣言空間で宣言されている名前に直接対応しています。

名前空間の中で宣言されている名前空間と型は、その名前空間のメンバーです。このようなメンバーは、その名前空間の宣言空間で宣言されている名前に直接対応しています。

名前空間には、アクセス制限はありません。名前空間をプライベート、プロテクト、または内部として宣言することはできず、名前空間の名前は常にパブリックにアクセスできます。

### 3.4.2 構造体のメンバー

構造体のメンバーは、構造体で宣言されるメンバーと、構造体の直接基底クラス `System.ValueType` および間接基底クラス `object` から継承されるメンバーです。

単純な型のメンバーは、単純な型をエイリアスとする構造型のメンバーに直接対応しています。

- `sbyte` のメンバーは `System.SByte` 構造体のメンバーです。
- `byte` のメンバーは `System.Byte` 構造体のメンバーです。
- `short` のメンバーは `System.Int16` 構造体のメンバーです。
- `ushort` のメンバーは `System.UInt16` 構造体のメンバーです。
- `int` のメンバーは `System.Int32` 構造体のメンバーです。

- `uint` のメンバーは `System.UInt32` 構造体のメンバーです。
- `long` のメンバーは `System.Int64` 構造体のメンバーです。
- `ulong` のメンバーは `System.UInt64` 構造体のメンバーです。
- `char` のメンバーは `System.Char` 構造体のメンバーです。
- `float` のメンバーは `System.Single` 構造体のメンバーです。
- `double` のメンバーは `System.Double` 構造体のメンバーです。
- `decimal` のメンバーは `System.Decimal` 構造体のメンバーです。
- `bool` のメンバーは `System.Boolean` 構造体のメンバーです。

### 3.4.3 列挙型メンバー

列挙型のメンバーは、列挙の中で宣言される定数、列挙型の直接基底クラス `System.Enum` および間接基底クラス `System.ValueType` から継承されるメンバー、および `object` です。

### 3.4.4 クラスのメンバー

クラスのメンバーは、そのクラスの中で宣言されているメンバー、および基底クラスから継承されたメンバーです(ただし、`object` クラスには基底クラスがないので例外です)。定数、フィールド、メソッド、プロパティ、イベント、インデクサー、演算子、型などのメンバーは基底クラスから継承されますが、基底クラスのインスタンス コンストラクター、デストラクター、および静的コンストラクターは継承されません。基底クラスのメンバーは、アクセシビリティに関係なく継承されます。

クラスの宣言には、定数、フィールド、メソッド、プロパティ、イベント、インデクサー、演算子、インスタンス コンストラクター、デストラクター、静的コンストラクター、および型の宣言を含めることができます。

`object` および `string` のメンバーは、エイリアスとして表現しているクラス型のメンバーに直接対応しています。

- `object` のメンバーは `System.Object` クラスのメンバーです。
- `string` のメンバーは `System.String` クラスのメンバーです。

### 3.4.5 インターフェイスのメンバー

インターフェイスのメンバーは、そのインターフェイスで宣言されているメンバーと、そのインターフェイスのすべての基本インターフェイスで宣言されているメンバーです。`object` クラスのメンバーは、厳密にはどのインターフェイスのメンバーでもありません(13.2 を参照)。ただし、`object` クラスのメンバーは、インターフェイス型のメンバー検索を通じて利用できます(7.4 を参照)。

### 3.4.6 配列のメンバー

配列のメンバーは、`System.Array` クラスから継承したメンバーです。

### 3.4.7 デリゲートのメンバー

デリゲートのメンバーは、`System.Delegate` クラスから継承したメンバーです。

## 3.5 メンバー アクセス

メンバーの宣言によって、メンバーへのアクセスを制御できます。メンバーのアクセシビリティは、そのメンバーに宣言されたアクセシビリティ (3.5.1 を参照)、および、そのメンバーを直接包含する型がある場合は包含する型のアクセシビリティとの組み合わせによって決定されます。

特定のメンバーへのアクセスが許可されている場合、そのメンバーを **アクセス可能な** メンバーと呼びます。反対に、特定のメンバーへのアクセスが許可されていない場合、そのメンバーを **アクセス不可** のメンバーと呼びます。アクセスを行うコードの記述位置がメンバーのアクセシビリティ ドメイン (3.5.2 を参照) に含まれている場合、メンバーへのアクセスが許可されます。

### 3.5.1 声明されたアクセシビリティ

メンバーに **宣言される** アクセシビリティは、次のいずれかです。

- パブリック。メンバーの宣言で **public** 修飾子が指定されている場合のアクセシビリティです。つまり、**public** のメンバーに対しては、"アクセスは制限されません"。
- プロテクト。メンバーの宣言で **protected** 修飾子が指定されている場合のアクセシビリティです。つまり、**protected** のメンバーには、"それを含んでいるクラス、またはそれを含んでいるクラスから派生した型だけがアクセスできます"。
- 内部。メンバーの宣言で **internal** 修飾子が指定されている場合のアクセシビリティです。つまり、**internal** のメンバーには、"同じプログラムだけがアクセスできます"。
- プロテクト内部(つまり、プロテクトまたは内部)。メンバーの宣言で **protected** 修飾子と **internal** 修飾子が指定されている場合のアクセシビリティです。つまり、**protected internal** のメンバーには、"同じプログラム、またはメンバーを包含するクラスから派生された型だけがアクセスできます"。
- プライベート。メンバーの宣言で **private** 修飾子が指定されている場合のアクセシビリティです。つまり、**private** のメンバーには、"それを含んでいる型だけがアクセスできます"。

メンバーの宣言が行われるコンテキストに応じて、宣言されたアクセシビリティを持つ特定の型のみがアクセスできます。さらに、メンバーの宣言にアクセス修飾子がまったく含まれていない場合は、宣言が行われるコンテキストに基づいて、既定の宣言されたアクセシビリティが決まります。

- 名前空間の暗黙の宣言されたアクセシビリティは **public** です。名前空間の宣言に対しては、アクセス修飾子を指定できません。
- コンパイル単位または名前空間で宣言された型には、宣言されたアクセシビリティとして **public** または **internal** を指定でき、既定では **internal** が設定されます。
- クラスのメンバーには、宣言されたアクセシビリティとして 5 種類のどれでも指定でき、既定では **private** が設定されます。クラスのメンバーとして宣言された型には宣言されたアクセシビリティとして 5 種類のどれでも指定できるのに対し、名前空間のメンバーとして宣言された型に対して指定できるのは **public** または **internal** だけであることに注意してください。
- 構造体のメンバーには、宣言されたアクセシビリティとして **public**、**internal**、または **private** を指定できます。構造体は暗黙的にシールされるので、既定では **private** に設定されます。構造体に導入される構造体のメンバー(つまり、構造体によって継承されないメンバー)には、宣言されたアクセシビリティとして **protected** または **protected internal** を指定できません。構造体のメンバーとして宣言された型には宣言されたアクセシビリティとして **public**、

`internal`、または `private` を指定できるのに対し、名前空間のメンバーとして宣言された型に対して指定できるのは `public` または `internal` だけであることに注意してください。

- インターフェイスのメンバーに対して暗黙で宣言されるアクセシビリティは `public` です。インターフェイスのメンバーの宣言に対しては、アクセス修飾子を指定できません。
- 列挙型のメンバーに対して暗黙で宣言されるアクセシビリティは `public` です。列挙型のメンバーの宣言に対しては、アクセス修飾子を指定できません。

### 3.5.2 アクセシビリティ ドメイン

メンバーのアクセシビリティ ドメインとは、そのメンバーへのアクセスが許可された、プログラムテキストの(分離していることもある)セクションです。メンバーのアクセシビリティ ドメインを定義するために、型の中で宣言されていないメンバーをトップレベルにあると言い、別の型の中で宣言されているメンバーを入れ子になっていると言います。さらに、プログラムのプログラム テキストは、プログラムのすべてのソース ファイルに含まれるすべてのプログラム テキストとして定義され、型のプログラム テキストは、その型(型に入れ子になった型も含む)の *type-declaration* に含まれるすべてのプログラム テキストとして定義されます。

定義済みの型 (`object`、`int`、`double` など) のアクセシビリティ ドメインには制限はありません。

プログラム `P` で宣言されているトップレベルの非バインド型 `T` (4.4.3 を参照) のアクセシビリティ ドメインは、次のように定義されます。

- `T` に宣言されたアクセシビリティが `public` の場合、`T` のアクセシビリティ ドメインは、`P` のプログラム テキストと、`P` を参照しているすべてのプログラムです。
- `T` に対して宣言されているアクセシビリティが `internal` の場合、`T` のアクセシビリティ ドメインは `P` のプログラム テキストになります。

以上の定義から、トップレベルの非バインド型のアクセシビリティ ドメインは常に、少なくとも、その型が宣言されているプログラムのプログラム テキストです。

構築された型 `T<A1, ..., AN>` のアクセシビリティ ドメインは、非バインドジェネリック型 `T` のアクセシビリティ ドメインと型引数 `A1, ..., AN` のアクセシビリティ ドメインの積集合です。

プログラム `P` の内部にある型 `T` の中で宣言されている入れ子になったメンバー `M` のアクセシビリティ ドメインは、次のように定義されます。ここでは、`M` 自体も型である可能性があることに注意してください。

- `M` に対して宣言されたアクセシビリティが `public` の場合、`M` のアクセシビリティ ドメインは `T` のアクセシビリティ ドメインになります。
- `M` に宣言されたアクセシビリティが `protected internal` の場合は、`P` のプログラム テキストと、`P` の外部で宣言されている `T` から派生したすべての型のプログラム テキストの和集合が `D` になります。この場合、`M` のアクセシビリティ ドメインは、`T` のアクセシビリティ ドメインと `D` の積集合です。
- `M` に宣言されたアクセシビリティが `protected` の場合は、`T` のプログラム テキストと、`T` から派生したすべての型のプログラム テキストの和集合が `D` になります。この場合、`M` のアクセシビリティ ドメインは、`T` のアクセシビリティ ドメインと `D` の積集合です。
- `M` に対して宣言されているアクセシビリティが `internal` の場合、`M` のアクセシビリティ ドメインは、`T` のアクセシビリティ ドメインと `P` のプログラム テキストとの積集合になります。

- M に対して宣言されているアクセシビリティが **private** の場合、M のアクセシビリティ ドメインは T のプログラム テキストになります。

以上の定義から、入れ子になったメンバーのアクセシビリティ ドメインは、常に少なくとも、そのメンバーが宣言されている型のプログラム テキストです。さらに、メンバーのアクセシビリティ ドメインは、そのメンバーが宣言されている型のアクセシビリティ ドメインより広くなることはありません。

まとめると、型またはメンバー M がアクセスされるときは、次の手順で評価されて、アクセスが許可されます。

- 最初に、M が、コンパイル単位や名前空間ではなく、型の中で宣言されている場合、その型がアクセス可能でないとコンパイル エラーになります。
- 次に、M が **public** の場合、アクセスは許可されます。
- M が **protected internal** の場合は、M が宣言されているプログラムの中でアクセスされるか、または M が宣言されているクラスから派生したクラスの中で、派生クラス型 (3.5.3 を参照) を通じてアクセスされると、アクセスは許可されます。
- M が **protected** の場合は、M が宣言されているクラスの中でアクセスされるか、または M が宣言されているクラスから派生したクラスの中で、派生クラス型 (3.5.3 を参照) を通じてアクセスされると、アクセスは許可されます。
- M が **internal** の場合は、M が宣言されているプログラムの中でアクセスされると、アクセスは許可されます。
- M が **private** の場合は、M が宣言されている型の中でアクセスされると、アクセスは許可されません。
- 以上のいずれにも該当しない場合、型またはメンバーはアクセス可能ではないため、コンパイル エラーになります。

次に例を示します。

```
public class A
{
    public static int X;
    internal static int Y;
    private static int Z;
}
internal class B
{
    public static int X;
    internal static int Y;
    private static int Z;
    public class C
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }
}
```

```

private class D
{
    public static int X;
    internal static int Y;
    private static int Z;
}

```

この例のクラスおよびメンバーのアクセシビリティ ドメインは、次のとおりです。

- A および A.X のアクセシビリティ ドメインは無制限です。
- A.Y、B、B.X、B.Y、B.C、B.C.X、および B.C.Y のアクセシビリティ ドメインは、これらを含むプログラムのプログラム テキストです。
- A.Z のアクセシビリティ ドメインは、A のプログラム テキストです。
- B.Z および B.D のアクセシビリティ ドメインは、B のプログラム テキストであり、B.C および B.D のプログラム テキストを含みます。
- B.C.Z のアクセシビリティ ドメインは、B.C のプログラム テキストです。
- B.D.X および B.D.Y のアクセシビリティ ドメインは、B のプログラム テキストであり、B.C および B.D のプログラム テキストを含みます。
- B.D.Z のアクセシビリティ ドメインは、B.D のプログラム テキストです。

例で示されているように、メンバーのアクセシビリティ ドメインは、それを含む型のアクセシビリティ ドメインより広くならることはできません。たとえば、X のすべてのメンバーの宣言されたアクセシビリティはパブリックですが、A.X を除くすべてのメンバーのアクセシビリティ ドメインは、それを含む型によって制限されます。

3.4 で説明されているように、インスタンス コンストラクター、デストラクター、および静的コンストラクターを除く基底クラスのすべてのメンバーは、派生型によって継承されます。これには、基底クラスのプライベート メンバーも含まれます。ただし、プライベート メンバーのアクセシビリティ ドメインには、メンバーが宣言されている型のプログラム テキストだけが含まれます。次に例を示します。

```

class A
{
    int x;
    static void F(B b) {
        b.x = 1;      // ok
    }
}
class B: A
{
    static void F(B b) {
        b.x = 1;      // Error, x not accessible
    }
}

```

B クラスは、A クラスからプライベート メンバー x を継承します。このメンバーはプライベートであるため、A の *class-body* の中でしかアクセスできません。したがって、b.x へのアクセスは、A.F メソッドでは成功しますが、B.F メソッドでは失敗します。

### 3.5.3 インスタンス メンバーへのプロテクト アクセス

`protected` のインスタンス メンバーに、そのメンバーが宣言されているクラスのプログラム テキストの外側からアクセスする場合や、`protected internal` のインスタンス メンバーに、そのメンバーが宣言されているプログラムのプログラム テキストの外側からアクセスする場合は、そのメンバーが宣言されているクラスから派生したクラス宣言内でアクセスする必要があります。また、その派生クラス型のインスタンスか、その派生クラス型から構築されたクラス型を通じてアクセスする必要があります。この制約により、ある派生クラスが他の派生クラスの保護されたメンバーにアクセスできなくなります。これは、そのメンバーが同じ基底クラスから継承されている場合も同様です。

`B` はプロテクト インスタンス メンバー `M` を宣言する基底クラスであり、`D` は `B` から派生されるクラスであるとします。`D` の *class-body* の中では、次のいずれかの形式で `M` にアクセスできます。

- `M` という形式の、修飾されていない *type-name* または *primary-expression*。
- `E.M` という形式の *primary-expression*。ただし、`E` の型が `T` または `T` から派生したクラスの場合で、`T` はクラス型 `D`、または `D` から構築したクラス型。
- `base.M` という形式の *primary-expression*。

これらのアクセス形式に加えて、派生クラスは、基底クラスのプロテクト インスタンス コンストラクターに *constructor-initializer* の中でアクセスできます(10.11.1 を参照)。

次に例を示します。

```
public class A
{
    protected int x;
    static void F(A a, B b) {
        a.x = 1;      // Ok
        b.x = 1;      // Ok
    }
}
public class B: A
{
    static void F(A a, B b) {
        a.x = 1;      // Error, must access through instance of B
        b.x = 1;      // Ok
    }
}
```

`A` の場合、`A` と `B` のいずれのインスタンスを通じても `x` にアクセスできます。これは、いずれの場合も、`A` のインスタンスまたは `A` の派生クラスを通じてアクセスが行われるためです。一方、`B` の場合は、`A` のインスタンスを通じて `x` にアクセスすることはできません。これは、`A` が `B` から派生していないためです。

次に例を示します。

```
class C<T>
{
    protected T x;
}
```

```

class D<T>: C<T>
{
    static void F()
    {
        D<T> dt = new D<T>();
        D<int> di = new D<int>();
        D<string> ds = new D<string>();
        dt.x = default(T);
        di.x = 123;
        ds.x = "test";
    }
}

```

`x`への3つの代入は、ジェネリック型から構築されたクラス型のインスタンスを通じて実行されるため、許可されます。

### 3.5.4 アクセシビリティの制約

C#言語の構造において、いくつかの型は、メンバーや他の型と同程度にアクセス可能である必要があります。型 `T` のアクセシビリティ ドメインが型 `M` のアクセシビリティ ドメインのスーパーセットになっている場合、`T` はメンバーまたは `M` と少なくとも同程度にアクセス可能であるといえます。つまり、`M` にアクセスできるすべてのコンテキストで `T` にアクセスできる場合、`T` は `M` と少なくとも同程度にアクセス可能です。

アクセシビリティには、次のような制約があります。

- クラスの型の直接基底クラスは、少なくとも、クラスの型自体と同程度にアクセス可能である必要があります。
- インターフェイスの型の明示的な基本インターフェイスは、少なくとも、インターフェイスの型自体と同程度にアクセス可能である必要があります。
- デリゲート型の戻り値の型およびパラメーターの型は、少なくとも、デリゲート型自体と同程度にアクセス可能である必要があります。
- 定数の型は、少なくとも定数自体と同程度にアクセス可能である必要があります。
- フィールドの型は、少なくともフィールド自体と同程度にアクセス可能である必要があります。
- メソッドの戻り値の型およびパラメーターの型は、少なくとも、メソッド自体と同程度にアクセス可能である必要があります。
- プロパティの型は、少なくともプロパティ自体と同程度にアクセス可能である必要があります。
- イベントの型は、少なくともイベント自体と同程度にアクセス可能である必要があります。
- インデクサーの型およびパラメーターの型は、少なくとも、インデクサー自体と同程度にアクセス可能である必要があります。
- 演算子の戻り値の型およびパラメーターの型は、少なくとも、演算子自体と同程度にアクセス可能である必要があります。
- インスタンス コンストラクターのパラメーターの型は、少なくともインスタンス コンストラクター自体と同程度にアクセス可能である必要があります。

次に例を示します。

```

class A {...}
public class B: A {...}

```

A のアクセシビリティは B と同程度未満であるため、B クラスはコンパイルエラーになります。

次も同様の例です。

```
class A {...}
public class B
{
    A F() {...}
    internal A G() {...}
    public A H() {...}
}
```

戻り値の型 A のアクセシビリティは B の H メソッドと同程度未満であるため、H メソッドはコンパイルエラーになります。

### 3.6 シグネチャとオーバーロード

メソッド、インスタンス コンストラクター、インデクサー、および演算子は、それぞれのシグネチャによって特徴付けられます。

- メソッドのシグネチャは、メソッドの名前、型パラメーターの数、各仮パラメーターの型とパラメーター受け渡しモード(値、参照、または出力)で構成されます。これらは左から右の順序で参照されます。このため、仮パラメーターの型で使用されるメソッドの型パラメーターは、名前によって識別されるのではなく、メソッドの型引数リスト内での順番によって識別されます。メソッドのシグネチャからは、戻り値の型、右端のパラメーターに対して指定される可能性のある `params` 修飾子、およびオプションの型パラメーター制約が明確に除外されています。
- インスタンス コンストラクターのシグネチャは、左から右に並んだ順序で参照される、各仮パラメーターの型とパラメーター受け渡しモード(値、参照、または出力)で構成されます。インスタンス コンストラクターのシグネチャからは、右端のパラメーターに対して指定される可能性のある `params` 修飾子が明確に除外されています。
- インデクサーのシグネチャは、左から右に並んだ順序で参照される、各仮パラメーターの型で構成されます。インデクサーのシグネチャからは、要素の型、および右端のパラメーターに対して指定される可能性のある `params` 修飾子が明確に除外されています。
- 演算子のシグネチャは、演算子の名前と、左から右に並んだ順序で参照される、各仮パラメーターの型で構成されます。演算子のシグネチャからは、結果の型が明確に除外されています。

メンバーの種類が同じ 2 つのシグネチャは、名前、型パラメーターの数、パラメーター受け渡しモードが同じで、かつ対応する型の間に恒等変換が存在する場合に、同じシグネチャと見なされます(6.1.1 を参照)。

シグネチャは、クラス、構造体、およびインターフェイスのメンバーのオーバーロードを可能にする機構です。

- メソッドのオーバーロードを利用すると、クラス、構造体、またはインターフェイスで、名前は同じでもシグネチャの異なる複数のメソッドを宣言できます。
- インスタンス コンストラクターのオーバーロードを利用すると、クラスまたは構造体で、シグネチャの異なる複数のインスタンス コンストラクターを宣言できます。
- インデクサーのオーバーロードを利用すると、クラス、構造体、またはインターフェイスで、シグネチャの異なる複数のインデクサーを宣言できます。

- 演算子のオーバーロードを利用すると、クラスまたは構造体で、名前は同じでもシグネチャの異なる複数の演算子を宣言できます。

**out** と **ref** のパラメーター修飾子はシグネチャの一部と見なされますが、1つの型で宣言される複数のメンバーは、シグネチャの **ref** と **out** だけが異なる場合、区別されません。2つのメンバーが同じ型で宣言され、すべてのパラメーターの **out** 修飾子を **ref** 修飾子に変更すると2つのメソッドのシグネチャが同じになる場合、コンパイルエラーが発生します。その他の目的(隠ぺいやオーバーライドなど)でシグネチャを比較する場合は、**ref** と **out** はシグネチャの一部と見なされ、互いに一致することはありません。この制限の目的は、**ref** と **out** だけが異なるメソッドを定義する方法が用意されていない共通言語基盤(CLI: Common Language Infrastructure)で実行するために、C# プログラムを簡単に変換できるようにすることです。

オーバーロードされたメソッドの宣言とそのシグネチャの例を次に示します。

```
interface ITest
{
    void F();                                // F()
    void F(int x);                            // F(int)
    void F(ref int x);                        // F(ref int)
    void F(out int x);                         // F(out int)      error
    void F(int x, int y);                     // F(int, int)
    int F(string s);                          // F(string)
    int F(int x);                            // F(int)          error
    void F(string[] a);                       // F(string[])
    void F(params string[] a);                // F(string[])     error
}
```

パラメーター修飾子の **ref** と **out**(10.6.1 を参照)は、シグネチャの一部です。したがって、**F(int)** と **F(ref int)** は、異なるシグネチャです。ただし **F(ref int)** と **F(out int)** は、シグネチャの違いが **ref** と **out** だけであるため、同じインターフェイス内で宣言できません。また、戻り値の型と **params** 修飾子はシグネチャに含まれません。したがって、戻り値の型だけに基づいて、または **params** 修飾子の指定の有無だけに基づいて、オーバーロードを行うことはできません。そのため、前に示した **F(int)** メソッドと **F(params string[])** メソッドの宣言は、コンパイル時のエラーになります。

### 3.7 スコープ

名前のスコープとは、名前を修飾せずにその名前で宣言されているエンティティを参照できる、プログラムテキストの範囲を指します。スコープは "入れ子" にでき、内部スコープで外部スコープの名前の意味を再宣言できます(ただし、3.3 で課せられた、入れ子になったブロック内では外側のブロック内のローカル変数と同じ名前を持つローカル変数を宣言できないという制限は維持されます)。スコープを入れ子にした場合、外側のスコープの名前は、内側のスコープによってカバーされるプログラムテキストの範囲内では隠ぺいされます。このため、外側の名前にアクセスするには、名前を修飾する必要があります。

- 外側に *namespace-declaration* のない *namespace-member-declaration*(9.5 を参照)によって宣言された名前空間メンバーのスコープは、プログラムテキスト全体になります。

- 完全修飾名が N である *namespace-declaration* の中で *namespace-member-declaration* によって宣言される名前空間メンバーのスコープは、完全修飾名が N であるか、先頭が N でその後にピリオド(.) が付く、すべての *namespace-declaration* の *namespace-body* になります。
- extern-alias-directive* によって定義される名前のスコープは、すぐ外側のコンパイル単位または名前空間本体の *using-directives*、*global-attributes*、および *namespace-member-declarations* までです。*extern-alias-directive* によって、基になる宣言空間に新しいメンバーが追加されることはありません。つまり、*extern-alias-directive* は波及性がなく、*extern-alias-directive* が記述されているコンパイル単位または名前空間の本体の中だけで有効です。
- using-directive* (9.4 を参照) を使って定義またはインポートされた名前のスコープは、*using-directive* が記述されている *compilation-unit* または *namespace-body* の *namespace-member-declaration* 全体になります。*using-directive* は、特定の *compilation-unit* または *namespace-body* の中で利用できる 0 個以上の名前空間または型名を作成できますが、基になる宣言空間に新規メンバーを追加することはありません。つまり、*using-directive* は波及性がなく、*using-directive* が記述されている *compilation-unit* や *namespace-body* の中だけで有効です。
- class-declaration* (10.1 を参照) で *type-parameter-list* によって宣言される型パラメーターのスコープは、その *class-declaration* の *class-base*、*type-parameter-constraints-clauses*、および *class-body* です。
- struct-declaration* (11.1 を参照) で *type-parameter-list* によって宣言される型パラメーターのスコープは、その *struct-declaration* の *struct-interfaces*、*type-parameter-constraints-clauses*、および *struct-body* です。
- interface-declaration* (13.1 を参照) で *type-parameter-list* によって宣言される型パラメーターのスコープは、その *interface-declaration* の *interface-base*、*type-parameter-constraints-clauses*、および *interface-body* です。
- delegate-declaration* (15.1 を参照) で *type-parameter-list* によって宣言される型パラメーターのスコープは、その *delegate-declaration* の *return-type*、*formal-parameter-list*、および *type-parameter-constraints-clauses* です。
- class-member-declaration* (10.1.6 を参照) で宣言されるメンバーのスコープは、宣言が行われている *class-body* になります。さらに、クラスメンバーのスコープは、メンバーのアクセシビリティ メイン (3.5.2 を参照) に含まれる派生クラスの *class-body* にまで及びます。
- struct-member-declaration* (11.2 を参照) で宣言されるメンバーのスコープは、宣言が行われている *struct-body* になります。
- enum-member-declaration* (14.3 を参照) で宣言されるメンバーのスコープは、宣言が行われている *enum-body* になります。
- method-declaration* (10.6 を参照) で宣言されるパラメーターのスコープは、その *method-declaration* の *method-body* になります。
- indexer-declaration* (10.9 を参照) で宣言されるパラメーターのスコープは、その *indexer-declaration* の *accessor-declaration* になります。
- operator-declaration* (10.10 を参照) で宣言されるパラメーターのスコープは、その *operator-declaration* の *block* になります。

- *constructor-declaration* (10.11 を参照) で宣言されるパラメーターのスコープは、その *constructor-declaration* の *constructor-initializer* と *block* になります。
- *lambda-expression* (7.15 を参照) で宣言されるパラメーターのスコープは、その *lambda-expression* の *lambda-expression-body* になります。
- *anonymous-method-expression* (7.15 を参照) で宣言されるパラメーターのスコープは、その *anonymous-method-expression* の *block* になります。
- *labeled-statement* (8.4 を参照) で宣言されるラベルのスコープは、宣言が行われている *block* になります。
- *local-variable-declaration* (8.5.1 を参照) で宣言されるローカル変数のスコープは、宣言が行われているブロックになります。
- *switch* ステートメント (8.7.2 を参照) の *switch-block* で宣言されるローカル変数のスコープは、その *switch-block* になります。
- *for* ステートメント (8.8.3 を参照) の *for-initializer* で宣言されるローカル変数のスコープは、その *for* ステートメントの *for-initializer*、*for-condition*、*for-iterator*、および含まれている *statement* になります。
- *foreach-statement*、*using-statement*、*lock-statement*、または *query-expression* の一部として宣言される変数のスコープは、指定した構造の展開によって決まります。

名前空間、クラス、構造体、または列挙型のメンバーのスコープでは、メンバーの宣言より前の記述位置でそのメンバーを参照できます。次に例を示します。

```
class A
{
    void F()
    {
        i = 1;
    }
    int i = 0;
}
```

この例では、*i* の宣言より前で *F* が *i* を参照しても問題ありません。

ローカル変数のスコープでは、ローカル変数の *local-variable-declarator* より前の記述位置でローカル変数を参照するとコンパイルエラーになります。次に例を示します。

```
class A
{
    int i = 0;
    void F()
    {
        i = 1; // Error, use precedes declaration
        int i;
        i = 2;
    }
    void G()
    {
        int j = (j = 1); // Valid
    }
}
```

```
void H() {
    int a = 1, b = ++a; // valid
}
```

前の例の F メソッドで、i への最初の代入は、厳密には外側のスコープで宣言されているフィールドを参照していません。逆に、ローカル変数を参照しているので、記述位置が変数の宣言より前にあることになり、コンパイルエラーになります。G メソッドでは、j の宣言に対する初期化子での j の使用は、*local-variable-declarator* より前ではないので有効です。H メソッドでは、同じ *local-variable-declarator* の中に、後の *local-variable-declarator* は、前にある *local-variable-declarator* によって宣言されているローカル変数を正しく参照しています。

ローカル変数に対するスコープの規則は、式のコンテキストで使われる名前の意味がブロック内で常に同じであることを保証するために設けられています。ローカル変数のスコープが、宣言からブロックの終了の範囲だけに及ぶようになっていると、上の例では、最初の代入はインスタンス変数に対して行われ、2 番目の代入はローカル変数に対して行われることになります。このため、後でブロックのステートメントの順序を変更すると、コンパイルエラーになる可能性があります。

ブロック内の名前の意味は、名前が使用されるコンテキストによって異なる場合があります。次に例を示します。

```
using System;
class A {}
class Test
{
    static void Main() {
        string A = "hello, world";
        string s = A; // expression context
        Type t = typeof(A); // type context
        Console.WriteLine(s); // writes "hello, world"
        Console.WriteLine(t); // writes "A"
    }
}
```

この例では、A という名前は、式のコンテキストではローカル変数 A を参照するために使われていて、型のコンテキストでは A クラスを参照するために使われています。

### 3.7.1 名前の隠ぺい

一般に、エンティティのスコープは、エンティティの宣言空間より多くのプログラムテキストを含んでいます。特に、エンティティのスコープには、同じ名前のエンティティを含む新しい宣言空間を導入する宣言があります。このような宣言により、元のエンティティは隠ぺいされることになります。逆に、隠ぺいされていないエンティティは可視であると言えます。

入れ子によってスコープがオーバーラップしたり、継承によってスコープがオーバーラップしたりすると、名前の隠ぺいが発生します。この 2 種類の隠ぺいの特徴については、次のセクションで説明します。

#### 3.7.1.1 入れ子による隠ぺい

入れ子による名前の隠ぺいは、名前空間または名前空間内の型を入れ子にした場合、クラスまたは構造体の中で型を入れ子にした場合、およびパラメーターやローカル変数を宣言した場合に、発生する可能性があります。

次に例を示します。

```
class A
{
    int i = 0;
    void F() {
        int i = 1;
    }
    void G() {
        i = 1;
    }
}
```

この例の `F` メソッドでは、インスタンス変数 `i` がローカル変数 `i` によって隠ぺいされていますが、`G` メソッドでは、`i` はインスタンス変数を参照しています。

外側のスコープの名前が内側のスコープの名前によって隠ぺいされる場合は、その名前がオーバーロードされたものもすべて隠ぺいされます。次に例を示します。

```
class Outer
{
    static void F(int i) {}
    static void F(string s) {}
    class Inner
    {
        void G() {
            F(1);           // Invokes Outer.Inner.F
            F("Hello");    // Error
        }
        static void F(long l) {}
    }
}
```

この例では、外部での `F` の宣言はすべて内部での宣言によって隠ぺいされるため、`F(1)` を呼び出すと、`Inner` で宣言されている `F` が呼び出されます。同じ理由から、`F("Hello")` の呼び出しはコンパイルエラーになります。

### 3.7.1.2 繙承による隠ぺい

クラスまたは構造体が基底クラスから継承した名前を宣言し直すと、継承による名前の隠ぺいが発生します。継承による名前の隠ぺいは、次のいずれかの形式で行われます。

- 定数、フィールド、プロパティ、イベント、型をクラスまたは構造体に導入すると、同じ名前を持つすべての基底クラス メンバーが隠ぺいされます。
- クラスまたは構造体にメソッドを導入すると、メソッド以外で同じ名前を持つすべての基底クラス メンバー、および同じシグネチャを持つ基底クラスのメソッドがすべて隠ぺいされます (3.6 を参照)。
- インデクサーをクラスまたは構造体で使用すると、同じシグネチャを持つすべての基底クラス インデクサーが隠ぺいされます (3.6 を参照)。

演算子の宣言に関する規則 (10.10 を参照) により、派生クラスは、基底クラスの演算子と同じシグネチャを持つ演算子を宣言することはできません。したがって、演算子が他の演算子を隠ぺいすることはありません。

外部スコープからの名前の隠ぺいとは異なり、継承したスコープにあるアクセス可能な名前を隠ぺいすると、警告が報告されます。次に例を示します。

```
class Base
{
    public void F() {}
}
class Derived: Base
{
    public void F() {}      // Warning, hiding an inherited name
}
```

`Derived` の `F` の宣言において警告が発生します。継承された名前の隠ぺいはエラーではありません。これをエラーにすると、基底クラスを独立して更新できなくなる可能性があります。たとえば、`Base` クラスを更新するときに、旧バージョンには存在しなかった `F` メソッドを導入すると、前の例のような状況が発生する場合があります。上の状況がエラーであるとすると、各バージョンのクラスライブラリで基底クラスになんらかの変更を行った場合、派生クラスが無効になる可能性があります。

継承された名前を隠ぺいしても警告が報告されないようにするには、`new` 修飾子を使用します。

```
class Base
{
    public void F() {}
}
class Derived: Base
{
    new public void F() {}
}
```

`new` 修飾子は、`Derived` の `F` が "新しい" ものであり、継承されたメンバーの隠ぺいを実際に意図していることを示しています。

新しいメンバーを宣言すると、継承されたメンバーは新しいメンバーのスコープ内だけで隠ぺいされます。

```
class Base
{
    public static void F() {}
}
class Derived: Base
{
    new private static void F() {}  // Hides Base.F in Derived only
}
class MoreDerived: Derived
{
    static void G() { F(); }        // Invokes Base.F
}
```

上の例では、`Derived` における `F` の宣言は、`Base` から継承した `F` を隠ぺいしますが、`Derived` の新しい `F` のアクセスはプライベートであるため、そのスコープは `MoreDerived` まで及びません。したがって、`MoreDerived.G` での呼び出し `F()` は有効であり、`Base.F` を呼び出します。

## 3.8 名前空間と型の名前

C# プログラムのいくつかのコンテキストでは、*namespace-name* または *type-name* を指定する必要があります。

```

namespace-name:
  namespace-or-type-name

type-name:
  namespace-or-type-name

namespace-or-type-name:
  identifier type-argument-listopt
  namespace-or-type-name . identifier type-argument-listopt
  qualified-alias-member

```

*namespace-name* は、名前空間を参照する *namespace-or-type-name* です。*namespace-name* の *namespace-or-type-name* は、以下で説明する解決規則に従って名前空間を参照する必要があります。この規則に従わないと、コンパイルエラーが発生します。型引数 (4.4.1 を参照) は *namespace-name* 内に存在できません (型引数を持つのは型のみです)。

*type-name* は、型を参照する *namespace-or-type-name* です。*type-name* の *namespace-or-type-name* は、以下で説明する解決規則に従って型を参照する必要があります。この規則に従わないと、コンパイルエラーが発生します。

*namespace-or-type-name* が *qualified-alias-member* であれば、その意味は 9.7 で説明されているとおりです。それ以外の場合、*namespace-or-type-name* の形式は、次の 4 つの形式のいずれかです。

- *I*
- *I<A<sub>1</sub>, ..., A<sub>K</sub>>*
- *N.I*
- *N.I<A<sub>1</sub>, ..., A<sub>K</sub>>*

ここで、*I* は単一の識別子、*N* は *namespace-or-type-name*、*<A<sub>1</sub>, ..., A<sub>K</sub>>* はオプションの *type-argument-list* です。*type-argument-list* が指定されていない場合、*K* はゼロと見なされます。

*namespace-or-type-name* の意味は、次の規則に従って決定されます。

- *namespace-or-type-name* が形式 *I* または形式 *I<A<sub>1</sub>, ..., A<sub>K</sub>>*:
  - *K* がゼロであり、*namespace-or-type-name* がジェネリック メソッド宣言 (10.6 を参照) の中に存在し、その宣言に名前 *I* を持つ型パラメーター (10.1.3 を参照) が含まれる場合、*namespace-or-type-name* はその型パラメーターを参照します。
  - それ以外の場合で、型宣言内に *namespace-or-type-name* がある場合は、その型宣言のインスタンス型を開始点として、外側のクラスまたは構造体宣言 (存在する場合) を順番にたどりながら、それぞれのインスタンス型 *T* (10.3.1 を参照) について次の評価が行われます。
    - *K* がゼロで、*T* の宣言に名前 *I* を持つ型パラメーターが含まれる場合、*namespace-or-type-name* はその型パラメーターを参照します。
    - それ以外の場合で、型宣言の本体内に *namespace-or-type-name* があり、*T* またはその基本型に名前が *I* で *K* 個の型パラメーターを持つ入れ子になったアクセス可能型が含まれる場合、*namespace-or-type-name* は、指定された型引数で構築されたその型を参照します。このような型が複数ある場合は、派生型内で宣言された型が選択されます。*namespace-or-type-name* の意味を決定するときは、型でないメンバ (定数、フィールド、メソッド、プロパティ、インデクサ、演算子、インスタンス コンストラクタ、デストラクタ、および静的コンストラクタ)、および型パラメータの数が異なる型メンバは無視されます。

- 前の手順が成功しなかった場合は、各名前空間  $N$  について、*namespace-or-type-name* が存在する名前空間から開始して、グローバル名前空間で終了するまで、それぞれの外側にある名前空間 (存在する場合) に対して順番に、エンティティが見つかるまで次の手順が評価されます。
  - $K$  がゼロで、 $I$  が  $N$  における名前空間の名前の場合は、次の規則に従います。
    - *namespace-or-type-name* の存在する場所が  $N$  の名前空間宣言で囲まれており、その名前空間宣言に含まれる *extern-alias-directive* または *using-alias-directive* によって名前  $I$  が名前空間または型と関連付けられている場合、この *namespace-or-type-name* はあいまいであり、コンパイル エラーになります。
    - それ以外の場合、*namespace-or-type-name* は  $N$  における  $I$  という名前の名前空間を参照します。
  - それ以外の場合で、名前が  $I$  で  $K$  個の型パラメーターを持つアクセス可能な型が  $N$  に含まれている場合は、次の規則に従います。
    - $K$  がゼロで、*namespace-or-type-name* の存在する場所が  $N$  の名前空間宣言で囲まれており、その名前空間宣言に含まれる *extern-alias-directive* または *using-alias-directive* によって名前  $I$  が名前空間または型と関連付けられている場合、この *namespace-or-type-name* はあいまいであり、コンパイル エラーになります。
    - それ以外の場合、*namespace-or-type-name* は指定された型引数によって構築された型を参照します。
  - それ以外の場合で、*namespace-or-type-name* の存在する場所が  $N$  の名前空間宣言に囲まれている場合は、次の規則に従います。
    - $K$  がゼロで、名前空間の宣言に *extern-alias-directive* または *using-alias-directive* が含まれており、それによって名前  $I$  とインポートされた名前空間または型が関連付けられている場合、*namespace-or-type-name* はその名前空間または型を参照します。
    - それ以外の場合で、名前空間の宣言の *using-namespace-directive* によってインポートされる名前空間に、名前が  $I$  で  $K$  個の型パラメーターを持つ型が 1 つだけ含まれている場合、*namespace-or-type-name* は指定された型引数で構築されたその型を参照します。
    - それ以外の場合で、名前空間の宣言の *using-namespace-directive* によってインポートされる名前空間に、名前が  $I$  で  $K$  個の型パラメーターを持つ型が 2 つ以上含まれている場合、*namespace-or-type-name* はあいまいであり、エラーになります。
  - 以上のいずれにも該当しない場合は、*namespace-or-type-name* は未定義で、コンパイル エラーになります。
- 以上のいずれにも該当しない場合は、*namespace-or-type-name* が形式  $N.I$  or of the form  $N.I<A_1, \dots, A_K>.N$  は最初に *namespace-or-type-name* として解決されます。  $N$  の解決が成功しない場合は、コンパイル エラーが発生します。 解決が成功した場合、 $N.I$  または  $N.I<A_1, \dots, A_K>$  は次のように解決されます。
  - $K$  がゼロで、 $N$  が名前空間を参照し、名前が  $I$  の入れ子になった名前空間が  $N$  に含まれる場合、*namespace-or-type-name* はその入れ子になった名前空間を参照します。

- それ以外の場合で、**N** が名前空間を参照し、名前が **I** で **K** 個の型パラメーターを持つアクセス可能な型が **N** に含まれている場合、*namespace-or-type-name* は指定された型引数で構築されたその型を参照します。
- それ以外の場合で、**N** が（多くの場合は構築された）クラスまたは構造型を参照し、名前が **I** で **K** 個の型パラメーターを持つ入れ子になったアクセス可能な型が **N**（または **N** の基底クラス）に含まれる場合、*namespace-or-type-name* は指定された型引数で構築されたその型を参照します。このような型が複数ある場合は、派生型内で宣言された型が選択されます。**N** の基底クラスの指定を解決する際に、**N.I** の意味が判別される場合、**N** の直接基底クラスはオブジェクトであると見なされます（10.1.4.1 を参照）。
- それ以外の場合は、**N.I** は無効な *namespace-or-type-name* であり、コンパイル エラーが発生します。

以下の場合にのみ、*namespace-or-type-name* は静的クラス（10.1.1.3 を参照）を参照できます。

- *namespace-or-type-name* が形式 **T.I** の *namespace-or-type-name* 内の **T** である場合
- または、*namespace-or-type-name* が形式 **typeof(T)** の *typeof-expression*（7.5.11 を参照）内の **T** である場合

### 3.8.1 完全修飾名

名前空間と型はすべて完全修飾名を持っています。完全修飾名は、特定の名前空間または型を一意に示します。**N** という名前空間または型の完全修飾名は、次の方法で決定されます。

- **N** がグローバル名前空間のメンバーである場合、そのメンバーの完全修飾名は **N** です。
- それ以外の場合、完全修飾名は **S.N** です。**S** は、**N** が宣言されている名前空間または型の完全修飾名です。

つまり、**N** の完全修飾名とは、**N** に至る筋道を示す、グローバル名前空間から始まる識別子の完全な階層的パスです。名前空間または型のメンバーはすべて固有の名前を持つ必要があるため、名前空間または型の完全修飾名は常に一意になります。

名前空間と型の宣言とそれに対応する完全修飾名の例を以下に示します。

```
class A {}           // A
namespace X          // X
{
    class B          // X.B
    {
        class C {}   // X.B.C
    }
    namespace Y      // X.Y
    {
        class D {}   // X.Y.D
    }
}
namespace X.Y        // X.Y
{
    class E {}       // X.Y.E
```

### 3.9 自動メモリ管理

C# では自動メモリ管理機能が採用されており、開発者は、オブジェクトが占有するメモリの割り当てと解放を手動で行う必要はありません。自動メモリ管理のポリシーは、ガベージコレクターによって実装されます。オブジェクトに対するメモリ管理のライフサイクルは次のとおりです。

1. オブジェクトが作成されると、メモリが割り当てられ、コンストラクターが実行され、オブジェクトが有効であると見なされます。
2. オブジェクトまたはその一部が、デストラクターを実行する方法を除き、どのような方法で実行を継続してもアクセスできない場合、そのオブジェクトは使用されていないと見なされ、破壊できる状態になります。C# コンパイラとガベージコレクターは、コードを解析して、将来使用される可能性のあるオブジェクトへの参照を判別できます。たとえば、スコープ内のローカル変数が、オブジェクトへの唯一の参照であるとします。しかし、そのローカル変数は、プロシージャの現在の実行ポイントから、どのような方法で実行を継続しても参照されることはありません。このようなオブジェクトを、ガベージコレクターは既に使用されなくなったオブジェクトとして処理できます。ただし、この処理は必須ではありません。
3. オブジェクトが破壊できる状態になると、そのオブジェクトに対するデストラクター(10.13 を参照)が存在する場合に、その後いずれかのタイミングでデストラクターが実行されます。通常の状況では、そのオブジェクトのデストラクターが実行されるのは1回だけですが、実装固有の API はこの動作がオーバーライドされることを許可することができます。
4. オブジェクトに対してデストラクターが実行された後は、オブジェクトまたはその一部が、デストラクターの実行も含め、どのような方法で実行を継続してもアクセスできない場合、そのオブジェクトはアクセス不可と見なされて、コレクションできる状態になります。
5. 最後に、オブジェクトがコレクションの対象になった後で、ガベージコレクターがそのオブジェクトに関連付けられたメモリを解放します。

ガベージコレクターは、オブジェクトの使用についての情報を保持します。この情報を使用して、新しく作成されたオブジェクトをメモリ内で配置する位置、オブジェクトを再配置する時期、オブジェクトが使用されなくなる時期やアクセスできなくなる時期の決定など、メモリ管理に関する決定を行います。

ガベージコレクターの存在を前提としている他の言語と同じように、C# も、ガベージコレクターが広範なメモリ管理ポリシーを実装できるように設計されています。たとえば、C# では、オブジェクトが破壊やコレクションの対象になっても、デストラクターの実行やコレクションをすぐには要求しません。また、デストラクターが特定の順序で実行されることや、特定のスレッドに対して実行されることも要求しません。

ガベージコレクターの動作は、`System.GC` クラスの静的メソッドを通じて、ある程度制御できます。このクラスは、コレクションの実行や、デストラクターの実行(または実行しないこと)などを要求するために使用できます。

ガベージコレクターでは、オブジェクトのコレクション時期やデストラクターの実行時期の決定をかなり自由に行うことができるため、準拠している実装でも、次のコードで示されているものとは結果が異なる場合があります。次のプログラムを参照してください。

```
using System;
```

```

class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }
}
class B
{
    object Ref;
    public B(object o) {
        Ref = o;
    }
    ~B() {
        Console.WriteLine("Destruct instance of B");
    }
}
class Test
{
    static void Main() {
        B b = new B(new A());
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}

```

このプログラムでは、**A** クラスのインスタンスと **B** クラスのインスタンスを作成しています。変数 **b** に値 **null** が代入されると、これ以降、ユーザーが記述したコードはオブジェクトにアクセスできなくなるため、オブジェクトはガベージコレクションできる状態になります。出力は次のいずれかになります。

```

Destruct instance of A
Destruct instance of B

```

または

```

Destruct instance of B
Destruct instance of A

```

オブジェクトのガベージコレクションが実行される順序については、C# 言語ではどのような制約も設けていません。

まれに、"破壊できる状態" と "コレクションできる状態" の違いが重要になる場合もあります。次に例を示します。

```

using System;
class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }
    public void F() {
        Console.WriteLine("A.F");
        Test.RefA = this;
    }
}
class B
{
    public A Ref;

```

```

~B() {
    Console.WriteLine("Destruct instance of B");
    Ref.F();
}
}

class Test
{
    public static A RefA;
    public static B RefB;

    static void Main() {
        RefB = new B();
        RefA = new A();
        RefB.Ref = RefA;
        RefB = null;
        RefA = null;

        // A and B now eligible for destruction
        GC.Collect();
        GC.WaitForPendingFinalizers();

        // B now eligible for collection, but A is not
        if (RefA != null)
            Console.WriteLine("RefA is not null");
    }
}

```

上のプログラムでは、ガベージコレクターが **B** の破壊前に **A** のデストラクターを実行すると、出力は次のようにになります。

```

Destruct instance of A
Destruct instance of B
A.F
RefA is not null

```

**A** のインスタンスは使用されておらず、**A** のデストラクターは実行されていますが、**A** のメソッド(この場合は **F**)が別のデストラクターから呼び出されることもあります。また、デストラクターを実行することにより、オブジェクトが再びメインのプロジェクトから使用できるようにすることもできます。この例では、**B** のデストラクターの実行により、それ以前は使用されていなかった**A** のインスタンスが、有効な参照 **Test.RefA** からアクセス可能になりました。**WaitForPendingFinalizers** を呼び出した後は、**B** のインスタンスはコレクションできる状態になりますが、**A** のインスタンスは、参照 **Test.RefA** があるためにコレクションできる状態にはなりません。

混乱や予期しない動作を避けるために、デストラクターでは、そのオブジェクト自身のフィールドに格納されたデータに対してだけクリーンアップを実行し、参照されているオブジェクトや静的フィールドに対しては処理を一切行わないようにすることをお勧めします。

デストラクターを使用する代わりに、クラスを使用して **System.IDisposable** インターフェイスを実装できます。これにより、そのオブジェクトのクライアントが、オブジェクトのリソースを解放する時期を決定できるようになります。これは通常、**using** ステートメント(8.13を参照)でオブジェクトにリソースとしてアクセスすることによって行います。

### 3.10 実行の順序

C# プログラムの実行時には、重大な実行ポイントで、各スレッドの実行によって発生した副作用が保持されます。“副作用”とは、**volatile** フィールドの読み取りまたは書き込み、非 **volatile** 変数への書き込み、外部リソースへの書き込み、および例外のスローです。副作用の順序を保持する必要がある重大な実行ポイントは、**volatile** フィールドへの参照(10.5.3を参照)、**lock** ステートメント(8.12を参

照)、およびスレッドの作成と終了です。実行環境では、C# プログラムの実行順序を自由に変更できますが、次の制限事項に従う必要があります。

- 実行スレッド内でデータの依存関係を保持する。つまり、各変数の値は、元のプログラムの順序でスレッドのすべてのステートメントを実行した場合と同じように処理される必要があります。
- 初期化の順序の規則を保持する (10.5.4 と 10.5.5 を参照)。
- volatile フィールドの読み取りと書き込みについて、副作用の順序を保持する (10.5.3 を参照)。式の値が使用されず、必要な副作用 (メソッドの呼び出しや volatile フィールドへのアクセスによって発生するすべての副作用を含む) が生成されないと推定できる場合は、その式の一部を実行環境で評価する必要はありません。非同期イベント (別のスレッドがスローした例外など) によってプログラムの実行が中断された場合は、本来なら見ることができる副作用を元のプログラムの実行順序で見ることができない可能性があります。

## 4. 型

C# 言語の型は、"値型" と "参照型" の 2 種類に分けられます。値型にも参照型にも、1 つ以上の "型パラメーター" を持つ "ジェネリック型" を使用できます。型パラメーターは値型と参照型の両方を指定できます。

```
type:  
  value-type  
  reference-type  
  type-parameter
```

第 3 の型であるポインターは、安全ではないコードでだけ使用できます。これは 18.2 で詳細に説明します。

値型と参照型の違いは、値型の変数にはデータが直接格納されるのに対し、参照型の変数にはデータへの "参照" が格納される点です。後者は、"オブジェクト" とも呼ばれます。参照型では 2 つの変数から同じオブジェクトを参照できるため、ある変数を操作することによって、他の変数が参照しているオブジェクトにも影響する可能性があります。値型の場合、各変数はデータの独自のコピーを保持しているため、ある変数を操作することによって他の変数にも影響することはありません。

C# の型システムは、すべての型の値をオブジェクトとして扱うことができるよう統一されています。C# の各型は、`object` クラス型から直接または間接に派生しており、`object` はすべての型の最終的な基底クラスです。参照型の値は、`object` 型として値を見ることで、簡単にオブジェクトとして扱うことができます。値型の値は、ボックス化とボックス化解除の操作を実行することで、オブジェクトとして扱うことができます(4.3 を参照)。

### 4.1 値型

値型は、構造型体と列挙型のいずれかです。C# には、"単純型" と呼ばれる定義済みの構造型体の集合が用意されています。単純型は、予約語によって識別されます。

```
value-type:  
  struct-type  
  enum-type  
  
struct-type:  
  type-name  
  simple-type  
  nullable-type  
  
simple-type:  
  numeric-type  
  bool  
  
numeric-type:  
  integral-type  
  floating-point-type  
  decimal
```

```

integral-type:
    sbyte
    byte
    short
    ushort
    int
    uint
    long
    ulong
    char

floating-point-type:
    float
    double

nullable-type:
    non-nullable-value-type ?

non-nullable-value-type:
    type

enum-type:
    type-name

```

参照型の変数とは異なり、値型の変数は `null` 許容型である場合にのみ `null` 値を取ることができます。`null` 非許容型の値型には必ず対応する `null` 許容型の値型が存在し、同じ値の集合に加えて値 `null` を示します。

値型の変数に値を代入すると、代入した値のコピーが作成されます。これは、参照型の変数への代入と異なる点で、参照型変数の場合は、参照はコピーされますが、参照によって示されるオブジェクトはコピーされません。

#### 4.1.1 System.ValueType 型

すべての値型は、暗黙的に `System.ValueType` クラスから継承します。また、このクラスは `Object` クラスから継承します。値型からはどのような型も派生することはできず、したがって、値型は暗黙でシールクラス(10.1.1.2を参照)になっています。

`System.ValueType` 自体は、`value-type` ではありません。`System.ValueType` は、すべての `value-type` の派生元である `class-type` です。

#### 4.1.2 既定のコンストラクター

すべての値型は、"既定のコンストラクター"と呼ばれる、パラメーターのないパブリックなインスタンスコンストラクターを暗黙に宣言します。既定のコンストラクターは、値型の"既定値"と呼ばれる、0で初期化されたインスタンスを返します。

- いずれの `simple-types` の場合も、既定値は、すべてゼロのビットパターンで生成される値です。
  - `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、および `ulong` では、既定値は 0 です。
  - `char` の既定値は '`\x0000`' です。
  - `float` の既定値は `0.0f` です。
  - `double` の既定値は `0.0d` です。

- `decimal` の既定値は `0.0m` です。
- `bool` の既定値は `false` です。
- `enum-type` の `E` に対する既定値は型 `E` に変換された `0` です。
- `struct-type` に対する既定値は、すべての値型フィールドにはそれぞれの型の既定値が設定されて、すべての参照型フィールドには `null` が設定された値です。
- `nullable-type` の既定値は、`HasValue` プロパティが `false` で `value` プロパティが未定義のインスタンスです。既定値は、`null` 許容型の "**null 値**" とも呼ばれます。

他のインスタンス コンストラクターと同様に、値型の既定のコンストラクターも `new` 演算子を使って呼び出されます。効率化のため、この要件では、実際に実装がコンストラクター呼び出しを生成することは意図されていません。次の例では、変数 `i` と `j` は共にゼロに初期化されています。

```
class A
{
    void F()
    {
        int i = 0;
        int j = new int();
    }
}
```

すべての値型にはパラメーターのないパブリックなインスタンス コンストラクターが暗黙で宣言されるため、構造型に対してパラメーターのないコンストラクターの明示的な宣言を含めることはできません。ただし、パラメーター付きのインスタンス コンストラクターを構造型で宣言することはできます(11.3.8 を参照)。

#### 4.1.3 構造型

構造型は、定数、フィールド、メソッド、プロパティ、インデクサー、演算子、インスタンス コンストラクター、静的コンストラクター、および入れ子になった型を宣言できる値型です。構造型の宣言については 11.1 で説明しています。

#### 4.1.4 単純型

C# には、"**単純型**" と呼ばれる定義済みの構造型の集合が用意されています。単純型は予約語を使って示されますが、これらの予約語は、`System` 名前空間で定義済みの構造型に対する单なるエイリアスです。予約語と構造型の関係は、次の表のとおりです。

予約語	エイリアスの元になっている型
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

単純型は構造体型に対するエイリアスなので、すべての単純型にはメンバーがあります。たとえば、`int`には、`System.Int32`で宣言されているメンバーと、`System.Object`から継承したメンバーがあり、次のステートメントを使用できます。

```
int i = int.MaxValue;           // System.Int32.MaxValue constant
string s = i.ToString();       // System.Int32.ToString() instance method
string t = 123.ToString();     // System.Int32.ToString() instance method
```

単純型は、他の構造体型と異なり、ある種の追加操作を行うことができます。

- 大部分の単純型では、"リテラル"を記述することで値を作成できます(2.4.4を参照)。たとえば、`123`は`int`型のリテラルであり、「`a`」は`char`型のリテラルです。C#では、一般に構造体型のリテラルについてはサポートされていません。また、他の構造体型の既定値以外の値は、常にその構造体型のインスタンスコンストラクターを通じて作成されます。
- 式のオペランドがすべて単純型定数なら、コンパイラはコンパイル時に式を評価できます。このような式は、*constant-expression*と呼ばれます(7.19を参照)。他の構造体型で定義されている演算子を含む式は、定数式とは見なされません。
- `const`宣言を使うことで、単純型の定数を宣言できます(10.4を参照)。他の構造体型では定数を宣言できませんが、`static readonly`フィールドを使うことで同様の効果を得ることができます。
- 単純型が関係する変換は他の構造体型で定義された変換演算子の評価の中で使用できますが、ユーザー定義の変換演算子を別のユーザー定義演算子の評価の中で使うことはできません(6.4.3を参照)。

#### 4.1.5 整数型

C#は、`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、および`char`の9つの整数型をサポートします。整数型のサイズおよび値の範囲は、以下のとおりです。

- `sbyte`型は、-128から127までの値を持つ、8ビットの符号付き整数を表します。
- `byte`型は、0から255までの値を持つ、8ビットの符号なし整数を表します。

- **short** 型は、-32768 から 32767 までの値を持つ、16 ビットの符号付き整数を表します。
- **ushort** 型は、0 から 65535 までの値を持つ、16 ビットの符号なし整数を表します。
- **int** 型は、-2147483648 から 2147483647 までの値を持つ、32 ビットの符号付き整数を表します。
- **uint** 型は、0 から 4294967295 までの値を持つ、32 ビットの符号なし整数を表します。
- **long** 型は、-9223372036854775808 から 9223372036854775807 までの値を持つ、64 ビットの符号付き整数を表します。
- **ulong** 型は、0 から 18446744073709551615 までの値を持つ、64 ビットの符号なし整数を表します。
- **char** 型は、符号なし 16 ビット整数値を表し、値の範囲は 0 ~ 65535 です。**char** 型に設定できる値の集合は、Unicode 文字セットに対応しています。**char** 型は **ushort** 型と同じ値を表現しますが、一部の演算は、いずれか一方の型に対してだけ実行できます。

整数型の単項演算子または二項演算子は、常に、符号付き 32 ビット精度、符号なし 32 ビット精度、符号付き 64 ビット精度、または符号なし 64 ビット精度の演算を行います。

- 単項演算子 + および ~ の場合、オペラントは型 **T** に変換されます。**T** は、**int**、**uint**、**long**、および **ulong** のうち、オペラントに設定される可能性のあるすべての値を完全に表現できる最初の型です。次に、**T** 型の有効桁数を使用して演算が実行されます。結果の型は **T** になります。
- 単項演算子 - の場合、オペラントは型 **T** に変換されます。**T** は、**int** および **long** のうち、オペラントに設定される可能性のあるすべての値を完全に表現できる最初の型です。次に、**T** 型の有効桁数を使用して演算が実行されます。結果の型は **T** になります。単項演算子 - は、**ulong** 型のオペラントには適用できません。
- 二項演算子 +、-、\*、/、%、&、^、|、==、!=、>、<、>=、および <= の場合、オペラントは型 **T** に変換されます。**T** は **int**、**uint**、**long**、および **ulong** のうち、両方のオペラントに設定される可能性のあるすべての値を完全に表現できる最初の型です。その後、型 **T** の精度を使って演算が実行され、結果の型は **T** (関係演算子の場合は **bool**) になります。二項演算子では、一方のオペラントに **long** 型を指定し、もう一方のオペラントに **ulong** 型を指定することはできません。
- 二項演算子 << および >> の場合、オペラントは型 **T** に変換されます。**T** は、**int**、**uint**、**long**、および **ulong** のうち、オペラントに設定される可能性のあるすべての値を完全に表現できる最初の型です。次に、**T** 型の有効桁数を使用して演算が実行されます。結果の型は **T** になります。

**char** 型は整数型に分類されますが、他の整数型とは次の 2 つの点で異なります。

- 他の型から **char** 型に暗黙の型変換が行われることはありません。特に、**sbyte**、**byte**、および **ushort** の各型で表現できる値の範囲は、**char** 型を使って完全に表現できますが、**sbyte**、**byte**、**ushort** から **char** への暗黙的な変換は行われません。
- **char** 型の定数は、*character-literal* として記述するか、**char** 型へのキャストと組み合わせて *integer-literal* として記述する必要があります。たとえば、(char)10 は '\x000A' と同じです。

整数型の算術演算および算術変換に対するオーバーフロー チェックを制御するには、**checked** および **unchecked** の演算子とステートメントを使用します (7.6.12 を参照)。**checked** のコンテキストでは、オーバーフローが発生すると、コンパイル エラーになるか、または **System.OverflowException** がスローされます。**unchecked** のコンテキストでは、オーバーフローは無視されて、結果が格納される型に収まらない上位ビットは破棄されます。

#### 4.1.6 浮動小数点型

C# でサポートされている浮動小数点型は、`float` と `double` の 2 種類です。`float` 型と `double` 型は、32 ビット単精度および 64 ビット倍精度の IEEE 754 フォーマットを使って表され、以下の値を表現できます。

- 正および負のゼロ。ほとんどの場合、正のゼロと負のゼロは、単純な値ゼロとまったく同じ動作を示します。ただし、ある種の演算では、正のゼロと負のゼロ (7.8.2 を参照) が区別されます。
- 正および負の無限大。無限大は、非ゼロの値をゼロで割った場合などに発生します。たとえば、`1.0 / 0.0` は正の無限大になり、`-1.0 / 0.0` は負の無限大になります。
- "**非数**" 値。しばしば `NaN` と省略表記されます。`NaN` は、ゼロをゼロで割るような、不正な浮動小数点演算で発生します。
- $s \times m \times 2^e$  型の有限個の非ゼロ値。 $s$  は 1 または -1 で、 $m$  と  $e$  は特定の浮動小数点型によって決定されます。`float` の場合、 $0 < m < 2^{24}$  および  $-149 \leq e \leq 104$ 、`double` の場合、 $0 < m < 2^{53}$  および  $-1075 \leq e \leq 970$  です。正規化されていない浮動小数点数は有効な非ゼロ値と見なされます。

`float` 型は、7 衔の有効桁数で約  $1.5 \times 10^{-45} \sim 3.4 \times 10^{38}$  の範囲の値を表現できます。

`double` 型は、15 ~ 16 衔の有効桁数で約  $5.0 \times 10^{-324} \sim 1.7 \times 10^{308}$  の範囲の値を表現できます。

二項演算子の一方のオペランドが浮動小数点型の場合、もう一方のオペランドは整数型または浮動小数点型である必要があり、演算は次のように評価されます。

- 一方のオペランドが整数型の場合、そのオペランドは、他方のオペランドの浮動小数点型に変換されます。
- その後、オペランドのいずれかが `double` 型の場合は、もう一方のオペランドも `double` 型に変換され、少なくとも `double` 型の範囲と精度を使って演算が実行されて、結果の型は `double` (関係演算子の場合は `bool`) になります。
- どちらのオペランドも `double` でない場合は、少なくとも `float` 型の範囲と精度を使って演算が実行されて、結果の型は `float` (関係演算子の場合は `bool`) になります。

代入演算子を含む浮動小数点演算子では、例外は発生しません。浮動小数点演算で例外状況が発生した場合は、次の規則に従って、ゼロ、無限大、または `NaN` が生成されます。

- 浮動小数点演算の結果が結果格納先の形式に対して小さすぎる場合は、演算の結果は正のゼロまたは負のゼロになります。
- 浮動小数点演算の結果が結果格納先の形式に対して大きすぎる場合は、演算の結果は正の無限大または負の無限大になります。
- 浮動小数点演算が無効である場合は、演算の結果は `NaN` になります。
- 浮動小数点演算のオペランドの一方または両方が `NaN` である場合は、演算の結果は `NaN` になります。

浮動小数点演算は、結果の型より高い精度で実行できます。たとえば、一部のハードウェアアーキテクチャは、`double` 型より範囲が広く精度が高い "extended" や "long double" のような浮動小数点型をサポートし、すべての浮動小数点演算をこの高精度型を使って暗黙に実行します。このようなハードウェアアーキテクチャが、低精度の浮動小数点数を扱うときにも高負荷の演算を行う設計になっている場合、パフォーマンスと精度の両方を低下させるような実装を強制することは望ましくないた

め、C# ではすべての浮動小数点演算に高精度の型を使用できます。より高い精度の結果が必要な場合を除き、これによって大きな違いが出ることはあまりありません。ただし、 $x * y / z$  のような形式の式で、乗算において **double** の範囲を超える結果になり、その後の除算で一時的な結果が **double** の範囲内に戻るような場合には、より広い範囲で式が評価されることにより、無限大ではなく有限の結果が生成される可能性があります。浮動小数点型の値をその型の厳密な精度に強制するには、明示的なキャストを使用します。

#### 4.1.7 Decimal 型

**decimal** 型は、財務計算や通貨計算に適した 128 ビットデータ型です。**decimal** 型は、28 ~ 29 桁の有効桁数で  $1.0 \times 10^{-28}$  から約  $7.9 \times 10^{28}$  までの範囲の値を表現できます。

**decimal** 型の有限個の値は、 $(-1)^s \times c \times 10^e$  の形式です。 $s$  は 0 または 1、係数  $c$  は  $0 \leq c < 2^{96}$ 、スケール  $e$  は  $0 \leq e \leq 28$  です。**decimal** 型は、符号付きゼロ、無限大、および非数 (NaN) をサポートしません。**decimal** は、10 の累乗でスケーリングされた 96 ビット整数として表されます。絶対値が 1.0m 未満の **decimal** 型の値は、小数点以下第 28 桁までは正確ですが、それより小さい値は正確ではありません。絶対値が 1.0m 以上の場合の **decimal** 型の値は、28 または 29 桁までは正確です。**float** 型および **double** 型とは異なり、**decimal** 型では、0.1 のような小数を正確に表すことができます。**float** 型や **double** 型では、このような数値は無限小数になることが多く、丸め誤差が発生する可能性が高くなります。

二項演算子のオペランドの 1 つが **decimal** 型の場合、もう 1 つのオペランドは整数型または **decimal** 型である必要があります。整数型のオペランドがある場合は、演算を実行する前に **decimal** 型に変換されます。

**decimal** 型の値の演算結果は、正確な結果 (各演算子に対して定義されたスケールを保持した値) を計算してから、表現に合わせて丸められた値です。結果は、最も近い表現可能な値に丸められます。表現可能な 2 つの値との差が等しい場合は、最下位の桁位置が偶数の値に丸められます。この方式は、"銀行型丸め方式" と呼ばれます。結果がゼロの場合、符号とスケールは常にゼロです。

10 進数の算術演算によって、絶対値が  $5 \times 10^{-29}$  以下の値になった場合、演算の結果はゼロになります。**decimal** 算術演算によって得られた結果が大きすぎて **decimal** 型で表現できない場合は、**System.OverflowException** がスローされます。

**decimal** 型は高い精度を備えていますが、値の範囲は浮動小数点型ほど大きくありません。したがって、浮動小数点型から **decimal** への変換ではオーバーフロー例外が発生する可能性があり、**decimal** から浮動小数点型への変換では精度の低下が発生する可能性があります。このような理由から、浮動小数点型と **decimal** の間では暗黙の型変換は行われず、明示的にキャストしないと、浮動小数点型と **decimal** のオペランドを同じ式の中に混在させることはできません。

#### 4.1.8 bool 型

**bool** 型は、ブール値の論理的な量を表現します。**bool** 型の有効な値は、**true** および **false** です。

**bool** 型と他の型の間には、標準的な型変換は存在しません。特に、**bool** 型と整数型は明確に区別されていて、整数値の代わりに **bool** 値を使ったり、**bool** 値の代わりに整数値を使ったりすることはできません。

C 言語および C++ 言語では、整数値または浮動小数点値のゼロ、または **null** ポインターは、ブール値の **false** に変換できます。整数値または浮動小数点値の非ゼロ、または非 **null** ポインターは、

ブール値の `true` に変換できます。C# では、このような型変換は、整数値または浮動小数点値とゼロとを明示的に比較するか、オブジェクト参照と `null` とを明示的に比較することで行います。

#### 4.1.9 列挙型

列挙型は、名前付き定数を持つ特別な型です。列挙型にはそれぞれ基になる型があり、`byte`、`sbyte`、`short`、`ushort`、`int`、`uint`、`long` または `ulong` のいずれかである必要があります。列挙型の値の集合は、その基になる型の値の集合と同じです。列挙型の値は、名前付き定数の値である必要はありません。列挙型は、列挙宣言によって定義します(14.1 を参照)。

#### 4.1.10 null 許容型

`null` 許容型は、"基になる型" のすべての値とその他の `null` 値を表すことができます。`null` 許容型は `T?` と表記され、基になる型は `T` と表記されます。この構文は、`System.Nullable<T>` の短縮形であり、この 2 つの形式はどちらを使用してもかまいません。

これに対し、"`null` 非許容の値型" は、`System.Nullable<T>` とその短縮形 `T?` (すべての `T`) 以外の値型のことであり、`null` 非許容の値型になるように制約されている型パラメーター(つまり、`struct` 制約を受ける型パラメーター)もその 1 つです。`System.Nullable<T>` 型は、`T` の値型制約を指定します(10.1.5 を参照)。したがって、`null` 許容型の基になる型は、どのような `null` 非許容の値型でもかまいません。`null` 許容型の基になる型には、`null` 許容型または参照型を使用できません。たとえば、`int??` および `string??` は無効な型です。

`null` 許容型 `T?` のインスタンスには、次の 2 つのパブリックな読み取り専用プロパティがあります。

- `bool` 型のプロパティ `HasValue`
- `T` 型のプロパティ `value`

`HasValue` が `true` のインスタンスを非 `null` と呼びます。非 `null` のインスタンスは既知の値を含み、`value` はその値を返します。

`HasValue` が `false` のインスタンスを `null` と呼びます。`null` のインスタンスの値は未定義です。`null` のインスタンスの `value` を読み取ろうとすると、`System.InvalidOperationException` がスローされます。`null` 許容インスタンスの `value` プロパティにアクセスする処理は、"ラップ解除" と呼ばれます。

すべての `null` 許容型 `T?` は、既定のコンストラクターに加えて、型 `T` の单一の引数を取るパブリックコンストラクターを持ちます。型 `T` の値 `x` の場合に、次の形式でコンストラクターを呼び出したとします。

```
new T?(x)
```

これにより、`value` プロパティが `x` である `T?` の非 `null` のインスタンスが作成されます。指定された値に対して `null` 許容型の非 `null` インスタンスを作成する処理は、"ラップ" と呼ばれます。

`null` リテラルから `T?` への暗黙の型変換(6.1.5 を参照)と、`T` から `T?` への暗黙の型変換(6.1.4 を参照)が可能です。

#### 4.2 参照型

参照型は、クラス型、インターフェイス型、配列型、またはデリゲート型です。

```
reference-type:  
  class-type  
  interface-type  
  array-type  
  delegate-type  
  dynamic  
  
class-type:  
  type-name  
  object  
  string  
  
interface-type:  
  type-name  
  
array-type:  
  non-array-type  rank-specifiers  
  
non-array-type:  
  type  
  
rank-specifiers:  
  rank-specifier  
  rank-specifiers  rank-specifier  
  
rank-specifier:  
  [ dim-separatorsopt ]  
  
dim-separators:  
  ,  
  dim-separators ,  
  
delegate-type:  
  type-name
```

参照型の値は、型の "インスタンス" に対する参照で、"オブジェクト" として知られています。特別な値 `null` は、すべての参照型と互換性があり、インスタンスが存在しないことを示します。

#### 4.2.1 クラス型

クラス型は、データ メンバー (定数とフィールド)、関数メンバー (メソッド、プロパティ、イベント、インデクサー、演算子、インスタンス コンストラクター、デストラクター、および静的コンストラクター)、および入れ子になった型を含むデータ構造を定義します。クラス型は、継承をサポートしています。継承とは、派生クラスが基底クラスを拡張および限定するための機構です。クラス型のインスタンスは、*object-creation-expressions* を使って作成します (7.6.10.1 を参照)。

クラス型については 10 で説明します。

次の表に示すように、いくつかの定義済みのクラス型は、C# 言語で特別な意味を持ちます。

クラス型	説明
<code>System.Object</code>	他のすべての型に対する最終的な基底クラス。4.2.2 を参照。
<code>System.String</code>	C# 言語の文字列型。4.2.4 を参照。
<code>System.ValueType</code>	すべての値型の基底クラス。4.1.1 を参照。
<code>System.Enum</code>	すべての列挙型の基底クラス。14 を参照。
<code>System.Array</code>	すべての配列型の基底クラス。12 を参照。
<code>System.Delegate</code>	すべてのデリゲート型の基底クラス。15 を参照。
<code>System.Exception</code>	すべての例外型の基底クラス。16 を参照。

#### 4.2.2 オブジェクト型

`object` クラス型は、他のすべての型の最終的な基底クラスです。C# のすべての型は、`object` クラス型から直接または間接に派生しています。

`object` というキーワードは、定義済みの `System.Object` クラスに対する単なるエイリアスです。

#### 4.2.3 動的な型

`object` などの `dynamic` 型は、任意のオブジェクトを参照できます。`dynamic` 型の式に演算子が適用される場合、その解決はプログラムの実行時まで遅延されます。このため、演算子を参照先オブジェクトに正常に適用できなくても、コンパイル時にはエラーが生成されません。代わりに、実行時に演算子の解決に失敗した時点で例外がスローされます。

動的な型の詳細は 4.7 で、動的バインディングの詳細は 7.2.2 で説明されています。

#### 4.2.4 文字列型

`string` 型は、`object` から直接継承したシールクラス型です。`string` クラスのインスタンスは、Unicode 文字列を表しています。

`string` 型の値は、リテラル文字列として記述できます(2.4.4.5 を参照)。

`string` というキーワードは、定義済みの `System.String` クラスに対する単なるエイリアスです。

#### 4.2.5 インターフェイス型

インターフェイスはコントラクトを定義します。インターフェイスを実装するクラスや構造体は、そのコントラクトと一致する必要があります。インターフェイスは複数の基本インターフェイスから継承できます。また、クラスや構造体は複数のインターフェイスを実装できます。

インターフェイス型については 13 で説明します。

#### 4.2.6 配列型

配列とは、算出されたインデックスを使用してアクセスされる 0 個以上の変数を含むデータ構造です。配列に含まれる変数は、配列の要素とも呼ばれ、すべて同じ型です。この型を配列の要素型と呼びます。

配列型については 12 で説明します。

#### 4.2.7 デリゲート型

デリゲートとは、1つまたは複数のメソッドを参照するデータ構造です。インスタンス メソッドの場合は、対応するオブジェクト インスタンスも参照します。

デリゲートと最も近い C または C++ の機能は関数ポインターですが、関数ポインターでは静的関数だけを参照できるのに対し、デリゲートでは静的メソッドとインスタンス メソッドの両方を参照できます。後者の場合、デリゲートは、メソッドのエントリ ポイントへの参照だけでなく、メソッドを呼び出すオブジェクト インスタンスへの参照も格納します。

デリゲート型については 15 で説明します。

### 4.3 ボックス化とボックス化解除

ボックス化とボックス化解除は、C# の型システムの中心的な概念です。これは、*value-type* のすべての値と **object** 型間の変換を可能にするため、*value-types* と *reference-types* との橋渡しの役目を果たします。ボックス化とボックス化解除の機能により、型システムを統一的に見ることができるようになり、すべての型の値を最終的にオブジェクトとして扱うことができます。

#### 4.3.1 ボックス化変換

ボックス化変換では、*value-type* から *reference-type* への暗黙の変換ができます。ボックス化変換には、次のような変換があります。

- 任意の *value-type* から **object** 型への変換
- 任意の *value-type* から **System.ValueType** 型への変換
- 任意の *non-nullable-value-type* から、*value-type* が実装する任意の *interface-type* への変換
- 任意の *nullable-type* から、*nullable-type* の基になる型が実装する任意の *interface-type* への変換
- 任意の *enum-type* から **System.Enum** 型への変換
- 基になる型が *enum-type* である任意の *nullable-type* から、**System.Enum** への変換

型パラメーターからの暗黙の型変換は、実行時に値型から参照型に変換することになった場合、ボックス化変換として実行されることに注意してください (6.1.10 を参照)。

*non-nullable-value-type* の値のボックス化では、オブジェクト インスタンスの割り当て、およびそのインスタンスへの *non-nullable-value-type* の値のコピーが行われます。

*nullable-type* の値をボックス化すると、それが **null** 値 (*HasValue* が **false**) の場合、**null** 参照が作成されます。それ以外の場合、基になる値をラップ解除してボックス化した結果になります。

*non-nullable-value-type* の値をボックス化する実際のプロセスは、ジェネリック "ボックス化クラス" の存在を仮定すると、よくわかります。このクラスは次のように宣言された場合と同様に動作します。

```
sealed class Box<T>: System.ValueType
{
    T value;
    public Box(T t) {
        value = t;
    }
}
```

このようにすると、*T*型の値 *v* のボックス化では、式 `new Box<T>(v)` が実行された後、結果として生成されるインスタンスが `object` 型の値として返されます。したがって、次のステートメントのようになります。

```
int i = 123;
object box = i;
```

上のステートメントは、概念的には次のステートメントに対応しています。

```
int i = 123;
object box = new Box<int>(i);
```

上記の `Box<T>` のようなボックス化クラスは実際には存在せず、ボックス化された値の動的な型は実際にはクラス型ではありません。実際は、ボックス化された型 *T* の値は動的な型 *T*を持ち、`is` 演算子を使って動的な型のチェックを行うと、単に型 *T*を参照できます。次に例を示します。

```
int i = 123;
object box = i;
if (box is int) {
    Console.WriteLine("Box contains an int");
}
```

この例では、コンソールに "Box contains an int" という文字列が出力されます。

ボックス化変換は、ボックス化される値のコピーを作成することを意味しています。これは、*reference-type* から `object` 型への変換とは異なります。参照型から `object` 型への場合、値は引き続き同じインスタンスを参照し、単に先に派生した `object` 型と見なされます。たとえば、次のような宣言について考えます。

```
struct Point
{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

この宣言に対する次のステートメントについて考えます。

```
Point p = new Point(10, 10);
object box = p;
p.x = 20;
Console.WriteLine(((Point)box).x);
```

この例では、`box` への `p` の代入において発生する暗黙のボックス化操作によって `p` の値がコピーされるため、10 という値がコンソールに出力されます。`Point` を代わりに `class` として宣言すると、`p` と `box` は同じインスタンスを参照するため、20 という値が出力されます。

### 4.3.2 ボックス化解除変換

ボックス化解除変換では、*reference-type* 型から *value-type* への明示的な変換ができます。ボックス化解除変換には、次のような変換があります。

- `object` 型から、任意の *value-type* への変換
- `System.ValueType` 型から、任意の *value-type* への変換
- 任意の *interface-type* から、その *interface-type* を実装する任意の *non-nullable-value-type* への変換

- 任意の *interface-type* から、基になる型がその *interface-type* を実装する任意の *nullable-type* への変換
- `System.Enum` 型から *enum-type* 型への変換
- `System.Enum` 型から、基になる型が *enum-type* である任意の *nullable-type* への変換

型パラメーターへの明示的な変換は、実行時に参照型から値型に変換することになった場合、ボックス化解除変換として実行されることに注意してください (6.2.6 を参照)。

*non-nullable-value-type* へのボックス化解除では、最初にオブジェクトインスタンスが指定の *non-nullable-value-type* のボックス化された値であることが確認され、次に値がインスタンスからコピーされます。

*nullable-type* へのボックス化解除では、変換前のオペランドが `null` の場合、*nullable-type* の `null` 値が作成されます。それ以外の場合、オブジェクトインスタンスを *nullable-type* の基になる型にボックス化解除した結果をラップしたものになります。

前のセクションで説明した仮想のボックス化クラスを参考にすると、`box` オブジェクトから *value-type T* へのボックス化解除変換では、式 `((Box<T>)box).value` が実行されます。したがって、次のステートメントのようになります。

```
object box = 123;
int i = (int)box;
```

上のステートメントは、概念的には次のステートメントに対応しています。

```
object box = new Box<int>(123);
int i = ((Box<int>)box).value;
```

実行時に、指定した *non-nullable-value-type* へのボックス化解除変換が成功するためには、変換前のオペランドの値がその *non-nullable-value-type* のボックス化された値への参照になっている必要があります。変換前のオペランドが `null` の場合、`System.NullReferenceException` がスローされます。変換前のオペランドとして互換性のないオブジェクトへの参照を指定すると、

`System.InvalidCastException` がスローされます。

実行時に、指定した *nullable-type* へのボックス化解除変換が成功するためには、変換前のオペランドの値が `null` であるか、その *nullable-type* の基になっている *non-nullable-value-type* のボックス化された値への参照になっている必要があります。変換前のオペランドとして互換性のないオブジェクトへの参照を指定すると、`System.InvalidCastException` がスローされます。

## 4.4 構築された型

ジェネリック型宣言は、自動的に "非バインド ジェネリック型" を示します。非バインド ジェネリック型は、"型引数" を適用することによって多くの異なる型を形成するための "設計図" として使用されます。型引数は、ジェネリック型の名前のすぐ後に山かっこ (<>) で囲んで記述します。少なくとも 1 つの型引数を持つ型を "構築された型" と呼びます。構築された型は、型名を使用できる言語内のほとんどの場所で使用できます。非バインド ジェネリック型は、*typeof-expression* (7.6.11 を参照) 内でのみ使用できます。

また、構築された型は、簡易名として式で使用 (7.6.2 を参照) したり、メンバーにアクセス (7.6.4 を参照) するときに使用したりできます。

*namespace-or-type-name* が評価されるときは、正しい数の型引数を持つジェネリック型のみが考慮されます。したがって、同じ識別子でも型によって型パラメーターの数が異なる場合は、複数の型を識

別できます。これは、同じプログラム内にジェネリック クラスと非ジェネリック クラスを混在させるときに便利です。

```
namespace Widgets
{
    class Queue {...}
    class Queue<TElement> {...}
}

namespace MyApplication
{
    using Widgets;
    class X
    {
        Queue q1;           // Non-generic Widgets.Queue
        Queue<int> q2;     // Generic Widgets.Queue
    }
}
```

*type-name* は、型パラメーターを直接指定しなくとも、構築された型を識別できる場合があります。これは、型がジェネリック クラス宣言の入れ子になっている場合で、次のように、外側の宣言のインスタンス型が暗黙的に名前検索に使用されます (10.3.8.6 を参照)。

```
class Outer<T>
{
    public class Inner {...}
    public Inner i;           // Type of i is Outer<T>.Inner
}
```

安全でないコードでは、構築された型を *unmanaged-type* (18.2 を参照) として使用できません。

#### 4.4.1 型引数

型引数リストの各引数は単なる *type* です。

```
type-argument-list:
    < type-arguments >

type-arguments:
    type-argument
    type-arguments , type-argument

type-argument:
    type
```

安全でないコード (18 を参照) では、*type-argument* はポインター型でない場合があります。各型引数は、対応する型パラメーターの制約を満たす必要があります (10.1.5 を参照)。

#### 4.4.2 オープン型とクローズ型

すべての型は、"オープン型" か "クローズ型" のいずれかに分類されます。オープン型は、型パラメーターと一緒に使用する型です。より具体的には、次のとおりです。

- 型パラメーターはオープン型を定義します。
- 配列型は、要素の型がオープン型の場合のみ、オープン型です。
- 構築された型は、1つ以上の型引数がオープン型の場合のみ、オープン型です。構築された入れ子になった型は、1つ以上の型引数または外側の型の型引数がオープン型の場合のみ、オープン型です。

クローズ型とは、オープン型でない型です。

実行時、ジェネリック型宣言内のすべてのコードは、ジェネリック宣言に型引数を適用することによって作成されたクローズ構築型のコンテキストで実行されます。ジェネリック型内の各型パラメーターは、特定の実行時の型にバインドされます。すべてのステートメントおよび式の実行時の処理ではクローズ型が発生し、オープン型は、コンパイル時の処理でのみ発生します。

クローズ構築型には独自の静的変数セットがあり、このセットは他のクローズ構築型と共有されません。オープン型は実行時には存在しないため、オープン型に関連付けられた静的変数はありません。2つのクローズ構築型は、同じ非バインド ジェネリック型から構築された場合は同じ型になり、対応する型引数も同じ型になります。

#### 4.4.3 バインド型と非バインド型

"**非バインド型**" という用語は、非ジェネリック型または非バインド ジェネリック型を意味します。"**バインド型**" という用語は、非ジェネリック型または構築された型を意味します。

非バインド型は、型宣言で宣言されたエンティティを意味します。非バインド ジェネリック型自体は型ではなく、変数、引数、または戻り値の型や、基本型として使用することはできません。非バインド ジェネリック型を参照できる唯一の構造体は、`typeof` 式 (7.6.11 を参照) です。

#### 4.4.4 制約の充足

構築された型またはジェネリック メソッドを参照する場合に、指定された型引数は、ジェネリック型またはメソッドで宣言されている型パラメーターの制約に照らしてチェックされます (10.1.5 を参照)。`where` 句ごとに、名前付き型パラメーターに対応する型引数 `A` が次のような制約に照らしてチェックされます。

- 制約がクラス型、インターフェイス型、または型パラメーターの場合に、`C` が、指定された型引数を制約内にある型パラメーターで置き換えた制約であるとします。制約を満たすには、型 `A` は次のいずれかの方法で型 `C` に変換できる必要があります。
  - 恒等変換 (6.1.1 を参照)
  - 暗黙の参照変換 (6.1.6 を参照)
  - 型 `A` が `null` 非許容の値型の場合は、ボックス化変換 (6.1.7 を参照)
  - 型パラメーター `A` から `C` への暗黙の参照変換、ボックス化変換、または型パラメーター変換
- 制約が参照型制約 (`class`) の場合、型 `A` は次のいずれかの条件を満たす必要があります。
  - `A` は、インターフェイス型、クラス型、デリゲート型、または配列型であること。`System.ValueType` および `System.Enum` は、この制約を満たす参照型です。
  - `A` が参照型であることがわかっている型パラメーターであること (10.1.5 を参照)。
- 制約が値型の制約 (`struct`) の場合、型 `A` は次のいずれかの条件を満たす必要があります。
  - `A` は構造体型または列挙型であり、`null` 許容型でないこと。`System.ValueType` および `System.Enum` は、この制約を満たさない参照型です。
  - `A` が値型の制約を持つ型パラメーターであること (10.1.5 を参照)。

- 制約がコンストラクター制約 `new()` である場合、型 `A` は `abstract` であり、パラメーターのないパブリック コンストラクターを持つ必要があります。これは、次のいずれかの条件が満たされる場合に成り立ちます。
  - `A` が値型であること。これは、すべての値型には既定のパブリック コンストラクターがあるためです (4.1.2 を参照)。
  - `A` がコンストラクター制約を持つ型パラメーターであること (10.1.5 を参照)。
  - `A` が値型の制約を持つ型パラメーターであること (10.1.5 を参照)。
  - `A` が `abstract` でないクラスで、明示的に宣言された、パラメーターを持たない `public` コンストラクターを含むこと。
  - `A` が `abstract` でなく、既定のコンストラクターを持つこと (10.11.4 を参照)。

型引数が型パラメーターの制約を 1 つでも満たさない場合は、コンパイル エラーになります。

型パラメーターは継承されないため、制約も継承されません。次の例では、`T` が基底クラス `B<T>` によって課せられた制約を満たすように、`D` は型パラメーター `T` に制約を指定する必要があります。それに対して、クラス `E` は制約を指定する必要はありません。これは、`List<T>` が任意の `T` に対して `IEnumerable` を実装するためです。

```
class B<T> where T: IEnumerable {...}
class D<T>: B<T> where T: IEnumerable {...}
class E<T>: B<List<T>> {...}
```

## 4.5 型パラメーター

型パラメーターは、実行時にそのパラメーターをバインドする値型または参照型を指定します。

*type-parameter:  
identifier*

型パラメーターはさまざまな実際の型引数によってインスタンス化できるため、型パラメーターの操作と制約は他の型と多少異なります。次のような制約があります。

- 型パラメーターを使用して、基底クラス (10.2.4 を参照) またはインターフェイス (13.1.3 を参照) を直接宣言することはできません。
- 型パラメーターのメンバー検索規則は、型パラメーターに適用される制約 (存在する場合) によって異なります。これについては、7.4 で詳細に説明しています。
- 型パラメーターで使用できる変換は、型パラメーターに適用される制約 (存在する場合) によって異なります。これについては、6.1.10 と 6.2.6 で詳細に説明しています。
- リテラル `null` は、型パラメーターが参照型であるとわかっている場合を除き、型パラメーターが指定する型には変換できません (6.1.10 を参照)。ただし、代わりに `default` 式 (7.6.13 を参照) を使用できます。また、型パラメーターが指定する型を持つ値は、その型パラメーターに値型の制約がない限り、`==` および `!=` (7.10.6 を参照) を使用して `null` と比較できます。
- `new` 式 (7.6.10.1 を参照) は、型パラメーターが `constructor-constraint` または値型の制約 (10.1.5 を参照) を受けている場合に限り、型パラメーターと一緒に使用できます。
- 型パラメーターは、属性内では使用できません。

- 型パラメーターをメンバー アクセス (7.6.4 を参照) または型名 (3.8 を参照) に使用して、静的メンバーまたは入れ子になった型を識別することはできません。
- 安全でないコードでは、型パラメーターを *unmanaged-type* (18.2 を参照) として使用できません。

型としては、型パラメーターは単なるコンパイル時の構造です。実行時に各型パラメーターは、ジェネリック型宣言に型引数を与えて指定した、実行時の型にバインドされます。したがって、型パラメーター付きで宣言された変数の型は、実行時にはクローズ構築型 (4.4.2 を参照) になります。型パラメーターを含むすべてのステートメントおよび式を実行するときは、そのパラメーターの型引数として指定した実際の型が使用されます。

## 4.6 式ツリー型

"式ツリー" を使用すると、ラムダ式を実行可能コードとしてではなく、データ構造として表すことができます。式ツリーは "式ツリー型" の値であり、`System.Linq.Expressions.Expression<D>` という形式になっています。ここで、`D` は任意のデリゲート型です。この後では、式ツリー型の表記に短縮形の `Expression<D>` を使用します。

ラムダ式からデリゲート型 `D` への変換が存在する場合、式ツリー型 `Expression<D>` への変換も存在します。ラムダ式からデリゲート型への変換では、そのラムダ式の実行可能コードを参照するデリゲートが生成されますが、式ツリー型への変換では、そのラムダ式の式ツリー表現が作成されます。

式ツリーでは、メモリ内でのラムダ式のデータ表現が効率よく行われ、ラムダ式の構造が明確にわかりやすくなります。

デリゲート型の `D` と同様に、`Expression<D>` は `D` と同じパラメーター型と戻り値の型を持ちます。

次の例は、ラムダ式を実行可能コードと式ツリーの両方で表したもので、`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> del = x => x + 1; // code
Expression<Func<int,int>> exp = x => x + 1; // data
```

上記の代入の後、デリゲート `del` は `x+1` を返すメソッドを参照し、式ツリー `exp` は式 `x => x + 1` を記述するデータ構造を参照します。

ジェネリック型 `Expression<D>` の正確な定義も、ラムダ式を式ツリー型に変換するときの式ツリーの構築に関する厳密な規則も、この仕様書の範囲外であるため、他の資料を参照してください。

次の 2 つのことを明確にすることが重要です。

- すべてのラムダ式が式ツリーに変換できるとは限りません。例を挙げると、ステートメント本体を持つラムダ式や、代入式を持つラムダ式は、式ツリーとして表すことができません。このような場合、変換は存在しますが、コンパイル時にエラーになります。これらの例外についての詳細な説明については、0 で説明しています。
- `Expression<D>` は、デリゲート型の `D` を生成するインスタンス メソッド `Compile` を提供します。

```
Func<int,int> del2 = exp.Compile();
```

このデリゲートを呼び出すと、式ツリーで表されているコードが実行されます。したがって、上記の定義が与えられている場合、`del` と `del2` は等価であり、次の 2 つのステートメントは同じ結果になります。

```
int i1 = del(1);
int i2 = del2(1);
```

このコードを実行した後、`i1` と `i2` はどちらも値が 2 になります。

## 4.7 動的な型

`dynamic` 型は、C#において特別な意味を持ちます。その目的は、7.2.2 で詳細に説明する動的バインディングを可能にすることです。

`dynamic` は、次の点を除いて `object` と同一であると見なされます。

- `dynamic` 型の式に対する演算は、動的にバインドできます (7.2.2 を参照)。
- 型推論 (7.5.2 を参照) で `dynamic` と `object` の両方が候補になる場合は、`dynamic` が優先されます。この等価性により、次の条件が満たされます。
- `object` と `dynamic` の間、および `dynamic` を `object` で置換する場合は同じ構築された型の間に、暗黙の恒等変換があります。
- `object` との暗黙の型変換および明示的な変換は、`dynamic` との間にも適用されます。
- `dynamic` を `object` で置換したときに同一になるメソッドシグネチャは、同じシグネチャであると見なされます。

`dynamic` 型は、実行時に `object` と区別できません。

`dynamic` 型の式は、"動的な式" と呼ばれます。

## 5. 変数

変数は格納場所を表しています。すべての変数は、その変数に格納できる値を指定する型を持ちます。C# はタイプセーフな言語であり、変数には常に適切な型の値が格納されることが、C# コンパイラによって保証されます。変数の値は、代入することで、または ++ 演算子および -- 演算子を使って、変更できます。

変数から値を取得するには、変数への "確実な代入" (5.3 を参照) が必要です。

以下で説明するように、変数の状態は、"初期代入あり"、"初期代入なし" のいずれかです。初期代入ありの変数は、明確に定義された初期値を持ち、常に確実に代入されているものと見なされます。初期代入なしの変数は、初期値を持っていません。初期代入なしの変数が特定の位置において確実に代入されているものと見なされるには、その位置に至る可能性のあるすべての実行パスにおいて、変数への代入が行われている必要があります。

### 5.1 変数のカテゴリ

C# で定義されている変数のカテゴリは、静的変数、インスタンス変数、配列要素、値パラメーター、参照パラメーター、出力パラメーター、およびローカル変数の 7 種類です。以下では、これらの各カテゴリについて説明します。

次に例を示します。

```
class A
{
    public static int x;
    int y;

    void F(int[] v, int a, ref int b, out int c) {
        int i = 1;
        c = a + b++;
    }
}
```

x は静的変数、y はインスタンス変数、v[0] は配列要素、a は値パラメーター、b は参照パラメーター、c は出力パラメーター、i はローカル変数です。

#### 5.1.1 静的変数

**static** 修飾子を指定して宣言されているフィールドは、**静的変数**です。静的変数は、静的コンストラクター (10.12 を参照) が、それを含む型に対して実行される前に発生し、関連付けられたアプリケーションドメインが消失すると消失します。

静的変数の初期値は、変数の型の既定値 (5.2 を参照) です。

確実な代入のチェックにあたっては、静的変数は初期代入ありと見なされます。

#### 5.1.2 インスタンス変数

**static** 修飾子を指定せずに宣言されたフィールドは、**インスタンス変数**と呼ばれます。

### 5.1.2.1 クラスのインスタンス変数

クラスのインスタンス変数は、そのクラスの新しいインスタンスが作成されると存在するようになり、そのインスタンスに対する参照がなくなってインスタンスのデストラクター(ある場合)が実行されると存在しなくなります。

クラスのインスタンス変数の初期値は、変数の型の既定値(5.2を参照)です。

確実な代入のチェックにあたっては、クラスのインスタンス変数は初期代入ありと見なされます。

### 5.1.2.2 構造体のインスタンス変数

構造体のインスタンス変数の有効期間は、それが属している構造体変数とまったく同じです。つまり、構造型型の変数が存在するようになると構造体のインスタンス変数も存在するようになります。構造型型の変数が存在しなくなると構造体のインスタンス変数も存在しなくなります。

構造体のインスタンス変数の初期代入状態は、それを含む構造体変数の初期代入状態と同じです。つまり、構造体変数が初期代入ありと見なされると、インスタンス変数も初期代入ありと見なされ、構造体変数が初期代入なしと見なされると、インスタンス変数も初期代入なしと見なされます。

### 5.1.3 配列要素

配列の要素は、配列インスタンスが作成されると存在するようになり、その配列インスタンスに対する参照がなくなると存在しなくなります。

配列の各要素の初期値は、配列要素の型の既定値(5.2を参照)です。

確実な代入のチェックにあたっては、配列要素は初期代入ありと見なされます。

### 5.1.4 値パラメーター

`ref`修飾子または`out`修飾子を付けずに宣言されたパラメーターは、"値パラメーター"です。

値パラメーターは、パラメーターが属している関数メンバー(メソッド、インスタンス コンストラクター、アクセサー、演算子)または匿名関数が呼び出されると存在するようになり、呼び出しで指定された引数の値で初期化されます。値パラメーターは通常、関数メンバーまたは匿名関数が呼び出し側に制御を返すと存在しなくなります。ただし、値パラメーターが匿名関数(7.15を参照)によってキャプチャされると、その寿命は少なくともその匿名関数から作成されたデリゲートまたは式ツリーがガーベジコレクションの対象になるまで延長されます。

確実な代入のチェックにあたっては、値パラメーターは初期代入ありと見なされます。

### 5.1.5 参照パラメーター

`ref`修飾子を使って宣言されたパラメーターは参照パラメーターです。

参照パラメーターは、新しい格納場所を作成しません。参照パラメーターは、関数メンバーまたは匿名関数の呼び出しで引数として指定された変数と同じ格納場所を表します。したがって、参照パラメーターの値は、基になっている変数と常に同じです。

参照パラメーターには、確実な代入に関する次の規則が適用されます。5.1.6で説明している出力パラメーターのさまざまな規則に注意してください。

- 関数メンバーまたはデリゲートの呼び出しの参照パラメーターとして変数を渡す前に、変数を確実に代入された状態(5.3を参照)にする必要があります。
- 関数メンバーまたは匿名関数の中では、参照パラメーターは初期代入ありと見なされます。

構造体型のインスタンス メソッドまたはインスタンス アクセサーでは、`this` キーワードは、構造体型の参照パラメーターとして正しく動作します (7.6.7 を参照)。

### 5.1.6 出力パラメーター

`out` 修飾子を使って宣言されたパラメーターは出力パラメーターです。

出力パラメーターは、新しい格納場所の作成を行いません。出力パラメーターは、関数メンバーまたはデリゲートの呼び出しで引数として指定された変数と同じ格納場所を表します。したがって、出力パラメーターの値は、基になっている変数と常に同じです。

出力パラメーターには、確実な代入に関する次の規則が適用されます。5.1.5 で説明している参照パラメーターのさまざまな規則に注意してください。

- 関数メンバーまたはデリゲートの呼び出しの出力パラメーターとして変数を渡す前に、変数を確実に代入された状態にする必要はありません。
- 関数メンバーまたはデリゲートの呼び出しが正常に完了した後、出力パラメーターとして渡された各変数は、その実行パスの中で代入されるものと見なされます。
- 関数メンバーまたは匿名関数の中では、出力パラメーターは初期代入なしと見なされます。
- 関数メンバーまたは匿名関数のすべての出力パラメーターは、関数メンバーまたは匿名関数が返る前に確実に代入 (5.3 を参照) される必要があります。

構造体型のインスタンス コンストラクターでは、`this` キーワードは、構造体型の出力パラメーターとして正しく動作します (7.6.7 を参照)。

### 5.1.7 ローカル変数

"ローカル変数" は、*local-variable-declaration* (*block*、*for-statement*、*switch-statement*、または *using-statement* で指定可能)、*foreach-statement*、または *try-statement* の *specific-catch-clause* によって宣言されます。

ローカル変数の有効期間は、プログラムの実行中にローカル変数の格納領域が確保されている間です。この有効期間は少なくとも、関係付けられた *block*、*for-statement*、*switch-statement*、*using-statement*、*foreach-statement*、または *specific-catch-clause* の開始から、その *block*、*for-statement*、*switch-statement*、*using-statement*、*foreach-statement*、または *specific-catch-clause* の実行が終了するまでです。囲まれた *block* に入ったり、メソッドを呼び出したりすると、現在の *block*、*for-statement*、*switch-statement*、*using-statement*、*foreach-statement*、または *specific-catch-clause* の実行が中断されますが、終了することはありません。ローカル変数が匿名関数 (7.15.5.1 を参照) によってキャプチャされた場合、その有効期間は少なくとも、その匿名関数から作成されたデリゲートまたは式のツリーが、キャプチャされた変数を参照する他のオブジェクトと共に、ガベージコレクションの対象になるまで延長されます。

親の *block*、*for-statement*、*switch-statement*、*using-statement*、*foreach-statement*、または *specific-catch-clause* が再帰的に呼び出される場合、呼び出されるたびにローカル変数の新しいインスタンスが作成され、*local-variable-initializer* があれば、そのたびに評価されます。

*local-variable-declaration* によって導入されたローカル変数は自動的には初期化されないので、既定値もありません。確実な代入のチェックにあたっては、*local-variable-declaration* によって導入されたローカル変数は初期代入なしと見なされます。*local-variable-declaration* では *local-variable-initializer*

を指定できます。*local-variable-initializer* を指定すると、変数は、初期化式の後でのみ明確に代入されているものと見なされます (5.3.3.4 を参照)。

*local-variable-declaration* によって導入されたローカル変数のスコープでは、*local-variable-declarator* より前の記述位置でそのローカル変数を参照すると、コンパイルエラーになります。ローカル変数宣言が暗黙の場合 (8.5.1 を参照)、その *local-variable-declarator* 内で変数を参照するとエラーが返されます。

*foreach-statement* または *specific-catch-clause* で導入されたローカル変数は、その変数のスコープ全体において確実に代入されているものと見なされます。

ローカル変数の実際の有効期間は、実装に依存します。たとえば、コンパイラは、ブロック内のローカル変数がそのブロックのごく一部でしか使用されないことを静的に判断する場合があります。コンパイラは、この分析に基づいて、変数の格納領域の有効期間が、変数を含むブロックの有効期間よりも短いコードを生成する場合があります。

ローカル参照変数が参照するストレージは、ローカル参照変数の有効期間にかかわらず個別に解放されます (3.9 を参照)。

## 5.2 既定値

以下のカテゴリの変数は、自動的に既定値に初期化されます。

- 静的変数
- クラスインスタンスのインスタンス変数
- 配列要素

変数の既定値は、変数の型に依存し、次のようにして決定されます。

- *value-type* の変数の場合、既定値は、*value-type* の既定のコンストラクター (4.1.2 を参照) によって計算された値と同じです。
- *reference-type* の変数の場合、既定値は `null` です。

既定値への初期化では、普通、メモリマネージャーまたはガベージコレクターでメモリを全ビットゼロに初期化した後、変数に割り当てます。そのため、全ビットゼロの値を使って `null` 参照を表すのが便利です。

## 5.3 確実な代入

コンパイラが関数メンバーの実行可能コードの特定の位置について静的なフロー分析を行い (5.3.3 を参照)、変数が自動的に初期化されていること、または少なくとも 1 つの代入の対象になっていることが証明された場合、その変数はその位置において "確実に代入" されています。非公式的には確実な代入の規則は、次のように述べられます。

- 初期代入ありの変数 (5.3.1 を参照) は、常に確実に代入されていると見なされます。
- 初期代入なしの変数 (5.3.2 を参照) は、ある位置に到達するまでに通過する可能性のあるすべての実行パスに以下の少なくとも 1 つが含まれている場合、その位置においては確実に代入されているものと見なされます。
  - 変数が左オペランドとして指定されている単純な代入 (7.17.1 を参照)。

- 変数を出力パラメーターとして渡す呼び出し式 (7.6.5 を参照) またはオブジェクト作成式 (7.6.10.1 を参照)。
- ローカル変数の場合は、変数初期化子を含むローカル変数宣言 (8.5.1 を参照)。

上記の非公式な規則の基礎となる公式な仕様については、5.3.1、5.3.2、5.3.3 で説明しています。

*struct-type* の変数のインスタンス変数に関する確実な代入の状態は、全体としてだけではなく個別にも追跡されます。*struct-type* の変数とそのインスタンス変数には、上記の規則に加えて以下の規則も適用されます。

- *struct-type* の変数が確実に代入されていると見なされる場合は、そこに含まれるインスタンス変数も確実に代入されていると見なされます。
- *struct-type* の変数のすべてのインスタンス変数が確実に代入されていると見なされる場合は、その *struct-type* の変数も確実に代入されていると見なされます。

確実な代入は、以下のコンテキストにおける要件です。

- 変数は、変数の値が取得されるすべての位置において、確実に代入されている必要があります。これにより、未定義の値が発生しないことが保証されます。式の中で使われている変数については、次の場合を除き、変数の値が取得されるものと見なします。
  - 変数が、単純な代入の左オペランドになっている場合。
  - 変数が、出力パラメーターとして渡されている場合。
  - 変数が *struct-type* の変数で、メンバーアクセスの左オペランドとして指定されている場合。
- 変数は、参照パラメーターとして渡されるすべての位置において、確実に代入されている必要があります。これにより、呼び出される関数メンバーは、参照パラメーターが初期代入ありであると見なすことができます。
- 関数メンバーのすべての出力パラメーターは、関数メンバーが制御を返す (**return** ステートメントにより、または実行が関数メンバー本体の終端に達したこと) すべての位置において、確実に代入されている必要があります。これにより、関数メンバーが出力パラメーターで未定義の値を返さないことが保証されるため、コンパイラは、出力パラメーターとして変数を受け取る関数メンバー呼び出しが変数への代入と同じであると見なすことができます。
- *struct-type* のインスタンス コンストラクターの **this** 変数は、インスタンス コンストラクターが制御を返すすべての位置で確実に代入されている必要があります。

### 5.3.1 初期代入ありの変数

次のカテゴリの変数は、初期代入ありに分類されます。

- 静的変数
- クラス インスタンスのインスタンス変数
- 初期代入ありの構造体変数のインスタンス変数
- 配列要素
- 値パラメーター
- 参照パラメーター

- `catch` 句または `foreach` ステートメントで宣言された変数

### 5.3.2 初期代入なしの変数

次のカテゴリの変数は、初期代入なしに分類されます。

- 初期代入なしの構造体変数のインスタンス変数
- 構造体のインスタンス コンストラクターの `this` 変数を含む出力パラメーター
- `catch` 句または `foreach` ステートメントで宣言されたローカル変数を除くローカル変数

### 5.3.3 確実な代入を判断するための正確な規則

使用する各変数への代入が確実に行われていると判断するには、このセクションで説明する内容に対応するプロセスをコンパイラで使用する必要があります。

コンパイラは、初期代入なしの変数を 1 つ以上持つ関数メンバーの本体を 1 つずつ処理します。初期代入なしの変数  $v$  について、コンパイラは、次に示すそれぞれのポイントで、関数メンバーに含まれている変数  $v$  の "確実な代入の状態" を判断します。

- 各ステートメントの先頭
- 各ステートメントの終了点 (8.1 を参照)
- 別のステートメントまたはステートメントの終了点に制御を移すすべてのアーチ
- 各式の先頭
- 各式の末尾

$v$  の確実な代入の状態は、次のいずれかになります。

- 代入が確実に行われる。このポイントに至る可能性のあるすべての制御フローで、変数  $v$  に値が代入されていることを示します。
- 代入が確実に行われない。`bool` 型の式の末尾で、変数への代入が確実に行われていない場合、変数の状態は次のいずれかのサブ状態になる可能性があります。ただし、必ずしもこれらの状態になるとは限りません。
  - `true` 式の後では代入が確実に行われる。布尔式が `true` の場合は  $v$  への代入が確実に行われますが、布尔式が `false` の場合は必ずしも代入が行われるわけではないことを示します。
  - `false` 式の後では代入が確実に行われる。布尔式が `false` の場合は  $v$  への代入が確実に行われますが、布尔式が `true` の場合は必ずしも代入が行われるわけではないことを示します。

それぞれの位置での変数  $v$  の状態は、以下に示す規則に従って判断されます。

#### 5.3.3.1 ステートメントの一般的な規則

- 関数メンバー本体の先頭では、 $v$  への代入は確実には行われません。
- 到達不可能なステートメントの先頭では、 $v$  への代入は確実に行われます。
- その他すべてのステートメントの先頭での  $v$  の確実な代入の状態は、そのステートメントの先頭に制御を移すすべての制御フロー移動での  $v$  の確実な代入の状態をチェックして判断されます。このようなすべての制御フロー移動で  $v$  への代入が確実に行われている場合に限り、ステートメ

ントの先頭で  $v$  への代入が確実に行われます。移動する可能性のある制御フローの集合を判断するには、ステートメントの到達可能性のチェックの場合と同じ方法を使用します (8.1 を参照)。

- ブロック、`checked`、`unchecked`、`if`、`while`、`do`、`for`、`foreach`、`lock`、`using`、または `switch` のステートメントの終点での  $v$  の確実な代入の状態は、そのステートメントの終点に制御を移すすべての制御フロー移動での  $v$  の確実な代入の状態をチェックして判断されます。このようなすべての制御フロー移動で  $v$  への代入が確実に行われている場合は、ステートメントの終了点で  $v$  への代入が確実に行われます。それ以外の場合、ステートメントの終了点では  $v$  への代入が確実に行われません。移動する可能性のある制御フローの集合を判断するには、ステートメントの到達可能性のチェックの場合と同じ方法を使用します (8.1 を参照)。

### 5.3.3.2 ブロック ステートメント、`checked` ステートメント、および `unchecked` ステートメント

ブロック内のステートメントリストの最初のステートメント (ステートメントリストが空の場合は、ブロックの終了点) への制御移動での  $v$  の確実な代入の状態は、ブロック ステートメント、`checked` ステートメント、または `unchecked` ステートメントより前の  $v$  の確実な代入の状態と同じです。

### 5.3.3.3 式ステートメント

式  $expr$  で構成される式ステートメントの  $stmt$  は、以下の規則に従います。

- $expr$  の先頭では、 $v$  の確実な代入の状態は  $stmt$  の先頭と同じです。
- $expr$  の末尾で  $v$  への代入が確実に行われている場合、 $stmt$  の終了点では代入が確実に行われます。それ以外の場合、 $stmt$  の終了点では代入が確実に行われません。

### 5.3.3.4 宣言ステートメント

- $stmt$  が初期化子のない宣言ステートメントの場合、 $stmt$  の終了点では、 $v$  の確実な代入の状態は  $stmt$  の先頭と同じです。
- $stmt$  が初期化子を持つ宣言ステートメントの場合は、 $v$  の確実な代入の状態を判断するときに、 $stmt$  がステートメントリストと見なされます。ステートメントリストには、初期化子を持つ宣言ごとに 1 つの代入ステートメントが (宣言の順序に従って) 記述されています。

### 5.3.3.5 If ステートメント

次の形式の `if` ステートメント  $stmt$  は、以下の規則に従います。

`if ( expr ) then-stmt else else-stmt`

- $expr$  の先頭では、 $v$  の確実な代入の状態は  $stmt$  の先頭と同じです。
- $expr$  の末尾で  $v$  への代入が確実に行われている場合は `then-stmt` への制御フロー移動で、また `else` 句がない場合は `else-stmt` または  $stmt$  の終了点への制御フロー移動で、 $v$  への代入が確実に行われます。
- $expr$  の末尾の  $v$  が "true 式の後では代入が確実に行われる" 状態の場合、`then-stmt` への制御フロー移動では代入が確実に行われますが、`else` 句がない場合の `else-stmt` または  $stmt$  の終了点への制御フロー移動では代入が確実に行われません。
- $expr$  の末尾の  $v$  が "false 式の後では代入が確実に行われる" 状態の場合、`else-stmt` への制御フロー移動では代入が確実に行われますが、`then-stmt` への制御フロー移動では代入が確実に行われません。また、 $stmt$  の終了点で代入が確実に行われていると見なされるのは、`then-stmt` の終了点で代入が確実に行われている場合だけです。

- 上記以外の場合は、*then-stmt* または *else-stmt* への制御フロー移動、あるいは *else* 句がないときの *stmt* の終了点への制御フロー移動のいずれにおいても、*v*への代入は確実に行われないものと見なされます。

### 5.3.3.6 switch ステートメント

制御式 *expr* を持つ **switch** ステートメントの *stmt* は、以下の規則に従います。

- expr* の先頭での *v* の確実な代入の状態は、*stmt* の先頭の *v* の状態と同じです。
- 到達可能な **switch** ブロックのステートメントリストへの制御フロー移動での *v* の確実な代入の状態は、*expr* の末尾での *v* の確実な代入の状態と同じです。

### 5.3.3.7 While ステートメント

次の形式の **while** ステートメント *stmt* は、以下の規則に従います。

**while** ( *expr* ) *while-body*

- expr* の先頭では、*v* の確実な代入の状態は *stmt* の先頭と同じです。
- expr* の末尾で *v*への代入が確実に行われている場合は、*while-body* および *stmt* の終了点への制御フロー移動で *v*への代入が確実に行われます。
- expr* の末尾の *v* が "true 式の後では代入が確実に行われる" 状態の場合、*while-body* への制御フロー移動では代入が確実に行われますが、*stmt* の終了点では代入が確実に行われません。
- expr* の末尾の *v* が "false 式の後では代入が確実に行われる" 状態の場合、*stmt* の終了点への制御フロー移動では代入が確実に行われますが、*while-body* への制御フロー移動では代入が確実に行われません。

### 5.3.3.8 Do ステートメント

次の形式の **do** ステートメント *stmt* は、以下の規則に従います。

**do** *do-body* **while** ( *expr* ) ;

- stmt* の先頭から *do-body* への制御フロー移動での *v* の確実な代入の状態は、*stmt* の先頭での状態と同じです。
- expr* の先頭では、*v* の確実な代入の状態は *do-body* の終了点と同じです。
- expr* の末尾で *v*への代入が確実に行われている場合は、*stmt* の終了点への制御フロー移動で *v*への代入が確実に行われます。
- expr* の末尾の *v* が "false 式の後では代入が確実に行われる" 状態の場合は、*stmt* の終了点への制御フロー移動で *v*への代入が確実に行われます。

### 5.3.3.9 For ステートメント

次の形式の **for** ステートメントの確実な代入のチェックは、以下の規則に従います。

**for** ( *for-initializer* ; *for-condition* ; *for-iterator* ) *embedded-statement*

このステートメントは、次のように記述されていると見なされます。

```
{  
    for-initializer ;  
    while ( for-condition ) {  
        embedded-statement ;  
        for-iterator ;  
    }  
}
```

`for` ステートメントで `for-condition` を指定しなかった場合は、確実な代入の評価によって、上記の展開された形式で `for-condition` が `true` に置き換えられたと見なされます。

### 5.3.3.10 Break ステートメント、Continue ステートメント、および Goto ステートメント

`break`、`continue`、`goto` の各ステートメントによって発生した制御フロー移動での  $v$  の確実な代入の状態は、ステートメントの先頭での  $v$  の確実な代入の状態と同じです。

### 5.3.3.11 Throw ステートメント

次の形式のステートメント `stmt` は、以下の規則に従います。

```
throw expr ;
```

`expr` の先頭での  $v$  の確実な代入の状態は、`stmt` の先頭の  $v$  の確実な代入の状態と同じです。

### 5.3.3.12 Return ステートメント

次の形式のステートメント `stmt` は、以下の規則に従います。

```
return expr ;
```

- `expr` の先頭での  $v$  の確実な代入の状態は、`stmt` の先頭の  $v$  の確実な代入の状態と同じです。
- $v$  が出力パラメーターの場合は、次のいずれかで  $v$  への代入が確実に行われている必要があります。
  - `expr` の後
  - または、`return` ステートメントの外側にある `try-finally` または `try-catch-finally` の `finally` ブロックの末尾

次の形式のステートメントの `stmt` は、以下の規則に従います。

```
return ;
```

- $v$  が出力パラメーターの場合は、次のいずれかで  $v$  への代入が確実に行われている必要があります。
  - `stmt` の前
  - または、`return` ステートメントの外側にある `try-finally` または `try-catch-finally` の `finally` ブロックの末尾

### 5.3.3.13 Try-catch ステートメント

次の形式のステートメント `stmt` は、以下の規則に従います。

```
try try-block  
catch(...) catch-block-1  
...  
catch(...) catch-block-n
```

- *try-block* の先頭での  $v$  の確実な代入の状態は、*stmt* の先頭の  $v$  の確実な代入の状態と同じです。
- *catch-block-i* ( $i$  が存在する場合) の先頭での  $v$  の確実な代入の状態は、*stmt* の先頭の  $v$  の確実な代入の状態と同じです。
- *stmt* の終了点の  $v$  に代入が確実に行われるには、*try-block* の終了点およびすべての *catch-block-i* (各  $i$  について 1 から  $n$ ) で  $v$  が確実に代入されている場合だけです。

#### 5.3.3.14 Try-finally ステートメント

次の形式の **try** ステートメント *stmt* は、以下の規則に従います。

**try** *try-block finally* *finally-block*

- *try-block* の先頭での  $v$  の確実な代入の状態は、*stmt* の先頭の  $v$  の確実な代入の状態と同じです。
- *finally-block* の先頭での  $v$  の確実な代入の状態は、*stmt* の先頭の  $v$  の確実な代入の状態と同じです。
- 次の条件のうち少なくとも 1 つを満たす場合に限り、*stmt* の終了点の  $v$  には代入が確実に行われます。
  - *try-block* の終了点で、 $v$  への代入が確実に行われている。
  - *finally-block* の終了点で、 $v$  への代入が確実に行われている。

*try-block* 内で開始して *try-block* の外で終了する制御フロー移動 (**goto** ステートメントなど) が行われる場合でも、*finally-block* の終了点で  $v$  への代入が確実に行われているときは、その制御フロー移動でも  $v$  への代入が確実に行われると見なされます。 $v$  への代入が確実に行われると見なされるのは、この場合だけではありません。別の理由から、この制御フロー移動で  $v$  への代入が確実に行われると見なされることもあります。

#### 5.3.3.15 Try-catch-finally ステートメント

次の形式の **try-catch-finally** ステートメントの確実な代入の分析は、以下の規則に従います。

**try** *try-block*  
**catch**(...) *catch-block-1*  
 ...  
**catch**(...) *catch-block-n*  
**finally** *finally-block*

このステートメントは、次に示すように、**try-catch** ステートメントの外側に **try-finally** ステートメントがあると見なされます。

**try** {  
 try *try-block*  
**catch**(...) *catch-block-1*  
 ...  
**catch**(...) *catch-block-n*  
}  
**finally** *finally-block*

次の例は、**try** ステートメント (8.10 を参照) の異なるブロックが確実な代入に与える影響を示しています。 \r \h

```

class A
{
    static void F() {
        int i, j;
        try {
            goto LABEL;
            // neither i nor j definitely assigned
            i = 1;
            // i definitely assigned
        }
        catch {
            // neither i nor j definitely assigned
            i = 3;
            // i definitely assigned
        }
        finally {
            // neither i nor j definitely assigned
            j = 5;
            // j definitely assigned
        }
        // i and j definitely assigned
        LABEL:;
        // j definitely assigned
    }
}

```

### 5.3.3.16 Foreach ステートメント

次の形式の **foreach** ステートメント *stmt* は、以下の規則に従います。

**foreach** ( *type identifier* **in** *expr* ) *embedded-statement*

- *expr* の先頭での *v* の確実な代入の状態は、*stmt* の先頭の *v* の状態と同じです。
- *embedded-statement* または *stmt* の終了点への制御フロー移動での *v* の確実な代入の状態は、*expr* の末尾での *v* の状態と同じです。

### 5.3.3.17 Using ステートメント

次の形式の **using** ステートメント *stmt* は、以下の規則に従います。

**using** ( *resource-acquisition* ) *embedded-statement*

- *resource-acquisition* の先頭での *v* の確実な代入の状態は、*stmt* の先頭の *v* の状態と同じです。
- *embedded-statement* への制御フロー移動での *v* の確実な代入の状態は、*resource-acquisition* の末尾での *v* の状態と同じです。

### 5.3.3.18 Lock ステートメント

次の形式の **lock** ステートメント *stmt* は、以下の規則に従います。

**lock** ( *expr* ) *embedded-statement*

- *expr* の先頭での *v* の確実な代入の状態は、*stmt* の先頭の *v* の状態と同じです。
- *embedded-statement* への制御フロー移動での *v* の確実な代入の状態は、*expr* の末尾での *v* の状態と同じです。

### 5.3.3.19 yield ステートメント

次の形式の `yield return` ステートメント *stmt* は、以下の規則に従います。

```
yield return expr ;
```

- *expr* の先頭での *v* の確実な代入の状態は、*stmt* の先頭の *v* の状態と同じです。
- *stmt* の末尾での *v* の確実な代入の状態は、*expr* の末尾の *v* の状態と同じです。

`yield break` ステートメントは、確実な代入の状態には影響しません。

### 5.3.3.20 単純な式の一般的な規則

リテラル (7.6.1 を参照)、簡易名 (7.6.2 を参照)、メンバー アクセス式 (7.6.4 を参照)、インデックスなしのベース アクセス式 (7.6.8 を参照)、`typeof` 式 (7.6.11 を参照)、および既定値の式 (7.6.13 を参照) の各式には次の規則が適用されます。

- このような式の末尾での *v* の確実な代入の状態は、式の先頭での *v* の確実な代入の状態と同じです。

### 5.3.3.21 埋め込まれた式を持つ式の一般的な規則

次の規則は、かっこで囲まれた式 (7.6.3 を参照)、要素へのアクセス式 (7.6.6 を参照)、インデックス付きのベース アクセス式 (7.6.8 を参照)、インクリメント式とデクリメント式 (7.6.9、7.7.5 を参照)、キャスト式 (7.7.6 を参照)、+、-、~、\* の各単項式、+、-、\*、/、%，<<、>>、<、<=、>、>=、==、!=、`is`、`as`、&、|、^ の各二項式 (7.8、7.9、7.10、7.11 を参照)、複合代入式 (7.17.2 を参照)、`checked` 式と `unchecked` 式 (7.6.12F を参照)、配列作成式とデリゲート作成式 (7.6.10 を参照) に適用されます。

各式には、決められた順序で無条件に評価される 1 つ以上の部分式があります。たとえば、二項演算子 % は、演算子の左側を評価し、次に右側を評価します。インデックス演算では、まずインデックス付きの式を評価してから、左から右へ順にインデックス式を 1 つずつ評価します。この順序で評価される部分式 *expr*<sub>1</sub>、*expr*<sub>2</sub>、...、*expr*<sub>*n*</sub> を持つ式 *expr* は、次の規則に従います。

- *expr*<sub>1</sub> の先頭での *v* の確実な代入の状態は、*expr* の先頭での確実な代入の状態と同じです。
- *expr*<sub>*i*</sub> (*i* は 1 より大きい) の先頭での *v* の確実な代入の状態は、*expr*<sub>*i-1*</sub> の末尾での確実な代入の状態と同じです。
- *expr* の末尾での *v* の確実な代入の状態は、*expr*<sub>*n*</sub> の末尾での確実な代入の状態と同じです。

### 5.3.3.22 呼び出し式とオブジェクト作成式

呼び出し式の *expr* は次の形式になります。

```
primary-expression ( arg1 , arg2 , ... , argn )
```

オブジェクト作成式は次の形式になります。

```
new type ( arg1 , arg2 , ... , argn )
```

- 呼び出し式の場合、*primary-expression* より前の *v* の確実な代入の状態は、*expr* より前の *v* の状態と同じです。
- 呼び出し式の場合、*arg*<sub>1</sub> より前の *v* の確実な代入の状態は、*primary-expression* より後の *v* の状態と同じです。

- オブジェクト作成式の場合、 $arg_1$  より前の  $v$  の確実な代入の状態は、 $expr$  より前の  $v$  の状態と同じです。
- 引数  $arg_i$  の場合、 $arg_i$  の後の  $v$  の確実な代入の状態は、通常の式の規則によって判断され、**ref** 修飾子や **out** 修飾子は無視されます。
- 各  $arg_i$  引数の  $i$  が 1 を超える場合、 $arg_i$  より前の  $v$  の確実な代入の状態は  $arg_{i+1}$  よりも後の  $v$  の状態と同じです。
- 任意の引数で、変数  $v$  を引数 **out** ("out  $v$ " の形式の引数) として渡した場合、 $expr$  の後の  $v$  には代入が確実に行われます。それ以外の場合、 $expr$  の後の  $v$  の状態は、 $arg_n$  の後の  $v$  の状態と同じです。
- 配列初期化子 (7.6.10.4 を参照)、オブジェクト初期化子 (7.6.10.2 を参照)、コレクション初期化子 (7.6.10.3 を参照)、および匿名オブジェクト初期化子 (7.6.10.6 を参照) の場合、確実な代入の状態は、これらの構成要素の定義の基となる拡張形式によって決定されます。

### 5.3.3.23 簡単な代入式

$w = expr\text{-}rhs$  の形式の式  $expr$  は、以下の規則に従います。

- $expr\text{-}rhs$  より前の  $v$  の確実な代入の状態は、 $expr$  より前の  $v$  の確実な代入の状態と同じです。
- $w$  と  $v$  の変数が同じ場合、 $expr$  の後の  $v$  には代入が確実に行われます。それ以外の場合、 $expr$  の後の  $v$  の確実な代入の状態は、 $expr\text{-}rhs$  の後の  $v$  の確実な代入の状態と同じです。

### 5.3.3.24 && 式

$expr\text{-}first \&\& expr\text{-}second$  の形式の式  $expr$  は、以下の規則に従います。

- $expr\text{-}first$  より前の  $v$  の確実な代入の状態は、 $expr$  より前の  $v$  の確実な代入の状態と同じです。
- $expr\text{-}first$  の後の  $v$  に代入が確実に行われているか、または "true 式の後では代入が確実に行われる" 状態の場合、 $expr\text{-}second$  の前の  $v$  には代入が確実に行われます。それ以外の場合、代入は確実に行われません。
- $expr$  の後の  $v$  の確実な代入の状態は、次の条件に基づいて判断されます。
  - $expr\text{-}first$  が値 **false** を持つ定数式であれば、 $expr$  の後の  $v$  の確実な代入の状態は  $expr\text{-}first$  の後の  $v$  の確実な代入の状態と同じです。
  - それ以外の場合、 $expr\text{-}first$  の後の  $v$  に代入が確実に行われている場合、 $expr$  の後の  $v$  には代入が確実に行われます。
  - それ以外の場合、 $expr\text{-}second$  の後の  $v$  に代入が確実に行われており、 $expr\text{-}first$  の後の  $v$  が "false 式の後では代入が確実に行われる" 状態ならば、 $expr$  の後の  $v$  には代入が確実に行われます。
  - それ以外の場合、 $expr\text{-}second$  の後の  $v$  に代入が確実に行われているか、または "true 式の後では代入が確実に行われる" 状態ならば、 $expr$  の後の  $v$  は "true 式の後では代入が確実に行われる" 状態になります。
  - それ以外の場合、 $expr\text{-}first$  の後の  $v$  が "false 式の後では代入が確実に行われる" 状態であり、 $expr\text{-}second$  の後の  $v$  が "false 式の後では代入が確実に行われる" 状態ならば、 $expr$  の後の  $v$  は "false 式の後では代入が確実に行われる" 状態になります。

- それ以外の場合、*expr* の後の *v* への代入が確実に行われるとは限りません。

次に例を示します。

```
class A
{
    static void F(int x, int y) {
        int i;
        if (x >= 0 && (i = y) >= 0) {
            // i definitely assigned
        }
        else {
            // i not definitely assigned
        }
        // i not definitely assigned
    }
}
```

この例の各メソッドの変数 *i* は、*if* ステートメントの一方の埋め込みステートメントでは確実に代入されていると見なされますが、もう一方の埋め込みステートメントでは確実に代入されていると見なされません。*F* メソッドの *if* ステートメントの場合、最初の埋め込みステートメントでは、この埋め込みステートメントの実行より前に式 (*i = y*) が必ず実行されるため、変数 *i* は確実に代入されています。それに対して、2 番目の埋め込みステートメントでは、*x >= 0* の検査が *false* になって変数 *i* への代入が行われない場合があるため、変数 *i* は確実に代入されていません。

### 5.3.3.25 || 式

*expr-first* || *expr-second* の形式の式 *expr* は、以下の規則に従います。

- *expr-first* より前の *v* の確実な代入の状態は、*expr* より前の *v* の確実な代入の状態と同じです。
- *expr-first* の後の *v* に代入が確実に行われているか、または "false 式の後では代入が確実に行われる" 状態の場合、*expr-second* の前の *v* には代入が確実に行われます。それ以外の場合、代入は確実に行われません。
- *expr* の後の *v* の確実な代入のステートメントは、次の条件に基づいて判断されます。
  - *expr-first* が値 **true** を持つ定数式であれば、*expr* の後の *v* の確実な代入の状態は *expr-first* の後の *v* の確実な代入の状態と同じです。
  - それ以外の場合、*expr-first* の後の *v* に代入が確実に行われている場合、*expr* の後の *v* には代入が確実に行われます。
  - それ以外の場合、*expr-second* の後の *v* に代入が確実に行われ、*expr-first* の後の *v* が "true 式の後では代入が確実に行われる" 状態ならば、*expr* の後の *v* には代入が確実に行われます。
  - それ以外の場合、*expr-second* の後の *v* に代入が確実に行われているか、または "false 式の後では代入が確実に行われる" 状態のときは、*expr* の後の *v* は "false 式の後では代入が確実に行われる" 状態になります。
  - それ以外の場合、*expr-first* の後の *v* が "true 式の後では代入が確実に行われる" 状態であり、*expr-second* の後の *v* が "true 式の後では代入が確実に行われる" 状態ならば、*expr* の後の *v* は "true 式の後では代入が確実に行われる" 状態になります。
  - それ以外の場合、*expr* の後の *v* への代入が確実に行われるとは限りません。

次に例を示します。

```

class A
{
    static void G(int x, int y) {
        int i;
        if (x >= 0 || (i = y) >= 0) {
            // i not definitely assigned
        }
        else {
            // i definitely assigned
        }
        // i not definitely assigned
    }
}

```

この例の各メソッドの変数 *i* は、*if* ステートメントの一方の埋め込みステートメントでは確実に代入されていると見なされますが、もう一方の埋め込みステートメントでは確実に代入されていると見なされません。*G* メソッドの *if* ステートメントの場合、第 2 の埋め込みステートメントでは、この埋め込みステートメントの実行より前に式 (*i* = *y*) が必ず実行されるため、変数 *i* は確実に代入されています。それに対して、最初の埋め込みステートメントでは、*x* >= 0 の検査が true になって変数 *i* への代入が行われない場合があるため、変数 *i* は確実に代入されていません。

### 5.3.3.26 !式

*! expr-operand* の形式の式 *expr* は、以下の規則に従います。

- *expr-operand* より前の *v* の確実な代入の状態は、*expr* より前の *v* の確実な代入の状態と同じです。
- *expr* の後の *v* の確実な代入の状態は、次の条件に基づいて判断されます。
  - *expr-operand* の後の *v* に代入が確実に行われている場合、*expr* の後の *v* には代入が確実に行われます。
  - *expr-operand* の後の *v* に代入が確実に行われていない場合、*expr* の後の *v* には代入が確実に行われません。
  - *expr-operand* の後の *v* の状態が "false 式の後では代入が確実に行われる" の場合、*expr* の後の *v* は "true 式の後では代入が確実に行われる" 状態になります。
  - *expr-operand* の後の *v* の状態が "true 式の後では代入が確実に行われる" の場合、*expr* の後の *v* は "false 式の後では代入が確実に行われる" 状態になります。

### 5.3.3.27 ?? 式

*expr-first ?? expr-second* の形式の式 *expr* は、以下の規則に従います。

- *expr-first* より前の *v* の確実な代入の状態は、*expr* より前の *v* の確実な代入の状態と同じです。
- *expr-second* より前の *v* の確実な代入の状態は、*expr-first* より後の *v* の確実な代入の状態と同じです。
- *expr* の後の *v* の確実な代入のステートメントは、次の条件に基づいて判断されます。
  - *expr-first* が値 null を持つ定数式 (7.19 を参照) であれば、*expr* の後の *v* の状態は *expr-second* の後の *v* の状態と同じです。
- それ以外の場合、*expr* の後の *v* の状態は、*expr-first* の後の *v* の確実な代入の状態と同じです。

### 5.3.3.28 ?: 式

*expr-cond* ? *expr-true* : *expr-false* の形式の式 *expr* は、以下の規則に従います。

- *expr-cond* より前の *v* の確実な代入の状態は、*expr* より前の *v* の状態と同じです。
- 次の条件のうち少なくとも 1 つを満たす場合に限り、*expr-true* の前の *v* には代入が確実に行われます。
  - *expr-cond* が値 **false** を持つ定数式である。
  - *expr-cond* の後の *v* に代入が確実に行われている、または "true 式の後では代入が確実に行われる"。
- 次の条件のうち少なくとも 1 つを満たす場合に限り、*expr-false* の前の *v* には代入が確実に行われます。
  - *expr-cond* が値 **true** を持つ定数式である
- *expr-cond* の後の *v* に代入が確実に行われている、または "false 式の後では代入が確実に行われる"。
- *expr* の後の *v* の確実な代入の状態は、次の条件に基づいて判断されます。
  - *expr-cond* が値 **true** を持つ定数式 (7.19 を参照) であれば、*expr* の後の *v* の状態は *expr-true* の後の *v* の状態と同じです。
  - それ以外の場合、*expr-cond* が値 **false** を持つ定数式 (7.19 を参照) であれば、*expr* の後の *v* の状態は *expr-false* の後の *v* の状態と同じです。
  - それ以外の場合、*expr-true* の後の *v* に代入が確実に行われ、*expr-false* の後の *v* に代入が確実に行われている場合、*expr* の後の *v* には代入が確実に行われます。
  - それ以外の場合、*expr* の後の *v* への代入が確実に行われるとは限りません。

### 5.3.3.29 匿名関数

本体 (*block* または *expression*) *body* を持つ *lambda-expression* または *anonymous-method-expression* *expr* は、次の規則に従います。

- *body* より前の外部変数 *v* の確実な代入の状態は、*expr* より前の *v* の状態と同じです。つまり、外部変数の確実な代入の状態は、匿名関数のコンテキストから継承されます。
- *expr* より後の外部変数 *v* の確実な代入の状態は、*expr* より前の *v* の状態と同じです。

次の例を参照してください。

```
delegate bool Filter(int i);
void F() {
    int max;
    // Error, max is not definitely assigned
    Filter f = (int n) => n < max;
    max = 5;
    DoWork(f);
}
```

これは、匿名関数が宣言された位置で *max* が確実に代入されていないため、コンパイル エラーになります。次の例を参照してください。

```
delegate void D();
```

```
void F() {
    int n;
    D d = () => { n = 1; };
    d();
    // Error, n is not definitely assigned
    Console.WriteLine(n);
}
```

これも、匿名関数の `n` の代入が匿名関数の外側の `n` の確実な代入の状態に影響しないため、コンパイルエラーになります。

## 5.4 変数参照

*variable-reference* は、変数として分類される *expression* です。*variable-reference* は、現在の値をフェッチしたり新しい値を格納したりするためにアクセスできる格納場所を表しています。

```
variable-reference:  
expression
```

C および C++ では、*variable-reference* は "左辺値" と呼ばれています。

## 5.5 変数参照の分割不能性

`bool`、`char`、`byte`、`sbyte`、`short`、`ushort`、`uint`、`int`、`float` の各データ型および参照型の読み取りと書き込みは分割不可能です。また、上記の型を基になる型に持つ列挙型の読み取りと書き込みも分割不可能です。`long`、`ulong`、`double`、`decimal` などのその他のデータ型、およびユーザー定義型の読み取りと書き込みは、分割不可能であることが保証されていません。この目的で設計されたライブラリ関数を除いて、インクリメントやデクリメントの場合などの分割不可能な読み取り/変更/書き込みは保証されていません。



# 6. 変換

**変換**を行うことで、式を特定の型として扱うことができます。変換により、特定の型の式が異なる型として扱われたり、型のない式が型を取得したりする場合があります。変換には**暗黙**の変換と**明示的**な変換があり、これによって、明示的なキャストが必要かどうかが決まります。たとえば、`int` 型から `long` 型への変換は暗黙的に行われるため、`int` 型の式は暗黙的に `long` 型として扱うことができます。反対に、`long` 型から `int` 型への変換は明示的であり、明示的なキャストが必要です。

```
int a = 123;
long b = a;    // implicit conversion from int to long
int c = (int) b; // explicit conversion from long to int
```

言語によって定義されている変換もあります。プログラムでは、独自の型変換も定義できます(6.4 を参照)。

言語では、式から型への変換と、型から型への変換が定義されます。型からの変換は、その型を持つすべての式に適用されます。

```
enum Color { Red, Blue, Green }
Color c0 = 0;      // The expression 0 converts implicitly to enum types
Color c1 = (Color)1; // other int expressions need explicit conversion
```

## 6.1 暗黙の型変換

以下の型変換は、暗黙の変換に分類されます。

- 恒等変換
- 暗黙の数値変換
- 暗黙の列挙変換
- 暗黙の `null` 許容変換
- `null` リテラル変換
- 暗黙の参照変換
- ポックス化変換
- 暗黙の動的変換
- 暗黙の定数式変換
- ユーザー定義の暗黙の変換
- 匿名関数の変換
- メソッドのグループの変換

暗黙的な変換は、関数メンバー呼び出し(7.5.4 を参照)、キャスト式(7.7.6 を参照)、代入など(7.17 を参照)、さまざまな状況で実行します。

定義済みの暗黙の型変換は常に成功し、例外がスローされることはありません。適切にデザインされたユーザー定義の暗黙の型変換は、この特徴を同様に備えている必要があります。

変換に関しては、`object` 型と `dynamic` 型は同等と見なされます。

ただし、動的な変換 (6.1.8 および 6.2.6 を参照) は、`dynamic` 型 (4.7 を参照) の式に対してのみ適用されます。

### 6.1.1 恒等変換

恒等変換は、任意の型を同じ型に変換します。この変換は、必要な型を既に備えているエンティティについて、その型に変換可能であることを示すことができるようになります。

`object` と `dynamic` は同等と考えられるため、`object` と `dynamic` の間、およびすべての `dynamic` を `object` で置換する場合は同じ構築された型の間に、恒等変換があります。

In most cases an identity conversion has no effect at runtime. However, since floating point operations may be performed at higher precision than prescribed by their type (§ 4.1.6), assignment of their results may result in a loss of precision, and explicit casts are guaranteed to reduce precision to what is prescribed by the type.

### 6.1.2 暗黙の数値変換

暗黙の数値変換は次のとおりです。

- `sbyte` から `short`、`int`、`long`、`float`、`double` または `decimal` への変換。
- `byte` から `short`、`ushort`、`int`、`uint`、`long`、`ulong`、`float`、`double` または `decimal` への変換。
- `short` から `int`、`long`、`float`、`double` または `decimal` への変換。
- `ushort` から `int`、`uint`、`long`、`ulong`、`float`、`double` または `decimal` への変換。
- `int` から `long`、`float`、`double`、または `decimal` への変換。
- `uint` から `long`、`ulong`、`float`、`double` または `decimal` への変換。
- `long` から `float`、`double` または `decimal` への変換。
- `ulong` から `float`、`double` または `decimal` への変換。
- `char` から `ushort`、`int`、`uint`、`long`、`ulong`、`float`、`double`、または `decimal` への変換。
- `float` から `double` への変換。

`int`、`uint`、`long`、または `ulong` から `float` への変換、および `long` または `ulong` から `double` への変換では、有効桁数が桁落ちすることがあります。絶対値は損なわれません。これ以外の暗黙の数値変換では、どのような情報も失われません。

`char` 型への暗黙の型変換は行われないため、他の整数型の値が `char` 型に自動的に変換されることはありません。

### 6.1.3 暗黙の列挙値変換

暗黙の列挙値変換により、`decimal-integer-literal` の `0` (または、`0L` など) は、任意の `enum-type` および基になる型が `enum-type` である任意の `nullable-type` に変換できます。後者の場合、変換の評価は、基になる `enum-type` に変換して結果をラップすることにより行われます (4.1.10 を参照)。

#### 6.1.4 暗黙の null 許容変換

null 非許容の値型を操作する定義済みの暗黙の変換は、それらの型の null 許容形式に対しても使用できます。null 非許容の値型  $S$  を null 非許容の値型  $T$  に変換する定義済みの暗黙の恒等変換および数値変換のそれぞれについて、次のような暗黙の null 許容変換が存在します。

- $S?$  から  $T?$  への暗黙の型変換です。
- $S$  から  $T?$  への暗黙の型変換です。

$S$  から  $T$  への基になる変換に基づく暗黙の null 許容変換の評価は、次のように処理されます。

- $S?$  から  $T?$  への null 許容変換の場合は、次のように処理されます。
  - ソース値が null (`HasValue` プロパティが `false`) の場合、結果は型  $T?$  の null 値になります。
  - それ以外の場合は、変換を  $S?$  から  $S$  へのラップ解除として評価し、引き続き  $S$  から  $T$  への基になる変換を実行して、 $T$  から  $T?$  へのラップ (4.1.10 を参照) を実行します。
- $S$  から  $T?$  への null 許容変換の場合は、変換を  $S$  から  $T$  への基になる変換として評価し、引き続き  $T$  から  $T?$  へのラップを実行します。

#### 6.1.5 null リテラル変換

null リテラルから任意の null 許容型への暗黙の変換が存在します。この変換は、指定した null 訸容型の null 値 (4.1.10 を参照) を生成します。

#### 6.1.6 暗黙の参照変換

暗黙の参照変換は次のとおりです。

- 任意の `reference-type` から `object` と `dynamic` への変換。
- 任意の `class-type`  $S$  から、任意の `class-type`  $T$  への変換。ただし、 $S$  が  $T$  から派生している場合。
- 任意の `class-type`  $S$  から任意の `interface-type`  $T$  への変換。ただし、 $S$  が  $T$  を実装している場合。
- 任意の `interface-type`  $S$  から、任意の `interface-type`  $T$  への変換。ただし、 $S$  が  $T$  から派生している場合。
- 要素型が  $S_E$  の `array-type`  $S$  から要素型が  $T_E$  の `array-type`  $T$  への変換。ただし、以下のすべてが満たされる場合。
  - $S$  と  $T$  は要素型だけが異なる。つまり、 $S$  と  $T$  の次元数が同じである。
  - ここで、 $S_E$  と  $T_E$  はいずれも `reference-type` です。
  - $S_E$  から  $T_E$  への暗黙の参照変換が存在する。
- 任意の `array-type` から `System.Array` およびそれが実装しているインターフェイスへの変換。
- 1 次元配列型  $S[]$  から `System.Collections.Generic.IList<T>` およびその基本インターフェイスへの変換。ただし  $S$  から  $T$  への暗黙の恒等変換または参照変換がある場合。
- 任意の `delegate-type` から `System.Delegate` およびそれが実装しているインターフェイスへの変換。
- null リテラルから、任意の `reference-type` への変換。

- 任意の *reference-type* から *reference-type T* への変換。ただし、*reference-type T<sub>0</sub>* への暗黙の恒等変換または参照変換があり、*T<sub>0</sub>* に *T* への恒等変換がある場合。
- 任意の *reference-type* からインターフェイスまたはデリゲート型 *T* への変換。ただし、インターフェイスまたはデリゲート型 *T<sub>0</sub>* への暗黙の恒等変換または参照変換があり、*T<sub>0</sub>* が *T* への変性変換(13.1.3.2 を参照)である場合。
- 参照型であることがわかっている型パラメーターを使用する暗黙の変換。型パラメーターを使用した暗黙の変換の詳細については、6.1.10 を参照してください。

暗黙の参照変換は、常に成功することが証明されている *reference-type* 間の変換であるため、実行時のチェックは必要ありません。

暗黙または明示による参照変換では、変換されるオブジェクトの参照 ID が変わることはありません。つまり、参照変換により参照の型が変わることはありますが、参照されるオブジェクトの型または値が変化することはありません。

### 6.1.7 ボックス化変換

ボックス化変換では、*value-type* から参照型への暗黙の変換ができます。任意の *non-nullable-value-type* から *object* および *dynamic* へのボックス化変換、*System.ValueType* へのボックス化変換、および *non-nullable-value-type* によって実装される任意の *interface-type* へのボックス化変換が存在します。さらに、*enum-type* は *System.Enum* 型に変換できます。

*nullable-type* から参照型へのボックス化変換は、基になる *non-nullable-value-type* から参照型へのボックス化変換が存在する場合にのみ存在します。

値型にインターフェイス型 *I<sub>0</sub>* へのボックス化変換があり、*I<sub>0</sub>* に *I* への恒等変換がある場合、値型にはインターフェイス型 *I* へのボックス化変換があります。

値型にインターフェイス型またはデリゲート型 *I<sub>0</sub>* へのボックス化変換があり、*I<sub>0</sub>* が *I* への変性変換(13.1.3.2 を参照)である場合、値型にはインターフェイス型 *I* へのボックス化変換があります。

*non-nullable-value-type* の値のボックス化では、オブジェクトインスタンスの割り当て、およびそのインスタンスへの *value-type* の値のコピーが行われます。構造体は *System.ValueType* 型に変換できます。この型はすべての構造体の基底クラスであるためです(11.3.2 を参照)。

*nullable-type* の値のボックス化は、次のように処理されます。

- ソース値が *null* (*HasValue* プロパティが *false*) の場合、結果はターゲットの型の *null* 参照になります。
- それ以外の場合、結果は、ソース値のラップ解除およびボックス化によって生成されるボックス化された *T* への参照になります。

ボックス化変換については、4.3.1 で詳細に説明します。

### 6.1.8 暗黙の動的変換

*dynamic* 型の式から任意の型 *T* への暗黙の動的変換が存在します。変換は動的にバインド(7.2.2 を参照)されます。つまり、式の実行時の型から *T* への暗黙の型変換が、実行時に検索されます。変換が見つからない場合は、実行時例外がスローされます。

この暗黙の型変換は、暗黙の型変換では例外が発生しないという 0 の最初にある説明に一見違反しています。しかし、例外の原因になるのは、変換自体ではなく、変換の検索です。動的バインディング

を使用する場合は、実行時例外が発生するリスクが常にあります。変換の動的バインディングが望ましくない場合は、式を最初に `object` に変換してから、目的の型に変換できます。

次の例では、暗黙の動的変換を示します。

```
object o = "object";
dynamic d = "dynamic";

string s1 = o; // Fails at compile-time - no conversion exists
string s2 = d; // Compiles and succeeds at run-time
int i      = d; // Compiles but fails at run-time - no conversion exists
```

`s2` および `i` への代入では、どちらも暗黙の動的変換が使用されます。動的変換では、操作のバインディングが実行時まで延期されます。実行時には、`d` の実行時の型(`string`)からターゲットの型への暗黙の型変換が検索されます。`string` への変換は見つかりますが、`int` への変換は見つかりません。

### 6.1.9 暗黙の定数式変換

暗黙の定数式変換を使用すると、次の変換を行うことができます。

- `int` 型の *constant-expression* (7.19 を参照) は、*constant-expression* の値が変換後の型の範囲内である場合、`sbyte`、`byte`、`short`、`ushort`、`uint`、または `ulong` の型に変換できます。
- `long` 型の *constant-expression* は、*constant-expression* の値が負でない場合、`ulong` 型に変換できます。

### 6.1.10 型パラメーターを使用する暗黙の変換

任意の型パラメーター `T` に対して、次のような暗黙の型変換が存在します。

- `T` からその実質的な基底クラス `C` へ、`T` から `C` の任意の基底クラスへ、および `T` から `C` によって実装される任意のインターフェイスへの変換。実行時には、`T` が値型の場合は、変換はボックス化変換として実行されます。`T` が値型でない場合、変換は暗黙の参照変換または恒等変換として実行されます。
- `T` から `T` の有効なインターフェイスセットのインターフェイス型 `I` へ、および `T` から `I` の任意の基本インターフェイスへの変換。実行時には、`T` が値型の場合は、変換はボックス化変換として実行されます。`T` が値型でない場合、変換は暗黙の参照変換または恒等変換として実行されます。
- `T` が型パラメーター `U` に依存する場合は、`T` から `U` への変換 (10.1.5 を参照)。実行時、`U` が値型の場合は、`T` と `U` が必ず同じ型となり、変換は実行されません。`T` が値型の場合は、変換はボックス化変換として実行されます。`T` が値型でない場合、変換は暗黙の参照変換または恒等変換として実行されます。
- `T` が参照型であることがわかっている場合は、`null` リテラルから `T` への変換。
- `T` から参照型 `I` への変換。参照型 `S0` への暗黙の型変換があり、`S0` に `S` への恒等変換がある場合。実行時に、`S0` への変換と同じ方法で、変換が実行されます。
- `T` からインターフェイス型 `I` への変換。インターフェイス型またはデリゲート型 `I0` への暗黙の型変換があり、`I0` が `I` への変性変換 (13.1.3.2 を参照) である場合。実行時には、`T` が値型の場合は、変換はボックス化変換として実行されます。`T` が値型でない場合、変換は暗黙の参照変換または恒等変換として実行されます。

`T` が参照型 (10.1.5 を参照) であることがわかっている場合は、上記のすべての変換は暗黙の参照変換に分類されます (6.1.6 を参照)。`T` が参照型でないことがわかっている場合は、上記の変換はボックス化変換に分類されます (6.1.7 を参照)。

### 6.1.11 ユーザー定義の暗黙の変換

ユーザー定義の暗黙の変換は、順に、標準暗黙変換 (省略可能)、ユーザー定義の暗黙による変換演算子の実行、別の標準暗黙変換 (省略可能) で構成されます。ユーザー定義の暗黙変換の評価に関する厳密な規則については、6.4.4 で説明します。

### 6.1.12 匿名関数の変換とメソッド グループの変換

匿名関数とメソッド グループは、それ自体では型を持ちませんが、デリゲート型または式ツリー型に暗黙的に変換できます。匿名関数の変換の詳細については 0 を、メソッド グループの変換の詳細については 6.6 を参照してください。

## 6.2 明示的な変換

以下の型変換は、明示的な変換に分類されます。

- すべての暗黙の変換
- 明示的な数値変換
- 明示的な列挙値変換
- 明示的な null 許容変換
- 明示的な参照変換
- 明示的なインターフェイス変換
- ボックス化解除変換
- 明示的な動的変換
- ユーザー定義の明示的な変換

明示的な変換は、キャスト式 (7.7.6 を参照) の中で使用できます。

明示的な変換には、暗黙の変換がすべて含まれます。これは、冗長なキャスト式が許されることを意味します。

暗黙の変換でない明示的な変換とは、常に成功することは証明されていない変換、情報が失われる可能性のある変換、および明示的な表記が必要なほど異なっている型の種類の間で行われる変換です。

### 6.2.1 明示的な数値変換の一覧表

明示的な数値変換は、*numeric-type* から別の *numeric-type* への変換であり、暗黙の数値変換 (0 を参照) が存在していないものです。

- `sbyte` から `byte`、`ushort`、`uint`、`ulong` または `char`  $\sim$ 。
- `byte` から `sbyte` および `char` への変換。
- `short` から `sbyte`、`byte`、`ushort`、`uint`、`ulong` または `char`  $\sim$ 。
- `ushort` から `sbyte`、`byte`、`short`、または `char`  $\sim$ 。

- `int` から `sbyte`、`byte`、`short`、`ushort`、`uint`、`ulong` または `char` へ。
- `uint` から `sbyte`、`byte`、`short`、`ushort`、`int` または `char` へ。
- `long` から `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`ulong`、または `char` へ。
- `ulong` から `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、または `char` へ。
- `char` から `sbyte`、`byte` または `short` へ。
- `float` から `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、または `decimal` への変換。
- `double` から `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、または `decimal` への変換。
- `decimal` から `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、または `double` への変換。

明示的な変換には、暗黙の数値変換と明示的な数値変換がすべて含まれているため、キャスト式 (7.7.6 を参照) を使用して、任意の *numeric-type* を他の任意の *numeric-type* に常に変換できます。

明示的な数値変換を行うと、情報が失われたり、例外がスローされたりする可能性があります。明示的な数値変換は、次のように処理されます。

- ある整数型から別の整数型への変換では、処理は、変換が行われるオーバーフロー チェックのコンテキスト (7.6.12 を参照) に依存します。
  - `checked` コンテキストでは、変換前のオペランドの値が変換先の型の範囲に収まる場合は変換が成功し、収まらない場合は `System.OverflowException` がスローされます。
  - `unchecked` コンテキストでは、変換は常に成功し、続けて以下の処理が行われます。
    - 変換元の型が変換先の型より大きい場合は、変換前の値から "余分な" 上位ビットが破棄されて切り詰められます。その後、結果は変換先の型の値として扱われます。
    - 変換元の型が変換先の型より小さい場合は、変換先の型と同じサイズになるように、変換前の値に対して符号拡張またはゼロ拡張が行われます。変換元の型が符号付きの場合は符号拡張が使われて、符号なしの場合はゼロ拡張が使われます。その後、結果は変換先の型の値として扱われます。
    - 変換元の型と変換先の型のサイズが同じ場合は、変換前の値は変換先の型の値として扱われます。
- `decimal` から整数型への変換では、変換前の値はゼロに向かって最も近い整数値に丸められて、この整数値が変換の結果になります。結果の整数値が変換先の型の範囲を超えた場合は、`System.OverflowException` がスローされます。
- `float` または `double` から整数型への変換では、処理は、変換が行われるオーバーフロー チェックのコンテキスト (7.6.12 を参照) に依存します。
  - `checked` コンテキストでは、変換は次のように処理されます。
    - オペランドの値が `NaN` または無限の場合は、`System.OverflowException` がスローされます。

- それ以外の場合、変換前のオペラントはゼロに向かって最も近い整数値に丸められます。この整数値が変換先の型の範囲内である場合は、この値が変換の結果になります。
- それ以外の場合は、`System.OverflowException` がスローされます。
- `unchecked` コンテキストでは、変換は常に成功し、続けて以下の処理が行われます。
  - オペラントの値が非数 (NaN) または無限の場合は、変換先の型の未指定値が変換の結果になります。
  - それ以外の場合、変換前のオペラントはゼロに向かって最も近い整数値に丸められます。この整数値が変換先の型の範囲内である場合は、この値が変換の結果になります。
  - それ以外の場合は、変換先の型の未指定値が変換の結果になります。
- `double` から `float` への変換では、`double` 値は最も近い `float` 値に丸められます。`double` の値が小さすぎて `float` で表すことができない場合、結果は正のゼロまたは負のゼロになります。`double` 値が大きすぎて `float` で表現できない場合、結果は正の無限大または負の無限大になります。オペラントが `double` の場合は、結果も NaN になります。
- `float` または `double` から `decimal` への変換では、変換前の値が `decimal` 表記に変換され、必要に応じて小数第 28 位までの近似値に丸められます (4.1.7 を参照)。変換前の値が小さすぎて `decimal` で表すことができない場合、結果はゼロになります。変換前の値が非数 (NaN) や無限大の場合、または大きすぎて `decimal` で表すことができない場合は、`System.OverflowException` がスローされます。
- `decimal` から `float` または `double` への変換では、`decimal` 値は最も近い `double` または `float` 値に丸められます。この変換では精度が損なわれる場合がありますが、例外がスローされることはありません。

### 6.2.2 明示的な列挙値変換

明示的な列挙値の変換は次のとおりです。

- `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、または `decimal` から任意の `enum-type` への変換。
- 任意の `enum-type` から `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、または `decimal` への変換。
- 任意の `enum-type` から他の任意の `enum-type` への変換

2つの型の間の明示的な列挙値変換は、関係するすべての `enum-type` を列挙型の基になっている `enum-type` として扱った後、結果の型の間で暗黙または明示的な数値変換を実行することで処理されます。たとえば、`enum-type E` の基になっている型が `int` であるとすると、`E` から `byte` への変換は `int` から `byte` への明示的な数値変換 (6.2.1 を参照) として処理され、`byte` から `E` への変換は、`byte` から `int` への暗黙的な数値変換 (0 を参照) として処理されます。

### 6.2.3 明示的な null 許容変換

明示的な `null` 許容変換では、`null` 非許容の値型を操作する定義済みの明示的変換を、それらの型の `null` 許容形式に対して使用できます。`null` 非許容の値型 `S` を `null` 非許容の値型 `T` (6.1.1、6.1.2、6.1.3、6.2.1、および 6.2.2 を参照) に変換する定義済みの明示的変換のそれについて、次のような `null` 許容変換が存在します。

- $S?$  から  $T?$  への明示的な変換。
- $S$  から  $T?$  への明示的変換。
- $S?$  から  $T$  への明示的変換。

$S$  から  $T$  への基になる変換に基づく null 許容変換の評価は、次のように処理されます。

- $S?$  から  $T?$  への null 許容変換の場合は、次のように処理されます。
  - ソース値が null (`HasValue` プロパティが false) の場合、結果は型  $T?$  の null 値になります。
  - それ以外の場合は、変換を  $S?$  から  $S$  へのラップ解除として評価し、引き続き  $S$  から  $T$  への基になる変換を実行して、 $T$  から  $T?$  へのラップを実行します。
- $S$  から  $T?$  への null 許容変換の場合は、変換を  $S$  から  $T$  への基になる変換として評価し、引き続き  $T$  から  $T?$  へのラップを実行します。
- $S?$  から  $T$  への null 許容変換の場合は、変換を  $S?$  から  $S$  へのラップ解除として評価し、引き続き  $S$  から  $T$  への基になる変換を実行します。

null 許容の値をラップ解除しようとすると、値が `null` の場合は例外がスローされることに注意してください。

#### 6.2.4 明示的な参照変換

明示的な参照の変換は次のとおりです。

- `object` および `dynamic` から任意の他の *reference-type* への変換。
- 任意の *class-type*  $S$  から、任意の *class-type*  $T$  への変換。ただし  $S$  が  $T$  の基底クラスである場合。
- 任意の *class-type*  $S$  から、任意の *interface-type*  $T$  への変換。ただし、 $S$  がシールクラスではなく、 $S$  が  $T$  を実装していない場合。
- 任意の *interface-type*  $S$  から、任意の *class-type*  $T$  への変換。ただし、 $T$  がシールクラスではないか、または  $T$  が  $S$  を実装している場合。
- 任意の *interface-type*  $S$  から、任意の *interface-type*  $T$  への変換。ただし、 $S$  が  $T$  から派生していない場合。
- 要素型が  $S_E$  の *array-type*  $S$  から要素型が  $T_E$  の *array-type*  $T$  への変換。ただし、以下のすべてが満たされる場合。
  - $S$  と  $T$  は要素型だけが異なる。つまり、 $S$  と  $T$  の次元数が同じである。
  - ここで、 $S_E$  と  $T_E$  はいずれも *reference-type* です。
  - $S_E$  から  $T_E$  への明示的な参照変換が存在する。
- `System.Array` とそれが実装しているインターフェイスから任意の *array-type* への変換。
- 1 次元配列型  $S[]$  から `System.Collections.Generic.IList<T>` およびその基本インターフェイスへの変換。ただし  $S$  から  $T$  への明示的な参照変換がある場合。
- `System.Collections.Generic.IList<S>` およびその基本インターフェイスから 1 次元配列型  $T[]$  への変換。ただし  $S$  から  $T$  への明示的な恒等変換または参照変換がある場合。
- `System.Delegate` とそれが実装しているインターフェイスから任意の *delegate-type* への変換。

- 参照型から参照型  $T$  への変換。参照型  $T_0$  への明示的な参照変換があり、 $T_0$  に  $T$  への恒等変換がある場合。
- 参照型からインターフェイス型またはデリゲート型  $T$  への変換。インターフェイス型またはデリゲート型  $T_0$  への明示的な参照変換があり、 $T_0$  が  $T$  への変性変換であるか  $T$  が  $T_0$  への変性変換 (13.1.3.2 を参照) である場合。
- $D<S_1 \cdots S_n>$  から  $D<T_1 \cdots T_n>$  への変換。 $D<X_1 \cdots X_n>$  は汎用デリゲート型であり、 $D<S_1 \cdots S_n>$  は  $D<T_1 \cdots T_n>$  に対して互換性を持たないか、または同一ではありません。 $D$  の各型パラメーター  $x_i$  については、次のいずれかになります。
  - $x_i$  が不变である場合、 $S_i$  は  $T_i$  と同一です。
  - $x_i$  が共変である場合、 $S_i$  から  $T_i$  への暗黙的または明示的な恒等変換または参照変換が存在します。
  - $x_i$  が反変である場合、 $S_i$  と  $T_i$  は同一であるか、または共に参照型です。
- 参照型であることがわかっている型パラメーターを使用する明示的な変換。型パラメーターを使用する明示的な変換の詳細については、6.2.7 を参照してください。

明示的な参照変換は参照型間の変換なので、正しいことを確認するために実行時のチェックが必要です。

明示的な参照変換が実行時に成功するためには、変換元オペランドの値が `null` であるか、または変換元オペランドで参照されているオブジェクトの実際の型が、暗黙の参照変換 (6.1.6 を参照) またはボックス化変換 (6.1.7 を参照) によって変換先の型に変換できる型である必要があります。明示的な参照変換が失敗すると、`System.InvalidCastException` がスローされます。

暗黙または明示による参照変換では、変換されるオブジェクトの参照 ID が変わることはありません。つまり、参照変換により参照の型が変わることはありますが、参照されるオブジェクトの型または値が変化することはありません。

## 6.2.5 ボックス化解除変換

ボックス化解除変換では、参照型から *value-type* への明示的な変換ができます。`object` 型、`dynamic` 型、および `System.ValueType` 型から任意の *non-nullable-value-type* へのボックス化解除変換、および任意の *interface-type* からその *interface-type* を実装する任意の *non-nullable-value-type* へのボックス化解除変換が存在します。さらに、`System.Enum` 型は、任意の *enum-type* にボックス化解除できます。

参照型から *nullable-type* へのボックス化解除変換が存在するのは、参照型から *nullable-type* の基になる *non-nullable-value-type* へのボックス化解除変換が存在する場合です。

値型にインターフェイス型  $I_0$  からのボックス化解除変換があり、 $I_0$  に  $I$  への項等変換がある場合、値型  $s$  にはインターフェイス型  $I$  からのボックス化解除変換があります。

値型にインターフェイス型またはデリゲート型  $I_0$  からのボックス化解除変換があり、 $I_0$  が  $I$  への変性変換であるか  $I$  が  $I_0$  への変性変換 (13.1.3.2 を参照) である場合、値型  $s$  にはインターフェイス型  $I$  からのボックス化解除変換があります。

ボックス化解除の操作では、最初にオブジェクトインスタンスが特定の *value-type* のボックス化された値であることが確認された後、値がインスタンスからコピーされます。`null` 参照から *nullable-type* にボックス化解除すると、*nullable-type* の `null` 値が生成されます。構造体は `System.ValueType` 型からボックス化解除できます。この型はすべての構造体の基底クラスであるためです (11.3.2 を参照)。

ボックス化解除変換については、4.3.2 で詳細に説明します。

### 6.2.6 明示的な動的変換

`dynamic` 型の式から任意の型 `T` への明示的な動的変換が存在します。変換は動的にバインド (7.2.2 を参照) されます。つまり、式の実行時の型から `T` への明示的な変換が、実行時に検索されます。変換が見つからない場合は、実行時例外がスローされます。

変換の動的バインディングが望ましくない場合は、式を最初に `object` に変換してから、目的の型に変換できます。

次のようなクラスが定義されているとします。

```
class C
{
    int i;
    public C(int i) { this.i = i; }
    public static explicit operator C(string s)
    {
        return new C(int.Parse(s));
    }
}
```

次の例では、明示的な動的変換を示します。

```
object o = "1";
dynamic d = "2";

var c1 = (C)o; // Compiles, but explicit reference conversion fails
var c2 = (C)d; // Compiles and user defined conversion succeeds
```

`o` から `C` への明示的な参照変換としての最適な変換はコンパイル時に見つかります。実行時には、“1”が実際には `C` でないため、これは失敗します。一方、`d` から `C` への変換は明示的な動的変換であるため、実行時まで延期されます。実行時には、`d` の実行時の型 (`string`) から `C` へのユーザー定義の変換が見つかり、成功します。

### 6.2.7 型パラメーターを使用する明示的な変換

指定された型パラメーター `T` には、次のような明示的な変換が存在します。

- `T` の実質的な基底クラス `C` から `T` および `T` の任意の基底クラスから `C` への変換。実行時には、`T` が値型の場合、変換はボックス化解除変換として実行されます。それ以外の場合、変換は明示的な参照変換または恒等変換として実行されます。
- 任意のインターフェイス型から `T` への変換。実行時には、`T` が値型の場合、変換はボックス化解除変換として実行されます。それ以外の場合、変換は明示的な参照変換または恒等変換として実行されます。
- `T` から `I` への暗黙の型変換がない場合は、`T` から任意の *interface-type I* への変換。実行時には、`T` が値型の場合、変換はボックス化変換の後で明示的な参照変換が行われる変換として実行されます。それ以外の場合、変換は明示的な参照変換または恒等変換として実行されます。
- `T` が型パラメーター `U` に依存する場合は、`U` から `T` への変換 (10.1.5 を参照)。実行時、`U` が値型の場合は、`T` と `U` が必ず同じ型となり、変換は実行されません。`T` が値型の場合、変換はボックス化解除変換として実行されます。それ以外の場合、変換は明示的な参照変換または恒等変換として実行されます。

$T$  が参照型とわかっている場合は、上記の変換はすべて明示的な参照変換 (6.2.4 を参照) に分類されます。 $T$  が参照型でないことがわかっている場合は、上記の変換はボックス化解除変換に分類されます (6.2.5 を参照)。

注意する必要がある点は、上記の規則では、制約されない型パラメーターを非インターフェイス型に直接明示的に変換できないということです。これは、混乱を回避し、これらの変換の意味を明確にするためです。たとえば、次のような宣言があるとします。

```
class X<T>
{
    public static long F(T t) {
        return (long)t;           // Error
    }
}
```

$t$  から `int` への直接の明示的な変換が許可された場合は、`X<int>.F(7)` が `7L` を返すという結果が容易に予測できます。しかし、標準的な数値変換はバインディング時に型が数値型とわかっている場合にのみ考慮されるため、ここでは、そのような結果にはなりません。意味を明確にするためには、上記の例を次のように書き換える必要があります。

```
class X<T>
{
    public static long F(T t) {
        return (long)(object)t;   // Ok, but will only work when T is long
    }
}
```

このコードはコンパイルされますが、実行時に `X<int>.F(7)` を実行すると例外がスローされます。これは、ボックス化された `int` は直接 `long` に変換できないためです。

## 6.2.8 ユーザー定義の明示的な変換

ユーザー定義の明示的な変換は、標準明示変換 (省略可能)、ユーザー定義の暗黙または明示による変換演算子の実行、別の標準明示変換 (省略可能) という順序で行われます。ユーザー定義の明示的な変換の評価に関する厳密な規則については、6.4.5 で説明します。

## 6.3 標準変換

標準変換は、ユーザー定義変換の一部として実行される可能性のある定義済みの変換です。

### 6.3.1 標準暗黙変換

以下の暗黙変換は、標準暗黙変換に分類されます。

- 恒等変換 (6.1.1 を参照)
- 暗黙の数値変換 (0 を参照)
- 暗黙の `null` 許容変換 (6.1.4 を参照)
- `null` リテラル変換 (6.1.5 を参照)
- 暗黙の参照変換 (6.1.6 を参照)
- ボックス化変換 (6.1.7 を参照)
- 暗黙の定数式変換 (6.1.8 を参照)
- 型パラメーターを使用する暗黙の変換 (6.1.10 を参照)

ユーザー定義の暗黙変換は、標準暗黙変換には含まれていません。

### 6.3.2 標準明示変換

標準明示変換は、すべての標準暗黙変換に、逆方向の標準暗黙変換が存在する明示変換のサブセットを加えたものです。つまり、型 A から型 B に対する標準暗黙変換が存在する場合は、型 A から型 B に対する標準明示変換と型 B から型 A に対する標準明示変換が存在します。

## 6.4 ユーザー定義変換

C# では、ユーザー定義変換を使って、定義済みの暗黙変換と明示変換を強化できます。ユーザー定義変換を導入するには、クラス型と構造体型で変換演算子(10.10.3 を参照)を宣言します。

### 6.4.1 使用できるユーザー定義変換

C# では、特定のユーザー定義変換だけを宣言できます。特に、既存の暗黙変換または明示変換を再定義することはできません。

変換元の型  $s$  と変換先の型  $T$  が指定されている場合、 $s$  または  $T$  が null 許容型であれば、 $s_0$  と  $T_0$  は基になる型を参照できますが、それ以外の場合、 $s_0$  と  $T_0$  はそれぞれ  $s$  と  $T$  に等しくなります。クラスまたは構造体において変換元の型  $s$  から変換先の型  $T$  への変換を宣言できるのは、以下の条件がすべて満たされている場合だけです。

- $s_0$  と  $T_0$  の型が異なる。
- $s_0$  または  $T_0$  が、演算子の宣言が行われるクラスまたは構造体の型である。
- $s_0$  と  $T_0$  のどちらも *interface-type* ではない。
- ユーザー定義の変換を除き、 $s$  から  $T$  への変換も  $T$  から  $s$  への変換も存在しない。

ユーザー定義変換に適用される制限の詳細については、10.10.3 で説明します。

### 6.4.2 リフト変換演算子

null 非許容の値型  $s$  を null 非許容の値型  $T$  に変換するユーザー定義変換演算子がある場合、 $s?$  を  $T?$  に変換する "リフト変換演算子" が存在します。このリフト変換演算子は、 $s?$  から  $s$  へのラップ解除を実行し、次に  $s$  から  $T$  へのユーザー定義変換を実行し、その次に  $T$  から  $T?$  へのラップを実行します。ただし、null 値の  $s?$  は直接 null 値の  $T?$  に変換されます。

リフト変換演算子は、基になるユーザー定義変換演算子と同じように暗黙的または明示的に分類されます。"ユーザー定義変換" という用語は、ユーザー定義変換演算子とリフト変換演算子の両方に適用されます。

### 6.4.3 ユーザー定義変換の評価

ユーザー定義変換は、変換元の型と呼ばれる型の値を、変換先の型と呼ばれる別の型に変換します。ユーザー定義変換の評価は、特定の変換元と変換先の型に対して、最も限定的なユーザー定義変換演算子を検出することを中心に行われます。この決定作業は、いくつかの手順に分かれます。

- ユーザー定義変換演算子を考慮する基になるクラスおよび構造体の組を発見します。この組は、変換元の型とその基底クラス、および変換先の型とその基底クラスで構成されています。ここでは、クラスと構造体だけがユーザー定義演算子を宣言でき、クラス以外の型には基底クラスがないということが、暗黙で仮定されています。この手順では、変換元または変換先の型が nullable-type の場合は、それぞれの基になる型が使用されます。

- 発見された型の組から、適用できるユーザー定義変換演算子およびリフト変換演算子を決定します。変換演算子が適切であるためには、変換元の型から演算子のオペランドの型への標準変換(6.3を参照)を実行でき、演算子の結果の型から変換先の型への標準変換を実行が必要です。
- 適用できるユーザー定義演算子の集合から、明らかに最も限定的な演算子を決定します。一般に、最も限定的な演算子とは、オペランドの型が変換元の型に"最も近く"、結果の型が変換先の型に"最も近い"演算子です。ユーザー定義変換演算子はリフト変換演算子より優先されます。最も限定的なユーザー定義変換演算子を確定する厳密な規則については、後のセクションで定義します。

最も限定的なユーザー定義変換演算子が識別されたら、ユーザー定義変換の実際の実行は、次の3つの手順で行います。

- 最初に、必要な場合は、変換元の型からユーザー定義変換演算子またはリフト変換演算子のオペランドの型への標準変換を実行します。
- 次に、ユーザー定義変換演算子またはリフト変換演算子を呼び出して変換を実行します。
- 最後に、必要な場合は、ユーザー定義変換演算子またはリフト変換演算子の結果の型から変換先の型への標準変換を実行します。

ユーザー定義変換の評価に複数のユーザー定義変換演算子またはリフト変換演算子が関係することはありません。つまり、型  $S$  から型  $T$  への変換では、最初にユーザー定義変換を実行して  $S$  から  $X$  に変換し、さらにユーザー定義変換を実行して  $X$  から  $T$  に変換する、という処理は行われません。

ユーザー定義の暗黙変換または明示変換の評価の厳密な定義については、後のセクションで説明します。定義では以下の用語を使用します。

- 型  $A$  から型  $B$  への標準暗黙変換(6.3.1を参照)があり、 $A$  も  $B$  も *interface-types* ではない場合、 $A$  は  $B$  に"包含されている"といい、 $B$  は  $A$  を"包含している"といいます。
- 式  $E$  から型  $B$  への標準暗黙変換(6.3.1を参照)があり、 $B$  も  $E$  の型(型を持つ場合)も *interface-types* ではない場合、 $E$  は  $B$  に"包含されている"といい、 $B$  は  $E$  を"包含している"といいます。
- 型の集合の中で"最も外側"の型とは、集合内の他のすべての型を包含している1つの型です。他のすべての型を包含する唯一の型がない場合、その集合に最も外側の型はありません。つまり、最も外側の型とは集合の中で"最も大きい"型であり、他の各型はその1つの型に対して暗黙に変換できます。
- 型の集合の中で"最も内側"の型とは、集合内の他のすべての型に包含されている1つの型です。他のすべての型に包含される唯一の型がない場合、その集合に最も内側の型はありません。つまり、最も内側の型とは集合の中で"最も小さい"型であり、その1つの型は他の各型に対して暗黙に変換できます。

#### 6.4.4 ユーザー定義の暗黙の変換

式  $E$  から型  $T$  へのユーザー定義の暗黙の変換は、以下のように処理されます。

- 型  $S$ 、 $S_0$ 、および  $T_0$  を決定します。
  - $E$  が型を持つ場合は、 $S$  をその型にします。
  - $S$  または  $T$  が null 許容型の場合、 $S_u$  と  $T_u$  をそれぞれの基になる型にします。それ以外の場合、 $S_u$  と  $T_u$  をそれぞれ  $S$  と  $T$  にします。

- $S_u$  または  $T_u$  が型パラメーターの場合、 $S_0$  と  $T_0$  をその実質的な基底クラスにします。それ以外の場合、 $S_0$  と  $T_0$  をそれぞれ、 $S_u$  と  $T_u$  にします。
- ユーザー定義変換演算子を考慮する基になる型の集合  $D$  を決定します。このセットは、 $S_0$  ( $S_0$  が存在し、クラスまたは構造体の場合)、 $S_0$  の基底クラス ( $S_0$  が存在し、クラスの場合)、および  $T_0$  ( $T_0$  がクラスまたは構造体の場合) で構成されます。型は、セットに含まれている別の型への恒等変換が存在しない場合にのみ、セット  $D$  に追加されます。
- 適用できるユーザー定義変換演算子およびリフト変換演算子の集合  $U$  を決定します。この集合は、 $D$  のクラスまたは構造体で宣言されているユーザー定義変換演算子およびリフト変換演算子で構成されています。これらの演算子は、 $E$  を包含する型から  $T$  によって包含される型への変換を行います。 $U$  が空の場合、変換は未定義で、コンパイル エラーになります。
- $S_x$  の演算子の中で最も限定期的な変換元の型  $U$  を決定します。
  - $S$  が存在し、 $U$  の演算子のいずれかが  $S$  からの変換を行う場合、 $S_x$  は  $S$  です。
  - それ以外の場合、 $S_x$  は、 $U$  に含まれる演算子の変換元の型の中で最も内側の型です。厳密に最も内側の型が見つからない場合、変換はあいまいであり、コンパイル エラーが発生します。
- $T_x$  の演算子の中で最も限定期的な変換先の型  $U$  を決定します。
  - $U$  の演算子のいずれかが  $T$  に変換される場合、 $T_x$  は  $T$  です。
  - それ以外の場合、 $T_x$  は、 $U$  に含まれる演算子の変換先の型の中で最も外側の型です。厳密に最も外側の型が見つからない場合、変換はあいまいであり、コンパイル エラーが発生します。
- 最も限定期的な変換演算子を、次のように発見します。
  - $S_x$  から  $T_x$  に変換するユーザー定義変換演算子が  $U$  の中に 1 つだけ存在する場合は、それが最も限定期的な変換演算子です。
  - それ以外の場合で、 $S_x$  から  $T_x$  に変換するリフト変換演算子が  $U$  の中にただ 1 つだけある場合は、それが最も限定期的な変換演算子です。
  - それ以外の場合、変換はあいまいであり、コンパイル エラーが発生します。
- 最後に、次のように変換を適用します。
  - $E$  がまだ型  $S_x$  を持たない場合は、 $E$  から  $S_x$  への標準暗黙変換が実行されます。
  - 最も限定期的な変換演算子が呼び出されて、 $S_x$  から  $T_x$  への変換が行われます。
  - $T_x$  が  $T$  ではない場合は、 $T_x$  から  $T$  への標準暗黙変換が実行されます。

型  $S$  の変数から型  $T$  へのユーザー定義の暗黙の変換が存在する場合、型  $S$  から型  $T$  へのユーザー定義の暗黙の変換が存在します。

#### 6.4.5 ユーザー定義の明示的な変換

式  $E$  から型  $T$  へのユーザー定義の明示的な変換は、以下のように処理されます。

- 型  $S$ 、 $S_0$ 、および  $T_0$  を決定します。
  - $E$  が型を持つ場合は、 $S$  をその型にします。
  - $S$  または  $T$  が null 許容型の場合、 $S_u$  と  $T_u$  をそれぞれの基になる型にします。それ以外の場合、 $S_u$  と  $T_u$  をそれぞれ  $S$  と  $T$  にします。

- $S_u$  または  $T_u$  が型パラメーターの場合、 $S_0$  と  $T_0$  をその実質的な基底クラスにします。それ以外の場合、 $S_0$  と  $T_0$  をそれぞれ、 $S_u$  と  $T_u$  にします。
- ユーザ一定義変換演算子を考慮する基になる型の集合  $D$  を決定します。このセットは、 $S_0$  ( $S_0$  が存在し、クラスまたは構造体の場合)、 $S_0$  の基底クラス ( $S_0$  が存在し、クラスの場合)、 $T_0$  ( $T_0$  がクラスまたは構造体の場合)、および  $T_0$  の基底クラス ( $T_0$  がクラスの場合) で構成されます。型は、セットに含まれている別の型への恒等変換が存在しない場合にのみ、セット  $D$  に追加されます。
- 適用できるユーザ一定義変換演算子およびリフト変換演算子の集合  $U$  を決定します。この集合は、 $D$  のクラスまたは構造体で宣言されているユーザ一定義またはリフトの暗黙または明示の変換演算子で構成されています。これらの演算子は、 $E$  を包含する型または  $S$  によって包含される型 (存在する場合) から、 $T$  を包含する型または  $T$  によって包含される型への変換を行います。 $U$  が空の場合、変換は未定義で、コンパイルエラーになります。
- $S_x$  の演算子の中で最も限定的な変換元の型  $U$  を決定します。
  - $S$  が存在し、 $U$  の演算子のいずれかが  $S$  からの変換を行う場合、 $S_x$  は  $S$  です。
  - それ以外で、 $U$  の演算子のいずれかが  $E$  を包含する型からの変換を行う場合は、 $S_x$  がこれらの演算子の変換元の型の中で最も内側の型です。最も内側の型が見つからない場合は、変換はあいまいであり、コンパイルエラーが発生します。
  - それ以外の場合、 $S_x$  は、 $U$  に含まれる演算子の変換元の型の中で最も外側の型です。厳密に最も外側の型が見つからない場合、変換はあいまいであり、コンパイルエラーが発生します。
- $T_x$  の演算子の中で最も限定的な変換先の型  $U$  を決定します。
  - $U$  の演算子のいずれかが  $T$  に変換される場合、 $T_x$  は  $T$  です。
  - それ以外で、 $U$  の演算子のいずれかが  $T$  によって包含される型からの変換を行う場合は、 $T_x$  がこれらの演算子の変換元の型の中で最も外側の型です。厳密に最も外側の型が見つからない場合、変換はあいまいであり、コンパイルエラーが発生します。
  - それ以外の場合、 $T_x$  は、 $U$  に含まれる演算子の変換先の型の中で最も内側の型です。最も内側の型が見つからない場合は、変換はあいまいであり、コンパイルエラーが発生します。
- 最も限定的な変換演算子を、次のように発見します。
  - $S_x$  から  $T_x$  に変換するユーザ一定義変換演算子が  $U$  の中に 1 つだけ存在する場合は、それが最も限定的な変換演算子です。
  - それ以外の場合で、 $S_x$  から  $T_x$  に変換するリフト変換演算子が  $U$  の中にただ 1 つだけある場合は、それが最も限定的な変換演算子です。
  - それ以外の場合、変換はあいまいであり、コンパイルエラーが発生します。
- 最後に、次のように変換を適用します。
  - $E$  が型  $S_x$  を持たない場合は、 $E$  から  $S_x$  への標準明示変換が実行されます。
  - 最も限定的なユーザ一定義変換演算子が呼び出されて、 $S_x$  から  $T_x$  への変換が行われます。
  - $T_x$  が  $T$  ではない場合は、 $T_x$  から  $T$  への標準明示変換が実行されます。

型  $S$  の変数から型  $T$  へのユーザ一定義の明示的な変換が存在する場合、型  $S$  から型  $T$  へのユーザ一定義の明示的な変換が存在します。

## 6.5 匿名関数の変換

*anonymous-method-expression* または *lambda-expression* は匿名関数 (7.15 を参照) に分類されます。この式には型がありませんが、互換性のあるデリゲート型または式ツリー型に暗黙的に変換できます。具体的には、匿名関数 *F* は次のような条件でデリゲート型 *D* との互換性を持ちます。

- *F* に *anonymous-function-signature* が含まれている場合、*D* と *F* は同数のパラメーターを持ちます。
- *F* に *anonymous-function-signature* が含まれていない場合は、*D* のパラメーターがいずれも *out* パラメーター修飾子を持たない限り、*D* はパラメーターを持たないことも、任意の型のパラメーターを任意の数だけ持つこともできます。
- *F* に明示的な型のパラメーター リストがある場合、*D* の各パラメーターは、*F* の対応するパラメーターと同じ型と修飾子を持ちます。
- *F* に暗黙の型のパラメーター リストがある場合、*D* は *ref* パラメーターも *out* パラメーターも持ちません。
- *F* の本体が式で、*D* の戻り値の型が *void* か、または *F* が非同期であり、*D* の戻り値の型が *Task* である場合、*F* の各パラメーターに *D* の対応するパラメーターの型を指定すると、*F* の本体は *statement-expression* (8.6 を参照) として使用できる有効な式 (7 を参照) になります。
- *F* の本体がステートメントブロックで、*D* の戻り値の型が *void* か、または *F* が非同期で、*D* の戻り値の型が *Task* である場合、*F* の各パラメーターに *D* の対応するパラメーターの型を指定すると、*F* の本体は、*return* ステートメントによる式の指定のない有効なステートメントブロック (8.2 を参照) になります。
- *F* の本体が式であり、*F* が非同期でなく *D* の戻り値の型 *T* が *void* 以外の場合、または *F* が非同期であり *D* の戻り値の型が *Task<T>* の場合、*F* の各パラメーターに *D* の対応するパラメーターの型を指定すると、*F* の本体は、戻り値型 *T* に暗黙的に変換可能な有効な式 (7 を参照) になります。
- *F* の本体がステートメントブロックであり、*F* が非同期でなく *D* の戻り値の型 *T* が *void* 以外の場合、または *F* が非同期であり *D* の戻り値の型が *Task<T>* の場合、*F* の各パラメーターに *D* の対応するパラメーターの型を指定すると、*F* の本体は、各 *return* ステートメントが戻り値型 *T* に暗黙的に変換可能な式を指定する到達不可能な終了点を持つ有効なステートメントブロック (8.2 を参照) になります。

説明を簡潔にするために、このセクションではタスクの型 *Task* および *Task<T>* の省略形を使用します (エラー! 参照元が見つかりません。 を参照)。

ラムダ式 *F* がデリゲート型 *D* と互換性がある場合、*F* は式ツリー型 *Expression<D>* と互換性があります。これは、匿名メソッドには適用されず、ラムダ式のみに適用されることに注意してください。

一部のラムダ式は式ツリー型に変換できません。変換は存在しますが、コンパイル時に失敗します。このようになるのは、ラムダ式が次の場合です。

- *block* 本体がある
- 単純または複合代入演算子を含みます
- 動的にバインドされる式を含む
- 非同期である

以下の例では、型 *A* の引数を使用し、型 *R* の値を返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);
```

次に代入の例を示します。

```
Func<int,int> f1 = x => x + 1;           // Ok
Func<int,double> f2 = x => x + 1;          // Ok
Func<double,int> f3 = x => x + 1;          // Error
Func<int, Task<int>> f4 = async x => x + 1; // Ok
```

この例では、各匿名関数のパラメーターと戻り値型は、その匿名関数が代入される変数の型から決定されます。

最初の代入により、匿名関数はデリゲート型 *Func<int,int>* に変換されます。これは、*x* に *int* 型が指定されると、*x+1* は *int* 型に暗黙的に変換できる有効な式になるためです。

同様に、2番目の代入により、匿名関数はデリゲート型 *Func<int,double>* に変換されます。これは、*x+1* (*int* 型) の結果が、*double* 型に暗黙的に変換可能であるためです。

しかし、3番目の代入はコンパイル時のエラーになります。これは、*x* に *double* 型が指定されると、*x+1* (*double* 型) の結果は、*int* 型に暗黙的に変換できないためです。

4番目の代入により、匿名非同期関数はデリゲート型 *Func<int, Task<int>>* に変換されます。これは、*x+1* (*int* 型) の結果が、タスクの型 *Task<int>* の結果の型 *int* に暗黙的に変換可能であるためです。

匿名関数はオーバーロード解決に影響することがあり、型推論にも関与します。7.5 を参照してください。詳細を確認できます。

### 6.5.1 デリゲート型への匿名関数の変換の評価

匿名関数をデリゲート型に変換すると、その匿名関数と、評価の時点でアクティブなキャプチャされた外部変数の集合（多くの場合は空の集合）を参照するデリゲートインスタンスが生成されます。そのデリゲートを呼び出すと、匿名関数の本体が実行されます。本体のコードは、デリゲートが参照するキャプチャされた外部変数の集合を使用して実行されます。

匿名関数から生成されるデリゲートの呼び出しリストには、单一のエントリがあります。デリゲートの厳密なターゲット オブジェクトとターゲットメソッドは指定されません。特に、デリゲートのターゲット オブジェクトが *null*、外側の関数メンバーの *this* 値、またはその他のオブジェクトのいずれであるかは指定されません。

キャプチャされた外部変数インスタンスの集合が同じで（多くの場合は空の集合）、意味が同じ匿名関数を同じデリゲート型に変換する場合、同じデリゲートインスタンスを返すことができますが、必須ではありません。意味が同じであるということは、同じ引数を指定してその匿名関数を実行すると、すべてのケースで同じ結果が得られるということを示します。この規則により、次のようなコードを最適化できます。

```
delegate double Function(double x);
```

```

class Test
{
    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void F(double[] a, double[] b) {
        a = Apply(a, (double x) => Math.Sin(x));
        b = Apply(b, (double y) => Math.Sin(y));
        ...
    }
}

```

この 2 つの匿名関数のデリゲートは、キャプチャされた外部変数の集合(空の集合)が同じであり、したがって、匿名関数は同じ意味を持つため、コンパイラは 2 つのデリゲートに同じターゲットメソッドを参照させることができます。実際に、コンパイラは、両方の匿名関数の式からまったく同じデリゲートインスタンスを返すことができます。

### 6.5.2 式ツリー型への匿名関数の変換の評価

匿名関数を式ツリー型に変換すると、式ツリー(4.6 を参照)が生成されます。より厳密には、匿名関数の変換の評価により、その匿名関数自体の構造を表すオブジェクト構造が構築されます。式ツリーの厳密な構造、およびそれを作成するための正確なプロセスは、実装で定義されています。

### 6.5.3 実装例

ここでは、匿名関数の変換の実装について、他の C# 構造に基づいて説明します。ここで説明する実装は、Microsoft C# コンパイラの原理と同じ原理に基づきますが、どのような観点からも必須の実装ではなく、また唯一可能な実装でもありません。式ツリーへの変換の正確なセマンティクスはこの仕様書の範囲外であるため、概要だけを示します。

この後では、さまざまな特性を持つ匿名関数を含むコードの例をいくつか紹介します。それぞれの例について、他の C# 構造のみを使用した場合のコードの書き換え例を示します。これらの例では、識別子 D は次のデリゲート型を表すものとします。

```
public delegate void D();
```

最も単純な形式の匿名関数は、外部変数をキャプチャしない匿名関数です。

```

class Test
{
    static void F() {
        D d = () => { Console.WriteLine("test"); };
    }
}

```

これは、次のように、コンパイラによって生成された静的メソッドを参照するデリゲートのインスタンス化に書き換えることができます。この静的メソッドの中に、匿名関数のコードが置かれます。

```

class Test
{
    static void F() {
        D d = new D(__Method1);
    }

    static void __Method1() {
        Console.WriteLine("test");
    }
}

```

次の例では、匿名関数は `this` のインスタンス メンバーを参照します。

```
class Test
{
    int x;
    void F() {
        D d = () => { Console.WriteLine(x); };
    }
}
```

これは、匿名関数のコードを含む、コンパイラによって生成されたインスタンス メソッドに書き換えることができます。

```
class Test
{
    int x;
    void F() {
        D d = new D(__Method1);
    }
    void __Method1() {
        Console.WriteLine(x);
    }
}
```

この例では、匿名関数はローカル変数をキャプチャします。

```
class Test
{
    void F() {
        int y = 123;
        D d = () => { Console.WriteLine(y); };
    }
}
```

ローカル変数の有効期間を、少なくとも、匿名関数のデリゲートの有効期間まで延長する必要があります。そのためには、コンパイラによって生成されたクラスのフィールドにローカル変数を "引き上げ" ます。ローカル変数のインスタンス化(7.15.5.2 を参照)は、コンパイラによって生成されたクラスのインスタンスの作成に対応し、ローカル変数のアクセスは、コンパイラによって生成されたクラスのインスタンスのフィールドへのアクセスに対応します。また、次のように、匿名関数はコンパイラによって生成されたクラスのインスタンス メソッドになります。

```
class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.y = 123;
        D d = new D(__locals1.__Method1);
    }
    class __Locals1
    {
        public int y;
        public void __Method1() {
            Console.WriteLine(y);
        }
    }
}
```

最後に、次の匿名関数は、`this` と、有効期間の異なる 2 つのローカル変数をキャプチャします。

```
class Test
{
    int x;
    void F() {
        int y = 123;
        for (int i = 0; i < 10; i++) {
            int z = i * 2;
            D d = () => { Console.WriteLine(x + y + z); };
        }
    }
}
```

ここでは、ローカル変数をキャプチャする各ステートメント ブロックに対して、コンパイラによって生成されたクラスが作成され、異なるブロックのローカル変数が独立した有効期間を持つことができるようになっています。`__Locals2` のインスタンスは、内側のステートメント ブロックに対してコンパイラによって生成されたクラスであり、ローカル変数 `z` と、`__Locals1` のインスタンスを参照するフィールドを含みます。`__Locals1` のインスタンスは、外側のステートメント ブロックに対してコンパイラによって生成されたクラスであり、ローカル変数 `y` と、外側の関数メンバーの `this` を参照するフィールドを含みます。このようなデータ構造では、`__Local2` のインスタンスを通じてすべてのキャプチャされた外部変数に到達でき、匿名関数のコードはそのクラスのインスタンスメソッドとして実装できます。

```
class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.__this = this;
        __locals1.y = 123;
        for (int i = 0; i < 10; i++) {
            __Locals2 __locals2 = new __Locals2();
            __locals2.__locals1 = __locals1;
            __locals2.z = i * 2;
            D d = new D(__locals2.__Method1);
        }
    }
    class __Locals1
    {
        public Test __this;
        public int y;
    }
    class __Locals2
    {
        public __Locals1 __locals1;
        public int z;
        public void __Method1() {
            Console.WriteLine(__locals1.__this.x + __locals1.y + z);
        }
    }
}
```

匿名関数を式ツリーに変換する場合、ローカル変数をキャプチャするためにここで適用された同じ方法を使用できます。つまり、コンパイラで生成されたオブジェクトへの参照は、式ツリーに保存でき、ローカル変数へのアクセスは、これらのオブジェクトでフィールドアクセスとして表すことができます。この方法の利点は、"リフトされた" ローカル変数をデリゲートおよび式ツリー間で共有できることです。

## 6.6 メソッドのグループの変換

メソッドグループ(7.1を参照)から互換性のあるデリゲート型への暗黙の変換(6.1を参照)が存在します。デリゲート型 **D** と、メソッドグループに分類される式 **E** があり、**D** のパラメーター型および修飾子を使用して構築された引数リストに対して標準形式(7.5.3.1を参照)で適用できるメソッドを **E** が少なくとも 1つ含む場合、以下に示したとおり、**E** から **D** への暗黙の型変換が存在します。

次に、コンパイル時のメソッド グループ **E** からデリゲート型 **D** への変換の適用について説明します。**E** から **D** への暗黙の型変換が存在しても、コンパイル時に変換がエラーなく正常に適用されることを保証されません。

- 形式 **E(A)** のメソッド呼び出し(7.6.5.1を参照)に対応する 1つのメソッド **M** が選択され、次のように変更されます。
  - 引数リスト **A** は、それぞれ変数として分類される式のリストで、各式は、**D** の *formal-parameter-list* で対応するパラメーターの型と修飾子(**ref** または **out**)を持ちます。ただし、対応する式が型 **dynamic** ではなく型 **object** を持つ型 **dynamic** のパラメーターを除きます。
  - 考慮される候補メソッドは、標準形式で使用でき、省略可能なパラメーターを省略しないメソッドのみです(7.5.3.1を参照)。したがって、拡張形式でのみ使用できる場合や、1つまたは複数の省略可能なパラメーターが **D** に対応するパラメーターを持たない場合、候補メソッドは無視されます。
- 7.6.5.1 のアルゴリズムによって、**D** と同数のパラメーターを持つ 1つの最適なメソッド **M** が生成される場合、変換が存在すると見なされます。
- 変換が存在する場合でも、選択されたメソッド **M** とデリゲート型 **D** との間に互換性がない場合(15.2を参照)はコンパイルエラーになります。
- 選択されたメソッド **M** がインスタンス メソッドの場合は、**E** に関連付けられたインスタンス式によってデリゲートの対象オブジェクトが決まります。
- 選択したメソッド **M** が、インスタンス式でのメンバー アクセスによって表される拡張メソッドである場合、そのインスタンス式によって、デリゲートのターゲット オブジェクトが決定されます。
- 結果は、**D** 型の値です。つまり、選択されたメソッドと対象のオブジェクトを示す、新しく作成されたデリゲートです。

7.6.5.1 のアルゴリズムがインスタンス メソッドを見つけることができずに **E(A)** の呼び出しを拡張メソッド呼び出し(7.6.5.2を参照)として処理することに成功すると、この処理は拡張メソッドに対するデリゲートを作成する可能性があります。このようにして作成されたデリゲートは最初の引数だけでなく、拡張メソッドもキャプチャします。

次の例は、メソッド グループの変換を示します。

```
delegate string D1(object o);
delegate object D2(string s);
delegate object D3();
delegate string D4(object o, params object[] a);
delegate string D5(int i);
class Test
{
    static string F(object o) {...}
```

```
static void G() {
    D1 d1 = F;           // Ok
    D2 d2 = F;           // Ok
    D3 d3 = F;           // Error - not applicable
    D4 d4 = F;           // Error - not applicable in normal form
    D5 d5 = F;           // Error - applicable but not compatible
}
```

d1 に代入すると、メソッド グループ F は型 D1 の値に暗黙的に変換されます。

d2 への代入は、派生階層が下位の(反変性の) パラメーター型と上位の(共変性の) 戻り値型を持つメソッドへのデリゲートを作成できることを示しています。

d3 への代入は、メソッドを適用できない場合は変換が存在しないことを示しています。

d4 への代入は、メソッドがその標準形式で適用できる必要があることを示しています。

d5 への代入では、デリゲートとメソッドの間でパラメーター型と戻り値の型が異なっていることが許可されるのは、参照型についてのみであることがわかります。

他のすべての暗黙および明示的な変換と同様に、キャスト演算子を使用してメソッド グループの変換を明示的に実行できます。次に例を示します。

```
object obj = new EventHandler(myDialog.okClick);
```

これは、次のように書き換えることができます。

```
object obj = (EventHandler)myDialog.okClick;
```

メソッド グループはオーバーロード解決に影響することがあり、型推論にも関与します。7.5 を参照してください。詳細を確認できます。

メソッド グループ変換の実行時評価は、次のように処理されます。

- コンパイル時に選択されたメソッドがインスタンス メソッドの場合、またはインスタンス メソッドとしてアクセスされる拡張メソッドの場合、デリゲートのターゲット オブジェクトは、E と関連付けられたインスタンス式から決定されます。
  - インスタンス式を評価します。この評価で例外が発生すると、それ以上の手順は実行されません。
  - インスタンス式が *reference-type* の場合は、インスタンス式によって計算された値が対象オブジェクトになります。選択されたメソッドがインスタンス メソッドで、対象オブジェクトが **null** の場合は、**System.NullReferenceException** がスローされ、以降の手順は実行されません。
  - インスタンス式が *value-type* の場合は、ボックス化演算(4.3.1 を参照)を行って値をオブジェクトに変換し、このオブジェクトが対象オブジェクトになります。
- それ以外の場合、選択したメソッドは静的メソッド呼び出しの一部であり、デリゲートのターゲット オブジェクトは **null** になります。
- デリゲート型 D の新しいインスタンスを割り当てます。新しいインスタンスを割り当てるための十分なメモリがない場合は、**System.OutOfMemoryException** がスローされ、以降の手順は実行されません。

- 新しいデリゲートインスタンスは、コンパイル時に決定されたメソッドに対する参照と、前記の処理で計算された対象オブジェクトに対する参照を使って、初期化されます。

## 7. 式

式は、一連の演算子とオペランドで構成されます。ここでは、構文、オペランドおよび演算子の評価順序、および式の意味を定義します。

### 7.1 式の分類

式は次のいずれかに分類されます。

- 値。すべての値には、型が関連付けられています。
- 変数。すべての変数は、関連付けられた型を持ちます。これを、変数の宣言された型と呼びます。
- 名前空間。この種類の式は、*member-access* (7.6.4 を参照) の左側にだけ記述できます。名前空間として分類される式がそれ以外のコンテキストで使用された場合は、コンパイルエラーが発生します。
- 型。この種類の式は、*member-access* (7.6.4 を参照) の左側、または **as** 演算子 (7.10.11 を参照)、**is** 演算子 (7.10.10 を参照)、**typeof** 演算子 (7.6.11 を参照) のオペランドとしてのみ記述できます。型として分類される式がそれ以外のコンテキストで使用された場合は、コンパイルエラーが発生します。
- メソッドグループ。メンバー検索の結果として作成されるオーバーロードされたメソッドの集合です (7.4 を参照)。メソッドグループには、インスタンス式と型引数リストが関連付けられています。インスタンスメソッドが呼び出されると、インスタンス式の評価結果が **this** で表されるインスタンスになります (7.6.7 を参照)。メソッドグループは *invocation-expression* (7.6.5 を参照)、*delegate-creation-expression* (7.6.10.5 を参照)、および演算子の左側として使用でき、互換性のあるデリゲート型 (6.6 を参照) に暗黙的に変換できます。それ以外のコンテキストで、メソッドグループに分類される式を使用すると、コンパイルエラーになります。
- **null** リテラル。この種類の式は、参照型または **null** 許容型に暗黙的に変換できます。
- 匿名関数。この種類の式は、互換性のあるデリゲート型または式ツリー型に暗黙的に変換できます。
- プロパティアクセス。すべてのプロパティアクセスは、関連付けられた型を持ちます。これを、プロパティの型と呼びます。さらに、プロパティアクセスには、インスタンス式が関連付けられる場合があります。インスタンスプロパティアクセスのアクセサー (**get** ブロックまたは **set** ブロック) が呼び出されると、インスタンス式の評価結果が、**this** (7.6.7 を参照) によって表されるインスタンスになります。
- イベントアクセス。すべてのイベントアクセスは、関連付けられた型を持ちます。これを、イベントの型と呼びます。さらに、イベントアクセスには、インスタンス式が関連付けられる場合があります。イベントアクセスは、**+=** 演算子および **-=** 演算子の左辺のオペランドで使用できます (7.17.3 参照)。イベントアクセスとして分類される式がそれ以外のコンテキストで使用された場合は、コンパイルエラーが発生します。

- インデクサー アクセス。すべてのインデクサー アクセスには、型が関連付けられています。つまり、インデクサーの要素の型です。さらに、インデクサーには、インスタンス式と引数リストが関連付けられています。インデクサー アクセスのアクセサー (`get` ブロックまたは `set` ブロック) が呼び出されると、インスタンス式の評価結果が `this` (7.6.7 を参照) によって表されるインスタンスになり、引数リストの評価結果が呼び出しのパラメーターリストになります。
- なし。これは、戻り値の型が `void` のメソッドを呼び出す式において発生します。なしに分類される式は、*statement-expression* (8.6 を参照) のコンテキストにおいてのみ有効です。

式の最終的な結果が名前空間、型、メソッド グループ、またはイベント アクセスになることはありません。先に示したように、これらの種類の式は中間的な構造で、特定のコンテキストだけで許されます。

プロパティ アクセスまたはインデクサー アクセスは、*get-accessor* または *set-accessor* の呼び出しを実行することで、常に値として分類し直されます。実際のアクセサーは、プロパティ アクセスまたはインデクサー アクセスのコンテキストによって決定します。アクセスが代入の対象である場合は、*set-accessor* を呼び出して新しい値を代入します (7.17.1 を参照)。それ以外の場合は、*get-accessor* を呼び出して現在の値を取得します (7.1.1 を参照)。

### 7.1.1 式の値

式が関係する構造のほとんどでは、最終的に式が "値" を表すことが求められます。このような場合に、実際の式が名前空間、型、メソッド グループ、または "なし" を表すと、コンパイル エラーになります。ただし、式がプロパティ アクセス、インデクサー アクセス、または変数を表す場合は、プロパティ、インデクサー、または変数の値に暗黙で置換されます。

- 変数の値は、単に、変数が示す格納場所に現在格納されている値です。確実に代入されている (5.3 を参照) と見なすことのできる変数からのみ、値を取得できます。そうでない場合は、コンパイル エラーが発生します。
- プロパティ アクセス式の値は、プロパティの *get-accessor* を呼び出して取得します。プロパティに *get-accessor* がない場合は、コンパイル エラーが発生します。存在する場合は、関数メンバーの呼び出し (7.5.4 を参照) が実行され、呼び出しの結果がプロパティ アクセス式の値になります。
- インデクサー アクセス式の値は、インデクサーの *get-accessor* を呼び出して取得します。インデクサーに *get-accessor* がない場合は、コンパイル エラーが発生します。存在する場合は、インデクサー アクセス式に関連付けられている引数リストを使って関数メンバーの呼び出し (7.5.4 を参照) が実行されて、呼び出しの結果が インデクサー アクセス式の値になります。

### 7.2 静的バインディングと動的バインディング

構成する式 (引数、オペランド、レシーバー) の型または値に基づいて操作の意味を決定するプロセスは、"バインディング" と呼ばれることがあります。たとえば、メソッド呼び出しの意味は、レシーバーと引数の型に基づいて決定されます。演算子の意味は、オペランドの型に基づいて決定されます。

C# では、操作の意味は、普通、コンパイル時に、構成する式のコンパイル時の型に基づいて決定されます。同様に、式にエラーが含まれる場合は、コンパイラによってエラーが検出されて報告されます。このような方法を "静的バインディング" といいます。

ただし、式が "動的式" (つまり、型が `dynamic`) の場合は、関係するすべてのバインディングは、コンパイル時の型ではなく、実行時の型 (つまり、オブジェクトが実行時に表す "実際の" 型) に基づき

ます。したがって、このような操作のバインディングは、プログラム実行中の操作が実行されるときまで遅延されます。これを "動的バインディング" と言います。

操作が動的バインディングの場合、コンパイラによる検査はほとんど、またはまったく実行されません。実行時のバインディングが失敗した場合、エラーは実行時に例外として報告されます。

C# では次の操作がバインディングの対象になります。

- メンバー アクセス: `e.M`
- メソッドの呼び出し: `e.M(e1, …, en)`
- デリゲートの呼び出し: `e(e1, …, en)`
- 要素へのアクセス: `e[e1, …, en]`
- オブジェクトの作成: `new C(e1, …, en)`
- オーバーロードされた単項演算子: `+`、`-`、`!`、`~`、`++`、`--`、`true`、`false`
- オーバーロードされた二項演算子: `+`、`-`、`*`、`/`、`%`、`&`、`&&`、`|`、`||`、`??`、`^`、`<<`、`>>`、`==`、`!=`、`>`、`<`、`>=`、`<=`
- 代入演算子: `=`、`+=`、`-=`、`*=`、`/=`、`%=`、`&=`、`|=`、`^=`、`<<=`、`>>=`
- 暗黙の型変換と明示的な変換

動的式が含まれない場合は、既定で静的バインディングになります。つまり、構成する式のコンパイル時の型が選択プロセスで使用されます。一方、先に示した操作を構成する式のいずれかが動的式のときは、操作は動的にバインドされます。

### 7.2.1 バインディング時

静的バインディングはコンパイル時に行われ、動的バインディングは実行時に行われます。以降のセクションでは、"バインディング時" という用語は、バインディングが行われるときに応じて、コンパイル時または実行時を意味します。

次の例では、バインディング時の静的バインディングと動的バインディングの概念を示します。

```
object o = 5;
dynamic d = 5;

Console.WriteLine(5); // static binding to Console.WriteLine(int)
Console.WriteLine(o); // static binding to Console.WriteLine(object)
Console.WriteLine(d); // dynamic binding to Console.WriteLine(int)
```

最初の 2 つの呼び出しは、静的にバインドされます。`Console.WriteLine` のオーバーロードは、引数のコンパイル時の型に基づいて選択されます。したがって、バインディング時は "コンパイル時" です。

3 番目の呼び出しは、動的にバインドされます。`Console.WriteLine` のオーバーロードは、引数の実行時の型に基づいて選択されます。このようになるのは、引数のコンパイル時の型が `dynamic`、つまり動的な式であるためです。したがって、3 番目の呼び出しのバインディング時は "実行時" です。

### 7.2.2 動的バインディング

動的バインディングの目的は、C# プログラムが "動的オブジェクト" つまり C# の型システムの通常の規則に従わないオブジェクトと対話できるようにすることです。動的オブジェクトとしては、型システムが異なる他のプログラミング言語のオブジェクト、または異なる操作に対して独自のバイン

ディング セマンティクスを実装するようにプログラムでセットアップされているオブジェクトなどがあります。

動的オブジェクトが独自のセマンティクスを実装するメカニズムは、実装で定義されています。特定のインターフェイス(やはり実装で定義されます)は、特別なセマンティクスであることをC#のランタイムに通知するように、動的オブジェクトによって実装されます。したがって、動的オブジェクトでの操作が動的にバインドされるたびに、このドキュメントで指定されているC#のセマンティクスではなく、独自のバインディング セマンティクスが引き継がれます。

動的バインディングの目的は動的オブジェクトとの対話を可能にすることですが、C#では、動的か静的かに関係なく、すべてのオブジェクトで動的バインディングを使用できます。動的オブジェクトでの操作の結果自体は動的オブジェクトであるとは限りませんが、プログラマがコンパイル時に型を知ることはできないので、これにより円滑に動的オブジェクトを統合できます。また、動的オブジェクトが関係しない場合であっても、動的バインディングはエラーが発生しやすいリフレクションベースのコードを排除するのに役立ちます。

以降のセクションでは、動的バインディングを適用するときの言語の正確な各言語構造、適用されるコンパイル時のチェック(ある場合)、およびコンパイル時の結果と式の分類について説明します。

### 7.2.3 構成する式の型

演算が静的にバインドされる場合、構成する式(例: レシーバーと、引数、インデックス、またはオペランドなど)の型は常に、その式のコンパイル時の型であると見なされます。

演算が動的にバインドされる場合、構成する式の型は、構成する式のコンパイル時の型に応じて次の方法で決定されます。

- コンパイル時の型 `dynamic` を持つ構成する式は、実行時に式が評価される実際の値の型を持つものと見なされます。
- コンパイル時の型として型パラメーターを持つ構成する式は、実行時に型パラメーターがバインドされる型を持つものと見なされます。
- それ以外の場合は、構成する式は自身のコンパイル時の型を持つものと見なされます。

## 7.3 演算子

式は、"オペランド"と"演算子"から構成されます。式の演算子は、オペランドに適用する演算を示します。`+`、`-`、`*`、`/`、`new`などが演算子の例です。オペランドには、リテラル、フィールド、ローカル変数、式などを指定します。

演算子には、次の3種類があります。

- 単項演算子。1つのオペランドを受け取り、前置形式(`-x`など)または後置形式(`x++`など)を使用します。
- 二項演算子。2つのオペランドを受け取り、すべて infix 形式(`x + y`など)を使用します。
- 三項演算子。`?:`の1つだけです。3つのオペランドを取り、infix 形式(`c? x: y`)を使用します。

式における演算子の評価順序は、演算子の"優先順位"と"結合規則"によって決まります(7.3.1を参照)。

式のオペランドは、左から右に評価されます。たとえば、`F(i) + G(i++) * H(i)`の場合は、メソッド `F` が `i` の古い値を使って呼び出され、次にメソッド `G` が `i` の古い値で呼び出され、最後にメソッド `H`

が *i* の新しい値で呼び出されます。これは、演算子の優先順位とは異なるものであり、関係ありません。

一部の演算子は、"オーバーロード" できます。演算子をオーバーロードすると、一方または両方のオペランドがユーザー定義のクラスまたは構造体型であるような操作に対して、ユーザー定義の演算子の実装を指定できます(7.3.2 を参照)。

### 7.3.1 演算子の優先順位と結合規則

式に複数の演算子が含まれていると、演算子の "優先順位" によって、各演算子が評価される順序が制御されます。たとえば、\* 演算子は + 二項演算子よりも優先順位が高いので、*x + y \* z* という式は *x + (y \* z)* と評価されます。演算子の優先順位は、その演算子に関連する文法生成規則の定義によって定められています。たとえば、*additive-expression* は、+ 演算子または - 演算子で区切られた一連の *multiplicative-expression* で構成されるため、+ と - の演算子には、\*、/、および% の各演算子より低い優先順位が与えられています。

次の表は、演算子を優先順位の高いものから低いものの順にまとめたものです。

セクション	カテゴリ	演算子
7.6	1 次式	<i>x.y f(x) a[x] x++ x-- new typeof default checked unchecked delegate</i>
7.7	単項式	+ - ! ~ ++x --x ( <i>T</i> )x await x
7.8	乗法演算	* / %
7.8	加法演算	+
7.9	シフト	<< >>
7.10	関係式と型検査	< > <= >= is as
7.10	等価比較	== !=
7.11	論理 AND	&
7.11	論理 XOR	^
7.11	論理 OR	
7.12	条件 AND	&&
7.12	条件 OR	
7.13	Null 合体	??
7.14	条件	?:
7.17, 7.15	代入とラムダ式	= *= /= %= += -= <=> &= ^=  = =>

同じ優先順位の 2 つの演算子の間にオペランドがある場合は、演算子の結合規則によって、演算の実行順序が制御されます。

- 代入演算子と null 合体演算子を除くすべての二項演算子の結合規則は、"左から右" です。つまり、演算は左から右に実行されます。たとえば、 $x + y + z$  は  $(x + y) + z$  と評価されます。
- 代入演算子、null 合体演算子、および条件演算子 (`?:`) の結合規則は "右から左" です。つまり、演算は右から左に実行されます。たとえば、 $x = y = z$  は  $x = (y = z)$  と評価されます。

優先順位と結合規則は、かっこを使って制御できます。たとえば、 $x + y * z$  という式の場合は、最初に  $y$  に  $z$  が乗算され、その結果に  $x$  が加算されます。一方、 $(x + y) * z$  という式の場合は、最初に  $x$  と  $y$  が加算され、その結果に  $z$  が乗算されます。

### 7.3.2 演算子のオーバーロード

すべての単項演算子と二項演算子には、任意の式で自動的に使用できる定義済みの実装があります。定義済みの実装に加えて、クラスおよび構造体に `operator` 宣言を加えることで、ユーザー定義の実装を導入できます (10.10 を参照)。ユーザー定義の演算子の実装は、常に、定義済みの演算子の実装より優先されます。7.3.32 と 7.3.4 で説明するように、適用できるユーザー定義演算子の実装が存在しないときにだけ、定義済み演算子の実装が考慮されます。

"オーバーロードできる単項演算子" は、次のとおりです。

```
+ - ! ~ ++ -- true false
```

`true` と `false` は式の中では明示的に使用されませんが (したがって、7.3.1 の優先順位表にも記載されていません)、ブール式 (7.20 を参照) や、条件演算子 (7.14 を参照) および条件論理演算子 (7.12 を参照) が関係する式など、いくつかの式のコンテキストで呼び出されるため、単項演算子と見なされます。

"オーバーロードできる二項演算子" は、次のとおりです。

```
+ - * / % & | ^ << >> == != > < >= <=
```

オーバーロードできる演算子は、上に挙げたものだけです。特に、メンバー アクセス、メソッド呼び出し、または `=`、`&&`、`||`、`??`、`?:`、`=>`、`checked`、`unchecked`、`new`、`typeof`、`default`、`as`、および `is` の各演算子はオーバーロードできません。

二項演算子をオーバーロードすると、対応する代入演算子 (ある場合) も暗黙でオーバーロードされます。たとえば、`*` 演算子をオーバーロードすると、`*=` 演算子もオーバーロードされます。これは 7.17.2 で詳細に説明します。代入演算子自体 (`=`) はオーバーロードできないことに注意してください。代入では、常に、値がビットごとに変数にコピーされます。

(T) $x$  などのキャスト演算をオーバーロードするには、ユーザー定義の変換を使います (6.4 を参照)。

`a[x]` などの要素アクセスは、オーバーロード可能な演算子ではありません。代わりに、インデクサーを使ったユーザー定義のインデックス付けがサポートされています (10.9 を参照)。

式においては、演算子は演算子表記を使って参照され、宣言においては、演算子は関数表記を使って参照されます。次の表は、単項演算子と二項演算子について、演算子表記と関数表記の関係をまとめたものです。1 行目の `op` は、オーバーロードできる単項前置演算子を表しています。2 行目の `op` は、単項後置演算子の `++` と `--` を表しています。3 行目の `op` は、オーバーロードできる二項演算子を表しています。

演算子表記	関数表記
$op\ x$	<code>operator op(x)</code>
$x\ op$	<code>operator op(x)</code>
$x\ op\ y$	<code>operator op(x, y)</code>

ユーザー定義演算子の宣言では、常に、少なくとも 1 つのパラメーターで、演算子の宣言を含むクラス型または構造体型を指定する必要があります。したがって、ユーザー定義演算子のシグネチャを定義済み演算子と同じにすることはできません。

ユーザー定義演算子の宣言では、演算子の構文、優先順位、または結合規則を変更することはできません。たとえば、/ 演算子は、常に二項演算子で、常に 7.3.1 で指定されている優先順位レベルを持ち、常に結合規則は左から右です。

ユーザー定義演算子ではどのような計算でも実行できますが、直感的に予想されるものと異なる結果を生成するような実装は避けることをお勧めします。たとえば、`operator ==` の実装は、2 つのオペランドが等しいかどうかを比較して適切な `bool` 値を返すものにする必要があります。

7.6 から 7.12 の個々の演算子の説明では、演算子の定義済み実装と各演算子に適用される追加の規則を示しています。この説明の中では、"単項演算子のオーバーロードの解決"、"二項演算子のオーバーロードの解決"、および "数値の上位変換" という用語が使われます。次のセクションでは、これらの用語の定義を示します。

### 7.3.3 単項演算子のオーバーロードの解決

オーバーロード可能な単項演算子  $op$  と  $X$  型の式  $x$  を使用する  $op\ x$  または  $x\ op$  の形式の演算は、以下の方法で処理されます。

- 演算 `operator op(x)` に対して  $x$  で宣言されているユーザー定義演算子の候補のセットを、7.3.5 の規則を使用して決定します。
- ユーザー定義演算子の候補の集合が空でない場合は、これがその演算に対する演算子候補の集合になります。空の場合は、リフト形式も含めて、定義済み単項演算子 `operator op` の実装が、その演算に対する演算子候補の集合になります。特定の演算子の定義済み実装については、演算子の説明で示します(7.6 および 7.7 を参照)。
- 7.5.3 のオーバーロードの解決規則がこの候補演算子のセットに適用されると、引数リスト ( $x$ ) に対して最適な演算子が選択され、この演算子がオーバーロードの解決処理の結果になります。オーバーロードの解決が、单一の最適な演算子を選択できない場合は、バインディング時エラーになります。

### 7.3.4 二項演算子のオーバーロードの解決

オーバーロード可能な二項演算子  $op$ 、 $X$  型の式  $x$ 、および  $Y$  型の式  $y$  を使用する  $x\ op\ y$  の形式の演算は、以下の方法で処理されます。

- 演算 `operator op(x, y)` に対して  $x$  および  $y$  で宣言されているユーザー定義演算子の候補のセットを決定します。この集合は、 $x$  で宣言されている演算子候補と  $y$  で宣言されている演算子候補を結合したもので、それぞれの候補は 7.3.5 の規則を使って決定します。結合された集合の場合、候補は次のようにマージされます。

- $x$  と  $y$  が同じ型の場合、または  $x$  と  $y$  が共通の基本型から派生している場合、両方に含まれる演算子候補は、結合された集合では 1 回だけ示されます。
- $x$  と  $y$  の間に恒等変換があり、 $y$  によって提供される演算子  $op_y$  が  $x$  によって提供される  $op_x$  と同じ戻り値の型を持ち、 $op_y$  が持つオペランドの型が  $op_x$  の対応するオペランドの型への恒等変換を持つ場合、 $op_x$  のみが集合内で発生します。
- ユーザー定義演算子の候補の集合が空でない場合は、これがその演算に対する演算子候補の集合になります。空の場合は、リフト形式も含めて、定義済み二項演算子 **operator**  $op$  の実装が、その演算に対する演算子候補の集合になります。特定の演算子の定義済み実装については、演算子の説明で示します (7.8 から 7.12 を参照)。定義済みの列挙型演算子とデリゲート演算子に対して、いずれかのオペランドのバインディング時の型である列挙型またはデリゲート型によって定義される演算子のみが考慮されます。
- 7.5.3 のオーバーロードの解決規則がこの演算子候補のセットに適用されると、引数リスト ( $x, y$ ) に対して最適な演算子が選択され、この演算子がオーバーロードの解決処理の結果になります。オーバーロードの解決が、单一の最適な演算子を選択できない場合は、バインディング時エラーになります。

### 7.3.5 ユーザー定義演算子候補

型  $T$  と演算 **operator**  $op(A)$  ( $op$  はオーバーロード可能な演算子、 $A$  は引数リスト) を例にすると、**operator**  $op(A)$  に対して  $T$  で宣言されているユーザー定義演算子候補は、次の方法で決定されます。

- 型  $T_0$  を決定します。 $T$  が **null** 許容型の場合、 $T_0$  はその基になる型です。それ以外の場合、 $T_0$  は  $T$  に等しくなります。
- $T_0$  におけるすべての **operator**  $op$  宣言、およびそのような演算子のすべてのリフト形式について、引数リスト  $A$  に関して適用可能な (7.5.3.1 を参照) 演算子が少なくとも 1 つある場合、演算子候補の集合は、 $T_0$  における適用可能なすべての演算子で構成されます。
- それ以外の場合、 $T_0$  が **object** であれば、演算子候補のセットは空になります。
- それ以外の場合、 $T_0$  によって宣言される演算子候補の集合は、 $T_0$  の直接基底クラスによって宣言される演算子候補の集合になります。また、 $T_0$  が型パラメーターである場合は、 $T_0$  の実質的な基底クラスによって宣言される演算子候補の集合になります。

### 7.3.6 数値の上位変換

数値の上位変換では、定義済みの単項数値演算子または二項数値演算子のオペランドに対して特定の暗黙変換が自動的に行われます。数値の上位変換は、独立した機能ではなく、定義済み演算子に対してオーバーロードの解決を適用することによる効果です。数値の上位変換は、ユーザー定義演算子の評価には影響を及ぼしませんが、同様の効果を示すように、ユーザー定義演算子を実装できます。

数値の上位変換の例として、二項演算子  $*$  の定義済み実装を考えます。

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
float operator *(float x, float y);
double operator *(double x, double y);
decimal operator *(decimal x, decimal y);
```

オーバーロードの解決規則 (7.5.3 を参照) がこの演算子集合に適用されると、オペラントの型からの暗黙変換が存在する最初の演算子が選択されます。たとえば、`b * s` という演算 (`b` は `byte`、`s` は `short`) の場合、オーバーロードの解決では、最適な演算子として `operator *(int, int)` が選択されます。したがって、`b` と `s` は `int` に変換されて、結果の型は `int` になります。同様に、`i * d` という演算 (`i` は `int`、`d` は `double`) の場合、オーバーロードの解決では、最適な演算子として `operator *(double, double)` が選択されます。

### 7.3.6.1 単数値上位変換

定義済み単項演算子 `+`、`-`、および `~` のオペラントに対しては、単項数値上位変換が行われます。単項数値上位変換は、`sbyte`、`byte`、`short`、`ushort`、または `char` の各型のオペラントを `int` 型に変換するだけです。さらに、単項演算子 `-` に対しては、単項数値上位変換は `uint` 型のオペラントを `long` 型に変換します。

### 7.3.6.2 二項数値上位変換

定義済み二項演算子 `+`、`-`、`*`、`/`、`%`、`&`、`|`、`^`、`==`、`!=`、`>`、`<`、`>=`、および `<=` のオペラントに対しては、二項数値上位変換が行われます。二項数値上位変換では、両方のオペラントが共通の型に変換されて、関係演算子以外の場合は、それが演算の結果の型にもなります。二項数値上位変換では、以下の規則がこの順序で適用されます。

- 一方のオペラントが `decimal` 型の場合、もう一方のオペラントが `decimal` 型に変換されます。ただし、もう一方のオペラントが `float` 型または `double` 型の場合は、バインディング エラーになります。
- 一方のオペラントが `double` 型の場合、もう一方のオペラントが `double` 型に変換されます。
- 一方のオペラントが `float` 型の場合、もう一方のオペラントが `float` 型に変換されます。
- 一方のオペラントが `ulong` 型の場合、もう一方のオペラントが `ulong` 型に変換されます。ただし、もう一方のオペラントが `sbyte` 型、`short` 型、`int` 型、または `long` 型の場合は、バインディング時のエラーになります。
- 一方のオペラントが `long` 型の場合、もう一方のオペラントが `long` 型に変換されます。
- 一方のオペラントが `uint` 型で、もう一方のオペラントが `sbyte` 型、`short` 型、または `int` 型の場合は、両方のオペラントが `long` 型に変換されます。
- 一方のオペラントが `uint` 型の場合、もう一方のオペラントが `uint` 型に変換されます。
- 上記のいずれでもない場合は、両方のオペラントが `int` 型に変換されます。

1 番目の規則により、`decimal` 型と `double` 型や `float` 型を混合した演算は禁止されています。この規則は、`decimal` 型と `double` 型や `float` 型の間には暗黙的な変換がないという事実によるものです。

また、一方のオペラントが符号付き整数型の場合は、もう一方のオペラントとして `ulong` 型が認められないことにも注意してください。これは、符号付き整数型と同様に `ulong` の全範囲を表現できる整数型が存在していないためです。

上のどちらの場合も、キャスト式を使って、一方のオペラントをもう一方のオペラントと互換性のある型に明示的に変換できます。

次に例を示します。

```
decimal AddPercent(decimal x, double percent) {
    return x * (1.0 + percent / 100.0);
}
```

この場合は、`decimal` に `double` を掛けることはできないため、バインディング エラーが発生します。2 番目のオペランドを `decimal` に明示的に変換することで、このエラーを解決できます。次に例を示します。

```
decimal AddPercent(decimal x, double percent) {
    return x * (decimal)(1.0 + percent / 100.0);
}
```

### 7.3.7 リフト演算子

“リフト演算子”を使用すると、`null` 非許容の値型を操作する定義済み演算子およびユーザー定義演算子を、`null` 非許容の値型の `null` 許容形式で使用できます。リフト演算子は、次の説明のように、いくつかの必要条件に適合する定義済み演算子とユーザー定義演算子から構成されます。

- 次の単項演算子の場合

`+ ++ - -- ! ~`

オペランドの型と結果の型の両方が `null` 非許容の値型であれば、これらの演算子のリフト形式が存在します。リフト形式を構築するには、オペランドの型と結果の型に单一の`?`修飾子を追加します。リフト演算子は、オペランドが `null` の場合は `null` 値を生成します。それ以外の場合、リフト演算子はオペランドをラップ解除し、基になる演算子を適用し、結果をラップします。

- 次の二項演算子の場合

`+ - * / % & | ^ << >>`

オペランドの型と結果の型がすべて `null` 非許容の値型であれば、これらの演算子のリフト形式が存在します。リフト形式を構築するには、各オペランドおよび結果の型に单一の`?`修飾子を追加します。1つまたは両方のオペランドが `null` の場合(例外である `bool?` 型の`&`および`|`演算子については 7.11.3 を参照)、リフト演算子は `null` 値を生成します。それ以外の場合、リフト演算子はオペランドをラップ解除し、基になる演算子を適用し、結果をラップします。

- 次の等値演算子の場合

`== !=`

オペランドの型が両方とも `null` 非許容の値型で結果の型が `bool` であれば、これらの演算子のリフト形式が存在します。リフト形式を構築するには、各オペランドの型に单一の`?`修飾子を追加します。リフト演算子では、2つの `null` 値は等しく、`null` 値と任意の非 `null` 値は等しくないと見なされます。両方のオペランドが非 `null` の場合は、リフト演算子がオペランドをラップ解除し、基になる演算子を適用して `bool` 型の結果を作成します。

- 関係演算子の場合

`< > <= >=`

オペランドの型が両方とも `null` 非許容の値型で結果の型が `bool` であれば、これらの演算子のリフト形式が存在します。リフト形式を構築するには、各オペランドの型に单一の`?`修飾子を追加します。1つまたは両方のオペランドが `null` の場合、リフト演算子は値 `false` を生成します。それ以外の場合、リフト演算子はオペランドをラップ解除し、基になる演算子を適用し、`bool` 型の結果を生成します。

## 7.4 メンバー検索

メンバー検索は、型のコンテキストにおける名前の意味を決定するための処理です。メンバー検索は、式の *simple-name* (7.6.2 を参照) または *member-access* (7.6.4 を参照) の評価の一部として行われる場合があります。*simple-name* または *member-access* が *invocation-expression* (7.6.5.1 を参照) の *primary-expression* として発生する場合、メンバーが "呼び出された" と言います。

メンバーがメソッドまたはイベントである場合、あるいはデリゲート型 (15 を参照) または型 *dynamic* (4.7 を参照) の定数、フィールド、またはプロパティである場合、メンバーは "呼び出し可能である" と言います。

メンバー検索では、メンバーの名前だけでなく、メンバーが持っている型パラメーターの数、およびメンバーがアクセス可能かどうかも考慮されます。メンバー検索への対応として、ジェネリックメソッドおよび入れ子になったジェネリック型はそれぞれの宣言に型パラメーターの数が示され、他のすべてのメンバーの型パラメーターの数はゼロになっています。

型 *T* の *K* 型パラメーターを持つ名前 *N* に対するメンバー検索は、次のように処理されます。

- まず、*N* という名前のアクセス可能なメンバーの集合を決定します。
  - *T* が型パラメーターの場合、この集合は、*object* 内の *N* という名前のアクセス可能なメンバーの集合と、*T* の主制約または 2 次制約 (10.1.5 を参照) として指定された型のそれぞれに含まれる *N* という名前のアクセス可能なメンバーの集合の和集合になります。
  - それ以外の場合、この集合は、*object* 内の *N* という名前の継承メンバーやアクセス可能なメンバーなど、*T* 内の *N* という名前のすべてのアクセス可能な (3.5 を参照) メンバーで構成されます。*T* が構築された型の場合、メンバーの集合は、10.3.2 で説明されている型引数を代入して取得します。*override* 修飾子を含むメンバーは、集合から除外します。
- 次に、*K* がゼロの場合は、宣言に型パラメーターを含むすべての入れ子になった型を削除します。*K* がゼロでない場合は、型パラメーターの数が異なるすべてのメンバーを削除します。*K* がゼロの場合は、型パラメーターを持つメソッドは削除されません。これは、型推論プロセス (7.5.2 を参照) で型引数を推論できる可能性があるためです。
- 次に、メンバーが呼び出されると、呼び出し不可のメンバーはすべて集合から削除されます。
- 次に、他のメンバーによって隠ぺいされているメンバーを集合から削除します。集合内の *S.M* (*S* は、メンバー *M* が宣言されている型) というすべてのメンバーに対し、次の規則を適用します。
  - *M* が定数、フィールド、プロパティ、イベント、または列挙のいずれかのメンバーである場合は、*S* の基本型で宣言されているすべてのメンバーを集合から削除します。
  - *M* が型宣言である場合は、*S* の基本型で宣言されている型以外のすべての型を集合から削除し、*S* の基本型で宣言されている *M* と同じ数の型パラメーターを持つすべての型宣言を集合から削除します。
  - *M* がメソッドの場合は、*S* の基本型で宣言されているメソッド以外のすべてのメンバーを集合から削除します。
- 次に、クラス メンバーによって隠ぺいされているインターフェイス メンバーを集合から削除します。この手順は、*T* が型パラメーターであり、*T* が *object* 以外の実質的な基底クラスと空白でない有効なインターフェイス セットの両方を持つ場合にのみ有効です (10.1.5 を参照)。集合内の

$S.M$  ( $S$  は、メンバー  $M$  が宣言されている型) というすべてのメンバーに対して、 $S$  が `object` 以外のクラス宣言の場合は、次の規則を適用します。

- $M$  が定数、フィールド、プロパティ、イベント、列挙メンバー、または型宣言の場合は、インターフェイス宣言に宣言されているすべてのメンバーを集合から削除します。
- $M$  がメソッドである場合は、インターフェイス宣言に宣言されているメソッド以外のすべてのメンバーを集合から削除し、インターフェイス宣言に宣言されている  $M$  と同じシグネチャを持つすべてのメソッドを集合から削除します。
- 最後に、隠ぺいされたメンバーを削除して、検索の結果を決定します。
  - メソッドではないメンバー 1 つのみによって集合が構成されている場合は、そのメンバーが検索の結果です。
  - 集合に含まれているのがメソッドだけである場合は、そのメソッドのグループが検索の結果です。
  - 上記以外の場合、検索結果はあいまいであります。バインディング エラーになります。

型パラメータとインターフェイス以外の型におけるメンバー検索、および厳密な单一継承であるインターフェイス（継承チェーン内の各インターフェイスが厳密に 0 または 1 つの直接基本インターフェイスを持つ場合）におけるメンバー検索では、検索規則を適用しても、派生メンバーによって同じ名前またはシグネチャを持つ基本メンバーが隠ぺいされるだけです。このような单一継承の検索は、不確定になることはありません。多重継承インターフェイスでのメンバー検索で発生する可能性のある不確定性については、13.2.5 で説明します。

#### 7.4.1 基本型

メンバー検索を行うにあたって、型  $T$  は次のような基本型を持つと見なされます。

- $T$  が `object` または `dynamic` の場合、 $T$  には基本型はありません。
- $T$  が *enum-type* の場合、 $T$  の基本型は、`System.Enum` クラス型、`System.ValueType` クラス型、および `object` クラス型です。
- $T$  が *struct-type* の場合、 $T$  の基本型は、`System.ValueType` クラス型および `object` クラス型です。
- $T$  が *class-type* の場合、 $T$  の基本型は、`object` クラス型を含む  $T$  の基底クラスです。
- $T$  が *interface-type* の場合、 $T$  の基本型は、 $T$  の基本インターフェイスと `object` クラス型です。
- $T$  が *array-type* の場合、 $T$  の基本型は、`System.Array` クラス型および `object` クラス型です。
- $T$  が *delegate-type* の場合、 $T$  の基本型は、`System.Delegate` クラス型および `object` クラス型です。

#### 7.5 関数メンバー

関数メンバーとは、実行可能なステートメントを含むメンバーです。関数メンバーは、常に型のメンバーであり、名前空間のメンバーになることはできません。C# では、次のカテゴリの関数メンバーが定義されています。

- メソッド
- プロパティ

- イベント
- インデクサー
- ユーザー定義演算子
- インスタンス コンストラクター
- 静的コンストラクター
- デストラクター

明示的に呼び出すことができないデストラクターと静的コンストラクターを除いて、関数メンバーに含まれるステートメントは、関数メンバーの呼び出しを通じて実行されます。関数メンバーの呼び出しを記述するための実際の構文は、関数メンバーのカテゴリに依存します。

関数メンバー呼び出しの引数リスト (7.5.1 を参照) は、関数メンバーのパラメーターに対して実際の値または変数参照を渡します。

ジェネリック メソッドの呼び出しでは、型推論を使用して、メソッドに渡す一連の型引数を判断できます。これは 7.5.2 で説明します。

メソッド、インデクサー、演算子、およびインスタンス コンストラクターの呼び出しでは、オーバーロードの解決を使用して、関数メンバーの候補から呼び出すメンバーを決定します。これは 7.5.3 で説明します。

バインディング時にオーバーロードの解決を行うなど、特定の関数メンバーを識別した後の、実行時に行われる実際の関数メンバー呼び出し処理については、7.5.4 を参照してください。

次の表は、明示的に呼び出すことができる関数メンバーの 6 つのカテゴリを含む構造で行われる処理をまとめたものです。この表では、`e`、`x`、`y`、および `value` は変数または値に分類される式、`T` は型に分類される式、`F` はメソッドの簡易名、`P` はプロパティの簡易名をそれぞれ表しています。

構成要素	例	説明
メソッドの呼び出し	<code>F(x, y)</code>	包含クラスまたは構造体の最適なメソッド <code>F</code> を選択するために、オーバーロードの解決が適用されます。メソッドが、引数リスト <code>(x, y)</code> を使って呼び出されます。メソッドが <code>static</code> ではない場合、インスタンス式は <code>this</code> です。
	<code>T.F(x, y)</code>	クラスまたは構造体である <code>T</code> の最適なメソッド <code>F</code> を選択するために、オーバーロードの解決が適用されます。メソッドが <code>static</code> ではない場合は、バインディング エラーになります。メソッドが、引数リスト <code>(x, y)</code> を使って呼び出されます。
	<code>e.F(x, y)</code>	<code>e</code> の型で与えられるクラス、構造体、またはインターフェイスの最善のメソッド <code>F</code> を選択するために、オーバーロードの解決が適用されます。メソッドが <code>static</code> の場合は、バインディング エラーになります。メソッドが、インスタンス式 <code>e</code> と引数リスト <code>(x, y)</code> を使って呼び出されます。

構成要素	例	説明
プロパティ アクセス	P	包含クラスまたは構造体のプロパティ P の <b>get</b> アクセサーが呼び出されます。P が書き込み専用の場合は、コンパイルエラーになります。P が <b>static</b> ではない場合、インスタンス式は <b>this</b> です。
	P = value	包含クラスまたは構造体のプロパティ P の <b>set</b> アクセサーが、引数リスト ( <b>value</b> ) を使って呼び出されます。P が読み取り専用の場合は、コンパイルエラーになります。P が <b>static</b> ではない場合、インスタンス式は <b>this</b> です。
	T.P	クラスまたは構造体である T のプロパティ P の <b>get</b> アクセサーが呼び出されます。P が <b>static</b> ではない場合、または P が書き込み専用の場合は、コンパイル時のエラーになります。
	T.P = value	クラスまたは構造体である T のプロパティ P の <b>set</b> アクセサーが、引数リスト ( <b>value</b> ) を使って呼び出されます。P が <b>static</b> ではない場合、または P が読み取り専用の場合は、コンパイル時のエラーになります。
	e.P	e の型で与えられるクラス、構造体、またはインターフェイスのプロパティ P の <b>get</b> アクセサーが、インスタンス式 e を使って呼び出されます。P が <b>static</b> の場合、または P が書き込み専用の場合は、バインディング時のエラーになります。
	e.P = value	e の型で与えられるクラス、構造体、またはインターフェイスのプロパティ P の <b>set</b> アクセサーが、インスタンス式 e と引数リスト ( <b>value</b> ) を使って呼び出されます。P が <b>static</b> の場合、または P が読み取り専用の場合は、バインディング時のエラーになります。
イベントア クセス	E += value	包含クラスまたは構造体のイベント E の <b>add</b> アクセサーが呼び出されます。E が <b>static</b> ではない場合、インスタンス式は <b>this</b> です。
	E -= value	包含クラスまたは構造体のイベント E の <b>remove</b> アクセサーが呼び出されます。E が <b>static</b> ではない場合、インスタンス式は <b>this</b> です。
	T.E += value	クラスまたは構造体である T のイベント E の <b>add</b> アクセサーが呼び出されます。E が <b>static</b> ではない場合は、バインディングエラーになります。
	T.E -= value	クラスまたは構造体である T のイベント E の <b>remove</b> アクセサーが呼び出されます。E が <b>static</b> ではない場合は、バインディングエラーになります。

構成要素	例	説明
	<code>e.E += value</code>	<code>e</code> の型で与えられるクラス、構造体、またはインターフェイスのイベント <code>E</code> の <code>add</code> アクセサーが、インスタンス式 <code>e</code> を使って呼び出されます。 <code>E</code> が <code>static</code> の場合は、バインディング エラーになります。
	<code>e.E -= value</code>	<code>e</code> の型で与えられるクラス、構造体、またはインターフェイスのイベント <code>E</code> の <code>remove</code> アクセサーが、インスタンス式 <code>e</code> を使って呼び出されます。 <code>E</code> が <code>static</code> の場合は、バインディング エラーになります。
インデクサー アクセス	<code>e[x, y]</code>	<code>e</code> の型で与えられるクラス、構造体、またはインターフェイスの最善のインデクサーを選択するために、オーバーロードの解決が適用されます。インデクサーの <code>get</code> アクセサーが、インスタンス式 <code>e</code> と引数リスト <code>(x, y)</code> を使って呼び出されます。インデクサーが書き込み専用の場合は、バインディング エラーになります。
	<code>e[x, y] = value</code>	<code>e</code> の型で与えられるクラス、構造体、またはインターフェイスの最善のインデクサーを選択するために、オーバーロードの解決が適用されます。インデクサーの <code>set</code> アクセサーが、インスタンス式 <code>e</code> と引数リスト <code>(x, y, value)</code> を使って呼び出されます。インデクサーが読み取り専用の場合は、バインディング エラーになります。
演算子の呼び出し	<code>-x</code>	<code>x</code> の型で与えられるクラスまたは構造体の最善の単項演算子を選択するために、オーバーロードの解決が適用されます。選択された演算子は、引数リスト <code>(x)</code> を使って呼び出されます。
	<code>x + y</code>	<code>x</code> および <code>y</code> の型で与えられるクラスまたは構造体の最善の二項演算子を選択するために、オーバーロードの解決が適用されます。選択された演算子は、引数リスト <code>(x, y)</code> を使って呼び出されます。
インスタンス コンストラクターの呼び出し	<code>new T(x, y)</code>	クラスまたは構造体である <code>T</code> の最適なインスタンス コンストラクターを選択するために、オーバーロードの解決が適用されます。インスタンス コンストラクターが、引数リスト <code>(x, y)</code> を使って呼び出されます。

### 7.5.1 引数リスト

すべての関数メンバーおよびデリゲートの呼び出しには、関数メンバーのパラメーターに対する実際の値または変数参照を渡す引数リストが含まれます。関数メンバー呼び出しの引数リストを指定するための構文は、関数メンバーのカテゴリに依存します。

- インスタンス コンストラクター、メソッド、インデクサー、およびデリゲートの場合の引数は、以下で説明するように *argument-list* として指定します。インデクサーの場合、`set` アクセサーを

呼び出すときは、代入演算子の右オペランドとして指定された式が、引数リストにさらに追加されます。

- プロパティの場合の引数リストは、`get` アクセサーを呼び出すときは空で、`set` アクセサーを呼び出すときは代入演算子の右オペランドとして指定された式で構成されます。
- イベントの場合の引数リストは、`+ =` 演算子または`- =` 演算子の右オペランドとして指定された式で構成されます。
- ユーザー定義演算子の場合の引数リストは、単項演算子の1つのオペランド、または二項演算子の2つのオペランドで構成されます。

プロパティ(10.7を参照)、イベント(10.8を参照)、およびユーザー定義の演算子(10.10を参照)は常に値パラメーター(10.6.1.1を参照)として渡されます。インデクサーの引数(10.9を参照)は、常に値パラメーター(10.6.1.1を参照)またはパラメーター配列(10.6.1.4を参照)として渡されます。これらのカテゴリの関数メンバーでは、参照パラメーターおよび出力パラメーターはサポートされていません。

インスタンス コンストラクター、メソッド、インデクサー、またはデリゲートの呼び出しの引数は、*argument-list* として指定されます。

```

argument-list:
  argument
  argument-list , argument

argument:
  argument-nameopt argument-value

argument-name:
  identifier :

argument-value:
  expression
  ref variable-reference
  out variable-reference

```

*argument-list* は、コンマで区切られた1つ以上の *argument* で構成されます。各引数は、オプションの *argument-name* とそれに続く *argument-value* で構成されます。*argument-name* のある *argument* は "名前付き引数" と呼ばれ、*argument-name* のない *argument* は "位置指定引数" と呼ばれます。*argument-list* で名前付き引数の後に位置指定引数があるとエラーになります。

*argument-value* は、次のいずれかの形式です。

- *expression*。引数が値パラメーター(10.6.1.1を参照)として渡されることを示します。
- キーワード **ref** とそれに続く *variable-reference*(5.4を参照)。引数が参照パラメーター(10.6.1.2を参照)として渡されることを示します。変数は、参照パラメーターとして渡すことができるようになる前に、明示的に代入する必要があります(5.3を参照)。
- キーワード **out** とそれに続く *variable-reference*(5.4を参照)。引数が出力パラメーター(10.6.1.3を参照)として渡されることを示します。変数は、出力パラメーターとして変数が渡される関数メンバー呼び出しの後では確実に代入されている(5.3を参照)ものと見なされます。

これにより、引数のパラメーター受け渡しモード(値、参照、または出力)がそれぞれ決定されます。

### 7.5.1.1 対応するパラメーター

引数リストの各引数に対応するパラメーターが、呼び出される関数メンバーまたはデリゲートに存在している必要があります。

使用されるパラメーター リストは次のように決定されます。

- クラスで定義されている仮想メソッドおよびインデクサーの場合、パラメーター リストは最も限定的な宣言または関数メンバーのオーバーライドから選択されます。 static 型のレシーバーから始めて、各基底クラスが検索されます。
- インターフェイスメソッドおよびインデクサーの場合、パラメーター リストはメンバーの最も限定的な定義から選択されます。インターフェイス型から始めて、各基本インターフェイスが検索されます。一意のパラメーター リストが見つからない場合は、アクセスできない名前を持ち、省略可能なパラメーターのないパラメーター リストが構築されます。これにより、呼び出しで名前付きパラメーターを使用したり、省略可能な引数を省略したりすることはできなくなります。
- 部分メソッドの場合、部分メソッド定義宣言のパラメーター リストが使用されます。
- 他のすべての関数メンバーおよびデリゲートの場合、パラメーター リストは 1 つしか存在せず、それが使用されます。

引数またはパラメーターの位置は、引数リストまたはパラメーター リストでその引数またはパラメーターより前にある引数またはパラメーターの数として定義されます。

関数メンバーの引数に対応するパラメーターは、次のように決定されます。

- インスタンス コンストラクター、メソッド、インデクサー、またはデリゲートの *argument-list* の引数:
  - パラメーター リスト内の同じ位置に固定パラメーターが出現する位置指定引数は、そのパラメーターに対応します。
  - 通常形式で呼び出されるパラメーター配列のある関数メンバーの位置指定引数は、パラメーター配列に対応します。パラメーター リスト内で同じ位置に出現する必要があります。
  - 展開形式で呼び出されるパラメーター配列のある関数メンバーの位置指定引数で、パラメーター リスト内の同じ位置に固定パラメーターが出現しない場合は、パラメーター配列の要素に対応します。
  - 名前付き引数は、パラメーター リスト内の同じ名前のパラメーターに対応します。
  - インデクサーの場合、**set** アクセサー呼び出すときは、代入演算子の右オペランドとして指定されている式が、**set** アクセサーの宣言の暗黙の **value** パラメーターに対応します。
- プロパティの場合、**get** アクセサーの呼び出しには引数はありません。**set** アクセサー呼び出すときは、代入演算子の右オペランドとして指定されている式が、**set** アクセサーの宣言の暗黙の **value** パラメーターに対応します。
- ユーザー定義の単項演算子の場合 (変換を含む)、1 つのオペランドが演算子の宣言の 1 つのパラメーターに対応します。
- ユーザー定義の二項演算子の場合、左オペランドは演算子宣言の 1 番目のパラメーターに対応し、右オペランドは 2 番目のパラメーターに対応します。

### 7.5.1.2 引数リストの実行時の評価

関数メンバー呼び出し (7.5.4 を参照) の実行時の処理中には、引数リストの式または変数参照が、左から右の順序で、次に示すように評価されます。

- 値パラメーターの場合は、引数式が評価されて、対応するパラメーター型への暗黙の変換 (0 を参照) が実行されます。結果の値が、関数メンバー呼び出しにおける値パラメーターの初期値になります。
- 参照パラメーターまたは出力パラメーターの場合は、変数参照が評価されて、結果の格納場所が、関数メンバー呼び出しのパラメーターで示される格納場所になります。参照パラメーターまたは出力パラメーターとして与えられた変数参照が *reference-type* の配列要素である場合は、配列の要素型とパラメーターの型が同じであることを確認するために、ランタイム チェックが実行されます。このチェックで問題があると、`System.ArrayTypeMismatchException` がスローされます。

メソッド、インデックサ、およびインスタンス コンストラクターでは、右端のパラメーターをパラメーター配列 (10.6.1.4 を参照) として宣言できます。このような関数メンバーは、標準形式と展開形式のどちらが適用可能であるか (7.5.3.1 を参照) に基づいて、どちらかの形式で呼び出されます。

- パラメーター配列を持つ関数メンバーが標準形式で呼び出されるときは、パラメーター配列で指定される引数は、パラメーター配列の型に暗黙で変換できる (0 を参照) 単一の式である必要があります。この場合、パラメーター配列は値パラメーターとまったく同じように機能します。
- パラメーター配列を持つ関数メンバーが展開形式で呼び出されるときは、パラメーター配列に対する 0 以上の位置指定引数が呼び出しで指定されている必要があります。このときの各引数は、パラメーター配列の要素型に暗黙で変換できる (0 を参照) 式です。この場合、呼び出しによって、引数の数に対応した長さを持つパラメーター配列型のインスタンスが作成され、配列インスタンスの要素が指定の引数值で初期化され、新規作成された配列インスタンスが実際の引数として使用されます。

引数リストの式は、常に、記述されている順序で評価されます。次に例を示します。

```
class Test
{
    static void F(int x, int y = -1, int z = -2) {
        System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
    }

    static void Main() {
        int i = 0;
        F(i++, i++, i++);
        F(z: i++, x: i++);
    }
}
```

この例では、次のように出力されます。

```
x = 0, y = 1, z = 2
x = 4, y = -1, z = 3
```

配列の共変性規則 (12.5 を参照) により、配列型の `A[]` から `B[]` に対する暗黙の参照変換が存在している場合は、配列型 `B` の値を配列型 `A` のインスタンスへの参照にできます。この規則により、*reference-type* の配列要素が参照パラメーターまたは出力パラメーターとして渡されたときは、配列の実際の要素型がパラメーターの型と一致することを確認するためのランタイム チェックが必要です。次に例を示します。

```

class Test
{
    static void F(ref object x) {...}

    static void Main() {
        object[] a = new object[10];
        object[] b = new string[10];
        F(ref a[0]);           // Ok
        F(ref b[1]);           // ArrayTypeMismatchException
    }
}

```

`F` の 2 番目の呼び出しでは `System.ArrayTypeMismatchException` がスローされます。これは `b` の実際の要素型が `string` であり、`object` ではないためです。

パラメーター配列を含む関数メンバーが展開形式で呼び出されるときは、展開されたパラメーターの周囲に、配列初期化子 (7.6.10.4 を参照) のある配列作成式が挿入された場合とまったく同様に呼び出しが処理されます。たとえば、次のような宣言について考えます。

```
void F(int x, int y, params object[] args);
```

次に示すのは、メソッドを展開形式で呼び出す例です。

```

F(10, 20);
F(10, 20, 30, 40);
F(10, 20, 1, "hello", 3.0);

```

上の呼び出しは、次の呼び出しに厳密に対応します。

```

F(10, 20, new object[] {});
F(10, 20, new object[] {30, 40});
F(10, 20, new object[] {1, "hello", 3.0});

```

パラメーター配列に対して 0 個の引数が渡された場合は空の配列が作成されることに特に注意してください。

対応する省略可能なパラメーターのある関数メンバーから引数を省略すると、関数メンバー宣言の既定の引数が暗黙的に渡されます。これらは常に定数であるため、その評価が他の引数の評価順序に影響を与えることはありません。

## 7.5.2 型推論

型引数を指定せずにジェネリック メソッドを呼び出すと、"型推論" プロセスによって、呼び出しの型引数の推論が試みられます。型推論により、ジェネリック メソッドの呼び出しで簡便な構文を使用でき、冗長な型情報の指定が不要になります。たとえば、次のようなメソッド宣言について考えます。

```

class Chooser
{
    static Random rand = new Random();

    public static T Choose<T>(T first, T second) {
        return (rand.Next(2) == 0)? first: second;
    }
}

```

次のように、型引数を明示的に指定せずに `Choose` メソッドを呼び出すことができます。

```

int i = Chooser.Choose(5, 213);           // Calls Choose<int>
string s = Chooser.Choose("foo", "bar");    // Calls Choose<string>

```

型推論により、メソッドの引数から型引数 `int` と `string` が決定されます。

型推論は、メソッド呼び出し(7.6.5.1を参照)のバインディング時の処理の一部として、呼び出しのオーバーロード解決の前に実行されます。メソッドの呼び出しに特定のメソッドグループが指定され、型引数がメソッド呼び出しの一部として指定されていない場合は、メソッドグループの各ジェネリックメソッドに対して型推論が適用されます。型推論が成功した場合は、推論された型引数を使用して、その後のオーバーロード解決の型が決定されます。オーバーロード解決で、呼び出すメソッドとしてジェネリックメソッドが選択された場合は、推論された型引数が呼び出しの実際の型引数として使用されます。型を推論できなかったメソッドは、オーバーロード解決に関与しません。型推論に失敗しても、それだけではバインディングエラーは発生しません。ただし、多くの場合は、オーバーロード解決で適用可能なメソッドが見つからぬためにバインディングエラーになります。

指定した引数の数がメソッドのパラメーターの数と異なる場合は、推論はすぐに失敗します。それ以外の場合は、ジェネリックメソッドに次のシグネチャがあるものと仮定します。

$T_r M<X_1 \dots X_n>(T_1 x_1 \dots T_m x_m)$

$M(E_1 \dots E_m)$  の形式のメソッド呼び出しでは、呼び出し  $M<S_1 \dots S_n>(E_1 \dots E_m)$  が有効になるように、型パラメーター  $X_1 \dots X_n$  のそれぞれに対して一意の型引数  $S_1 \dots S_n$  を見つけることが、型推論の課題です。

推論の過程では、各型パラメーター  $x_i$  は特定の型  $S_i$  に固定されるか、関連する範囲内で固定されません。各範囲はいずれかの型  $T$  です。最初、各型変数  $x_i$  は、範囲の空の集合で固定されていません。

型推論は段階的に行われます。各フェーズでは、前のフェーズでわかったことに基づき、より多くの型変数の型引数の推論が試みられます。最初のフェーズでは、範囲について最初の推論を行いますが、2番目のフェーズでは、型変数を特定の型に固定し、範囲をさらに推論します。2番目のフェーズは数回繰り返すことがあります。

メモ: 型推論が行われるのは、ジェネリックメソッドが呼び出されたときだけとは限りません。メソッドグループの変換の型推論については 7.5.2.13 を、一連の式に共通する最適な型を見つける方法については 7.5.2.14 を参照してください。

### 7.5.2.1 第1フェーズ

メソッド引数  $E_i$  のそれぞれについて、次のように推論が行われます。

- $E_i$  が匿名関数の場合は、 $E_i$  から  $T_i$  への明示的なパラメーター型推論(7.5.2.7を参照)が行われます。
- それ以外の場合、 $E_i$  が型  $U$  で、 $x_i$  が値パラメーターであれば、下限の推論は  $U$  から  $T_i$  に対して行われます。
- それ以外の場合、 $E_i$  が型  $U$  で、 $x_i$  が `ref` または `out` パラメーターであれば、正確な推論は  $U$  から  $T_i$  に対して行われます。
- それ以外の場合、この引数に対して推論は行われません。

### 7.5.2.2 第2フェーズ

第2フェーズは次のように処理されます。

- 固定されていない型変数  $x_i$  のうち、 $x_j$  に依存しない(7.5.2.5を参照)ものはすべて固定されます(7.5.2.10を参照)。
- そのような型変数が存在しない場合は、以下の条件を満たす固定されていない型変数  $x_i$  はすべて固定されます。
  - $x_i$  に依存する型変数  $x_j$  が少なくとも1つある。

- $x_i$  に空でない範囲がある。
- そのような型変数が存在せず、固定されていない型変数がまだある場合、型推論は失敗します。
- それ以外の場合、固定されていない型変数が他になければ、型推論は成功します。
- 上記以外の場合、固定されていない型変数  $x_j$  が出力型(7.5.2.4 を参照)に含まれるが、入力型(7.5.2.3 を参照)にはそれが含まれない、対応するパラメーター型  $T_i$  を持つすべての引数  $E_i$  では、出力型推論(7.5.2.6 を参照)が  $E_i$  から  $T_i$  に対して行われます。その後、第 2 のフェーズが繰り返されます。

### 7.5.2.3 入力型

$E$  がメソッドグループまたは暗黙の型の匿名関数であり、 $T$  がデリゲート型または式ツリー型である場合、 $T$  のすべてのパラメーター型は、型  $T$  を持つ  $E$  の入力型になります。

### 7.5.2.4 出力型

$E$  がメソッドグループまたは匿名関数であり、 $T$  がデリゲート型または式ツリー型である場合、 $T$  の戻り値の型は、型  $T$  を持つ  $E$  の出力型になります。

### 7.5.2.5 依存関係

一部の引数について、型  $T_k$   $x_j$  の  $E_k$  が型  $T_k$  の入力型  $E_k$  にあり、 $x_i$  が型  $T_k$  の出力型  $E_k$  にある場合、固定されていない型変数  $x_i$  は固定されていない型変数  $x_j$  に直接依存します。

$x_j$  が  $x_i$  に直接依存する場合、または  $x_i$  が  $x_k$  に直接依存し、 $x_k$  が  $x_j$  に依存する場合、 $x_j$  は  $x_i$  に依存します。したがって、"依存" は推移的ですが、"直接依存" の再帰的閉包ではありません。

### 7.5.2.6 出力型推論

出力型推論は、式  $E$  から型  $T$  に対して次の方法で行われます。

- $E$  が推論された戻り値型  $U$  (7.5.2.12 を参照) を持つ匿名関数であり、 $T$  が戻り値型  $T_b$  を持つデリゲート型または式ツリー型である場合、下限の推論(7.5.2.9 を参照)は  $U$  から  $T_b$  に対して行われます。
- それ以外の場合、 $E$  がメソッドグループで、 $T$  がパラメーター型  $T_1 \dots T_k$  と戻り値型  $T_b$  を持つデリゲート型または式ツリー型で、型  $T_1 \dots T_k$  を持つ  $E$  のオーバーロード解決により戻り値型  $U$  の単一のメソッドが返される場合、下限の推論は  $U$  から  $T_b$  に対して行われます。
- それ以外の場合、 $E$  が型  $U$  の式であれば、下限の推論は  $U$  から  $T$  に対して行われます。
- それ以外の場合、推論は行われません。

### 7.5.2.7 明示的なパラメーター型推論

明示的なパラメーター型推論は、式  $E$  から型  $T$  に対して次の方法で行われます。

- $E$  がパラメーター型  $U_1 \dots U_k$  を持つ明示的な型の匿名関数であり、 $T$  がパラメーター型  $V_1 \dots V_k$  を持つデリゲート型または式ツリー型である場合、各  $U_i$  について、正確な推論(7.5.2.8 を参照)が  $U_i$  から対応する  $V_i$  に対して行われます。

### 7.5.2.8 正確な推論

型  $U$  からの型  $V$  の正確な推論は、次の方法で行われます。

- $v$  が固定されていない  $x_i$  の 1 つである場合、 $u$  は  $x_i$  の正確な範囲に追加されます。
  - それ以外の場合、セット  $v_1 \dots v_k$  および  $u_1 \dots u_k$  は、次のいずれかの条件が該当するかどうかをチェックすることで決定されます。
    - $v$  が配列型  $v_1[\dots]$  で、 $u$  が同じランクの配列型  $u_1[\dots]$  である。
    - $v$  が型  $v_1?$  で  $u$  が型  $u_1?$  である。
    - $v$  が構築された型  $C<v_1 \dots v_k>$  and  $u$  が構築された型  $C<u_1 \dots u_k>$  である。
- これらのいずれかに該当する場合、各  $u_i$  から対応する  $v_i$  への "正確な推論" が行われます。
- それ以外の場合、推論は行われません。

### 7.5.2.9 下限の推論

型  $u$  からの型  $v$  の下限の推論は、次の方で行われます。

- $v$  が固定されていない  $x_i$  の 1 つである場合、 $u$  は  $x_i$  の下限の範囲に追加されます。
- それ以外の場合、 $v$  が型  $v_1?$  で、 $u$  が型  $u_1?$  の式であれば、下限の推論は  $u_1$  から  $v_1$  に対して行われます。
- それ以外の場合、セット  $u_1 \dots u_k$  および  $v_1 \dots v_k$  は、次のいずれかの条件が該当するかどうかをチェックすることで決定されます。
  - $v$  が配列型  $v_1[\dots]$  で、 $u$  が同じランクの配列型  $u_1[\dots]$  である。
  - $v$  が `IEnumerable<V1>`、`ICollection<V1>`、または `IList<V1>` のいずれかで、 $u$  が 1 次元配列型  $u_1[]$  である。
  - $v$  が構築されたクラス、構造体、インターフェイス、またはデリゲート型  $C<v_1 \dots v_k>$  で、 $u$  ( $u$  が型パラメーターの場合は、その実質的な基底クラスまたはその実質的なインターフェイスセットの任意のメンバー) と同一であるか、その(直接的または間接的)継承元であるか、それが  $C<u_1 \dots u_k>$  (直接的または間接的に) を実装するような一意の型  $C<u_1 \dots u_k>$  が存在する。

("一意性"の制限は、`interface C<T>{}` `class U: C<X>, C<Y>{}` という場合に、 $u_1$  は  $X$  にも  $Y$  にもなり得るため、 $u$  から  $C<T>$  への推論が行われないことを意味します。)

これらのいずれかに該当する場合、各  $u_i$  から対応する  $v_i$  への推論は次のように行われます。

- $u_i$  が参照型であるとわかつていない場合は、"正確な推論" が行われます。
- それ以外の場合、 $u$  が配列型であれば、"下限の推論" が行われます。
- それ以外の場合、 $v$  が  $C<v_1 \dots v_k>$  であれば、推論は  $C$  の  $i$  番目の型パラメーターに依存します。
  - 共変の場合は "下限の推論" が行われます。
  - 反変の場合は "上限の推論" が行われます。
  - 不変の場合は "正確な推論" が行われます。
- それ以外の場合、推論は行われません。

### 7.5.2.10 上限の推論

型  $u$  から型  $v$  への "上限の推論" は、次のように行われます。

- $v$  が固定されていない  $x_i$  の 1 つである場合、 $u$  は  $x_i$  の上限の範囲に追加されます。
- それ以外の場合、セット  $v_1 \dots v_k$  および  $u_1 \dots u_k$  は、次のいずれかの条件が該当するかどうかをチェックすることで決定されます。
  - $u$  が配列型  $u_1[\dots]$  で、 $v$  が同じランクの配列型  $v_1[\dots]$  である。
  - $u$  が `IEnumerable<U>`、`ICollection<U>`、または `IList<U>` のいずれかで、 $v$  が 1 次元配列型  $v_1[]$  である。
  - $u$  が型  $u_1?$  で  $v$  が型  $v_1?$  である。
  - $u$  が構築されたクラス、構造体、インターフェイス、またはデリゲート型  $C<U_1 \dots U_k>$  で、 $v$  が  $C<V_1 \dots V_k>$  と同一であるか、それを(直接的または間接的に)継承するか、それを(直接的または間接的に)実装するクラス、構造体、インターフェイス、またはデリゲート型である。  
(“一意性”の制限は、`interface C<T>{}` `class V<Z>: C<X<Z>>, C<Y<Z>>{}` という場合に、 $C<U_1>$  から  $V<Q>$  への推論が行われないことを意味します。 $U_1$  から  $X<Q>$  への推論も  $Y<Q>$  への推論も行われません。)

これらのいずれかに該当する場合、各  $u_i$  から対応する  $v_i$  への推論は次のように行われます。

- $u_i$  が参照型であるとわかつていない場合は、“正確な推論”が行われます。
- それ以外の場合、 $v$  が配列型であれば、“上限の推論”が行われます。
- それ以外の場合、 $u$  が  $C<U_1 \dots U_k>$  であれば、推論は  $C$  の  $i$  番目の型パラメーターに依存します。
  - 共変の場合は “上限の推論”が行われます。
  - 反変の場合は “下限の推論”が行われます。
  - 不変の場合は “正確な推論”が行われます。
- それ以外の場合、推論は行われません。

### 7.5.2.11 固定

範囲内で固定されていない型変数  $x_i$  は、次のように固定されます。

- *candidate types*  $u_j$  は、 $x_i$  の範囲内ですべての型として開始されます。
- その後、 $x_i$  の各境界を順に調べます。 $x_i$  の各正確な境界  $u$  について、 $u$  と同一ではない型  $u_j$  はすべて候補の集合から削除されます。 $x_i$  の各下限  $u$  について、 $u$  からの暗黙の型変換が存在しない型  $u_j$  はすべて候補の集合から削除されます。 $x_i$  の各上限  $u$  について、 $u$  への暗黙の型変換が存在しない型  $u_j$  はすべて候補の集合から削除されます。
- 残りの候補型  $u_j$  の中に、その他すべての候補型からの暗黙の型変換が存在する一意の型  $v$  がある場合、 $x_i$  は  $v$  に固定されます。
- それ以外の場合は、型推論は失敗します。

### 7.5.2.12 推論された戻り値型

匿名関数  $F$  の “推論された戻り値の型” は、型推論およびオーバーロードの解決で使用されます。推論される戻り値型は、すべてのパラメーター型が、明示的に指定されているか、匿名関数の変換によって指定されているか、外側のジェネリック メソッド呼び出しでの型推論で推論されているために、明らかになっている匿名関数についてのみ決定できます。

**推論される結果の型**は次のように決定されます。

- $F$  の本体が式である場合、 $F$  の推論される戻り値の型はその式の型になります。
- $F$  の本体が *block* であり、そのブロックの **return** ステートメントに含まれる一連の式が共通する最適な型  $T$  (7.5.2.14 を参照) を持つ場合、 $F$  の推論される結果の型は  $T$  になります。
- それ以外の場合、 $F$  の結果の型は推論できません。

**推論される戻り値型**は次のように決定されます。

- $F$  が非同期で  $F$  の本体が "なし" と分類される式 (7.1 を参照) であるか、**return** ステートメントに式を含まないステートメントブロックである場合、推論される戻り値の型は `System.Threading.Tasks.Task` になります。
- $F$  が非同期で推論される結果型  $T$  である場合、推論される戻り値型は `System.Threading.Tasks.Task<T>` になります。
- $F$  が非同期以外で、推論される結果の型が  $T$  である場合、推論される戻り値の型は  $T$  になります。
- それ以外の場合、 $F$  の戻り値の型は推論できません。

匿名関数を使用した型推論の例として、`System.Linq.Enumerable` クラスで宣言された **Select** 拡張メソッドを考えます。

```
namespace System.Linq
{
    public static class Enumerable
    {
        public static IEnumerable<TResult> Select<TSource, TResult>(
            this IEnumerable<TSource> source,
            Func<TSource, TResult> selector)
        {
            foreach (TSource element in source) yield return selector(element);
        }
    }
}
```

`System.Linq` 名前空間が **using** 句でインポートされたと仮定し、`string` 型の `Name` プロパティのあるクラス `Customer` を指定すると、**Select** メソッドを使用して顧客のリストの名前を選択できます。

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

**Select** の拡張メソッド呼び出し (7.6.5.2 を参照) 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って処理します。

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

型引数は明示的に指定されていないため、型推論を使用して型の引数が推論されます。まず、引数 `customers` を `source` パラメータに関連付け、`T` を `Customer` として推論します。次に、上で説明した匿名関数の型推論プロセスを使用して、`c` に型 `Customer` を指定し、式 `c.Name` を `selector` パラメーターの戻り値の型に関連付け、`s` を `string` として推論します。したがって、呼び出しへは次のようになります。

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

結果は、型 `IEnumerable<string>` になります。

次の例では、匿名関数の型推論により、ジェネリック メソッドの呼び出しにおいて、引数間で型情報を "フロー" させる方法を示しています。メソッドは次のとおりです。

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

呼び出しの推論を入力します。

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

この型推論は、次のように処理します。まず、引数 "1:15:30" を `value` パラメーターに関連付け、`X` を `string` として推論します。次に、最初の匿名関数のパラメーター `s` に推論型 `string` を指定し、式 `TimeSpan.Parse(s)` を `f1` の戻り値の型に関連付け、`Y` を `System.TimeSpan` として推論します。最後に、2番目の匿名関数のパラメーター `t` に推論型 `System.TimeSpan` を指定し、式 `t.TotalSeconds` を `f2` の戻り値の型に関連付け、`Z` を `double` として推論します。呼び出しの結果は `double` 型になります。

### 7.5.2.13 メソッド グループの変換での型推論

ジェネリック メソッドの呼び出しと同様に、型推論は、ジェネリック メソッドを含むメソッド グループ `M` が特定のデリゲート型 `D` (§ 6.6) に変換される際にも適用する必要があります。次のようなメソッドがあるとします。

$T_r M<X_1 \dots X_n>(T_1 x_1 \dots T_m x_m)$

このメソッド、およびメソッド グループ `M` がデリゲート型 `D` に代入される場合、型推論の役目は型引数 `S_1 \dots S_n` を見つけ、式

$M<S_1 \dots S_n>$

が `D` と互換になることです (15.1 参照)。

この場合、ジェネリック メソッド呼び出しの型推論アルゴリズムとは異なり、引数の型があるのみで、式はありません。特に、匿名関数がないため、推論を複数のフェーズに分けて行う必要はありません。

代わりに、すべての `X_i` は固定されていないと見なされ、下限の推論は `D` の各引数型 `U_j` から対応する `M` のパラメーター型 `T_j` に行われます。`X_i` のいずれかに範囲が見つからない場合は、型推論は失敗します。それ以外の場合、すべての `X_i` は、型推論の結果である、対応する `S_i` に固定されます。

### 7.5.2.14 式の集合の最適な共通の型の特定

式の集合の共通の型を推論することが必要になる場合があります。特に、暗黙に型が指定された配列の要素型、および本体が `block` である匿名関数の戻り値の型は、次の方法で特定します。

直感的には、式の集合 `E_1 \dots E_m` では、この推論は、メソッド

$T_r M<X>(X x_1 \dots X x_m)$

の呼び出し (`E_i` を引数とする) と同等である必要があります。

より厳密には、推論は固定されていない型変数 `X` で開始されます。出力型推論は、各 `E_i` から `X` に対して行われます。最後に、`X` が固定され、正常に処理された場合は、結果の型 `S` が式の最適な共通の型になります。この `S` が存在しない場合、式の最適な共通の型は存在しません。

### 7.5.3 オーバーロードの解決法

オーバーロードの解決とは、引数リストと関数メンバー候補の集合に基づいて呼び出す最適な関数メンバーを選択するための、バインディング時の機構です。オーバーロードの解決により、C# の次のような異なるコンテキストにおいて呼び出す関数メンバーが選択されます。

- *invocation-expression* で名前を指定されたメソッドの呼び出し (7.6.5.1 を参照)
- *object-creation-expression* で名前を指定されたインスタンス コンストラクターの呼び出し (7.6.10.1 を参照)
- *element-access* を通したインデックス アクセサーの呼び出し (7.6.6 を参照)
- 式で参照されている定義済み演算子またはユーザー定義演算子の呼び出し (7.3.3 および 7.3.4 を参照)

前のセクションで詳しく説明しているように、各コンテキストでは、関数メンバー候補の集合と引数のリストをコンテキスト固有の方法で定義しています。たとえば、メソッド呼び出し候補の集合には、**override** (7.4 を参照) でマークされたメソッドは含まれません。また、派生クラスのメソッドを適用できる場合、基底クラスのメソッドは候補になりません (7.6.5.1 を参照)。

関数メンバー候補と引数リストがいったん識別された後、最適な関数メンバーを選択する方法は、すべてのコンテキストで同じです。

- 適用可能な関数メンバー候補の集合の中から、最適な関数メンバーが特定されます。集合に含まれている関数メンバーが 1 つだけの場合は、それが最適な関数メンバーです。集合に複数の関数メンバーが含まれている場合は、7.5.3.2 の規則を使って関数メンバーを相互に比較し、指定されている引数リストに関して他のすべての関数メンバーよりも適切な関数メンバーが、最適な関数メンバーになります。他のすべての関数メンバーより適切な関数メンバーを 1 つだけに絞り込むことができない場合、関数メンバーの呼び出しあいまいであり、バインディング エラーになります。

以下のセクションでは、"適用可能な関数メンバー" と、"より適切な関数メンバー" という用語の正確な意味を定義します。

#### 7.5.3.1 適用可能な関数メンバー

関数メンバーは、以下のすべてが満たされた場合に、引数リスト A に関する "適切な関数メンバー" であることになります。

- A の各引数が 7.5.1.1 で説明されているように関数メンバー宣言のパラメーターに対応し、最大で 1 つの引数が各パラメーターに対応し、対応する引数がないパラメーターは省略可能なパラメーターである。
- A の各引数について引数のパラメーター受け渡しモード (7.5.1 を参照) が対応するパラメーターのパラメーター受け渡しモード (1.6.6.1 を参照) と一致し、さらに次のいずれかが成立する。
  - 値パラメーターまたはパラメーター配列の場合は、引数式から対応するパラメーターの型への暗黙の変換 (0 を参照) が存在する。
  - **ref** パラメーターまたは **out** パラメーターの場合は、引数式の型が対応するパラメーターの型と一致する。つまり、**ref** パラメーターと **out** パラメーターはどちらも、受け渡す引数のエイリアスです。

パラメーター配列を含む関数メンバーが、上記の規則により適用可能である場合は、"標準形式"で適用可能であると言います。パラメーター配列を含む関数メンバーが標準形式で適用可能でない場合は、代わりに"展開形式"で適用可能である可能性があります。

- 拡張形式は、関数メンバー宣言のパラメーター配列をパラメーター配列の要素型で 0 個以上の値パラメーターに置き換えて、引数リスト A の引数の数がパラメーターの総数と一致するように構成されています。A の引数の数が関数メンバー宣言の固定パラメーターの数よりも少ない場合は、関数メンバーの拡張形式を構成できず、したがって適切ではありません。
- それ以外の場合で、A の各引数について引数のパラメーター受け渡しモードが対応するパラメーターのパラメーター受け渡しモードと一致し、さらに次のどちらかが成立する場合、展開形式は適用可能です。
  - 固定値パラメーターまたは展開によって作成された値パラメーターの場合は、引数の型から対応するパラメーターの型への暗黙の変換 (0 を参照) が存在する。
  - ref パラメーターまたは out パラメーターの場合は、引数の型が対応するパラメーターの型と一致する。

### 7.5.3.2 より適切な関数メンバー

より適切な関数メンバーを決定するためには、元の引数リストと同じ順序で引数式自体のみを含む、最小限の引数リスト A が作成されます。

各関数メンバー候補のパラメーター リストは、次のように作成されます。

- 関数メンバーが展開形式でのみ適用可能であった場合は、展開形式が使用されます。
- 対応する引数のない省略可能なパラメーターは、パラメーター リストから削除されます。
- 引数リストの対応する引数と同じ位置になるように、パラメーターの順序が変更されます。

引数式のセット { E<sub>1</sub>, E<sub>2</sub>, ..., E<sub>N</sub> } の引数リスト A と、パラメーター型 { P<sub>1</sub>, P<sub>2</sub>, ..., P<sub>N</sub> } と { Q<sub>1</sub>, Q<sub>2</sub>, ..., Q<sub>N</sub> } を持つ 2 つの適用可能な関数メンバー M<sub>P</sub> と M<sub>Q</sub> がある場合、次の条件が満たされると、M<sub>P</sub> は M<sub>Q</sub> と比較して、より適切な関数メンバーであると定義されます。

- 各引数について、E<sub>x</sub> から Q<sub>x</sub> への暗黙の型変換が、E<sub>x</sub> から P<sub>x</sub> への暗黙の型変換よりも適切ではない。
- 少なくとも 1 つの引数について、E<sub>x</sub> から P<sub>x</sub> への変換の方が、E<sub>x</sub> から Q<sub>x</sub> への変換よりも適切である。

この評価を行うとき、M<sub>P</sub> または M<sub>Q</sub> が展開形式で適用可能である場合は、P<sub>x</sub> または Q<sub>x</sub> は、パラメーター リストの展開形式におけるパラメーターを示します。

パラメーター型シーケンス { P<sub>1</sub>, P<sub>2</sub>, ..., P<sub>N</sub> } と { Q<sub>1</sub>, Q<sub>2</sub>, ..., Q<sub>N</sub> } が同等である(つまり、各 P<sub>i</sub> に対応する Q<sub>i</sub> への恒等変換がある)場合は、次の決定規則を適用して、より適切な関数メンバーを決定します。

- M<sub>P</sub> が非ジェネリック メソッドで M<sub>Q</sub> がジェネリック メソッドの場合、M<sub>P</sub> は M<sub>Q</sub> よりも適切です。
- それ以外の場合で、M<sub>P</sub> が標準形式で適用可能であり、M<sub>Q</sub> に params 配列があり、MQ が展開形式でのみ適用可能である場合、M<sub>P</sub> は M<sub>Q</sub> よりも適切です。

- それ以外の場合で、 $M_P$  の宣言されたパラメーターの数が  $M_Q$  の宣言されたパラメーターの数よりも多い場合、 $M_P$  は  $M_Q$  よりも適切です。この状況は、両方のメソッドに `params` 配列があり、拡張形式でのみ使用できる場合に発生することがあります。
- それ以外の場合で、 $M_P$  のすべてのパラメーターには対応する引数が存在するのに対して、 $M_Q$  では 1 つ以上の省略可能なパラメーターを既定の引数に置き換える必要がある場合、 $M_P$  は  $M_Q$  より適切です。
- それ以外の場合で、 $M_P$  が  $M_Q$  よりも限定的なパラメーター型を持つ場合、 $M_P$  は  $M_Q$  よりも適切です。 $\{R_1, R_2, \dots, R_N\}$  と  $\{S_1, S_2, \dots, S_N\}$  が  $M_P$  と  $M_Q$  のインスタンス化されず拡張されていないパラメーター型であるとします。各パラメーターに関して、 $R_x$  が  $S_x$  よりも限定的でなく、少なくとも 1 つのパラメーターに対して  $R_x$  が  $S_x$  よりも限定的である場合、 $M_P'$  のパラメーター型は  $M_Q$  のパラメーター型よりも限定的です。
  - 型パラメーターは、型でないパラメーターより限定的ではありません。
  - 再帰的に、構築された型が別の構築された型（型引数の数は同じ）よりも限定的とされるのは、少なくとも 1 つの型引数がもう一方の対応する型引数よりも限定的であり、もう一方の対応する型引数よりも限定的でない型引数が存在しない場合です。
  - 配列型が別の配列型（次元数は同じ）よりも限定的とされるのは、最初の配列の要素型がもう一方の要素型よりも限定的である場合です。
- それ以外の場合、1 つのメンバーが非リフト演算子で、もう 1 つがリフト演算子である場合は、非リフト演算子の方が適切です。
- 上記以外の場合は、どちらの関数メンバーも "より適切" ではありません。

### 7.5.3.3 式からの "より適切な変換"

式  $E$  を型  $T_1$  に変換する暗黙の型変換  $C_1$ 、および式  $E$  を型  $T_2$  に変換する暗黙の型変換  $C_2$  があるとすると、次のいずれかの条件が満たされる場合、 $C_2$  よりも  $C_1$  の方が "適切な変換" となります。

- $E$  に型  $S$  があり、 $S$  から  $T_1$  への恒等変換は存在するが、 $S$  から  $T_2$  には存在しない
- $E$  が匿名関数ではなく、 $T_1$  の方が  $T_2$  より適切な変換対象である（7.5.3.5 を参照）
- $E$  が匿名関数、 $T_1$  がデリゲート型  $D_1$  または式ツリー型 `Expression<D1>、 $T_2$  がデリゲート型  $D_2$  または式ツリー型 Expression<D2> で、次のいずれかである。
 
  - $D_1$  の方が  $D_2$  より適切な変換対象である
  - $D_1$  と  $D_2$  のパラメーターリストが同じで、次のいずれかである
    - $D_1$  の戻り値の型が  $Y_1$ 、 $D_2$  の戻り値の型が  $Y_2$  で、推論された戻り値の型  $X$  がそのパラメーターリストのコンテキストにおいて  $E$  に対して存在し（7.5.2.12 を参照）、 $X$  から  $Y_1$  への変換の方が、 $X$  から  $Y_2$  への変換より適切である
    - $E$  が非同期で、 $D_1$  の戻り値の型が Task<Y1>、 $D_2$  の戻り値の型が Task<Y2> で、推論された戻り値の型 Task<X> がそのパラメーターリストのコンテキストにおいて  $E$  に対して存在し（7.5.2.12 を参照）、 $X$  から  $Y_1$  への変換の方が、 $X$  から  $Y_2$  への変換より適切である
    - $D_1$  の戻り値の型が  $Y$  であり、 $D_2$  が void を返す`

### 7.5.3.4 型からの "より適切な変換"

型  $S$  を型  $T_1$  に変換する変換  $C_1$ 、および型  $S$  を型  $T_2$  に変換する変換  $C_2$  があるとすると、次のいずれかの条件が満たされる場合、 $C_2$  よりも  $C_1$  の方が "**より適切な変換**" となります。

- $S$  から  $T_1$  への恒等変換は存在するが、 $S$  から  $T_2$  への恒等変換は存在しない。
- $T_1$  の方が  $T_2$  より適切な変換対象である (7.5.3.5 を参照)

### 7.5.3.5 より適切な変換対象

異なる 2 つの型  $T_1$  と  $T_2$  がある場合、次の条件が 1 つでも満たされると、 $T_1$  の方が  $T_2$  より適切な変換対象です。

- $T_1$  から  $T_2$  への暗黙の型変換が存在し、 $T_2$  から  $T_1$  への暗黙の型変換が存在しない
- $T_1$  が符号付き整数型であり、 $T_2$  が符号なし整数型である具体的には、以下の警告があります。
  - $T_1$  が `sbyte` で  $T_2$  が `byte`、`ushort`、`uint`、または `ulong` です。
  - $T_1$  が `short` で  $T_2$  が `ushort`、`uint`、または `ulong` です。
  - $T_1$  が `int` で  $T_2$  は `uint`、または `ulong` です。
  - $T_1$  が `long` で、 $T_2$  が `ulong` です。

### 7.5.3.6 ジェネリック クラスのオーバーロード

シグニチャは一意で宣言する必要がありますが、型引数の置換によって同じシグニチャが生成される可能性があります。同じシグニチャが生成された場合は、上記のオーバーロード解決の決定規則によって最も限定的なメンバーが選択されます。

次の例は、この規則に従って有効になるオーバーロードと無効になるオーバーロードを示します。

```
interface I1<T> { ... }
interface I2<T> { ... }
class G1<U>
{
    int F1(U u);           // Overload resolution for G<int>.F1
    int F1(int i);         // will pick non-generic
    void F2(I1<U> a);     // valid overload
    void F2(I2<U> a);
}
class G2<U,V>
{
    void F3(U u, V v);    // valid, but overload resolution for
    void F3(V v, U u);    // G2<int,int>.F3 will fail
    void F4(U u, I1<V> v); // valid, but overload resolution for
    void F4(I1<V> v, U u); // G2<I1<int>,int>.F4 will fail
    void F5(U u1, I1<V> v2); // valid overload
    void F5(V v1, U u2);
    void F6(ref U u);      // valid overload
    void F6(out V v);
}
```

#### 7.5.4 動的なオーバーロードの解決のコンパイル時チェック

動的にバインドされる操作のオーバーロード解決は実行時に行われますが、場合によっては、オーバーロードが選択される関数メンバーのリストをコンパイル時に確認することができます。

- 型、または静的な型が `dynamic` ではない値でのメソッド呼び出し (7.6.5.1 を参照) の場合、メソッドグループ内のアクセス可能なメソッドのセットをコンパイル時に確認できます。
- オブジェクト作成式 (7.6.10.1 を参照) の場合、型に含まれるアクセス可能なコンストラクターのセットをコンパイル時に確認できます。
- インデクサーアクセス (7.6.6.2 を参照) の場合、レシーバーに含まれるアクセス可能なインデクサーのセットをコンパイル時に確認できます。

これらのケースでは、関数メンバーの既知のセット内の各メンバーに対して限定的なコンパイル時チェックが実行され、実行時に特定のメンバーが呼び出されないことを把握できるかどうかが確認されます。それぞれの関数メンバー `F` に対し、変更されたパラメーターと引数リストが構築されます。

- 最初に、`F` がジェネリック メソッドで、型引数が提供されている場合、これらはパラメーター リスト内の型パラメーターと置き換えられます。ただし、型引数が提供されていない場合は、置き換えは行われません。
- 次に、開いた型を持つ(つまり型パラメーターを持つ)パラメーター (4.4.2 を参照) がすべて、対応するパラメーターと共に削除されます。

`F` がこのチェックに合格するためには、次のすべての条件が満たさる必要があります。

- `F` の変更されたパラメーター リストが、7.5.3.1 の観点から、変更された引数リストに適用可能である。
- 変更されたパラメーター リストのすべての構築された型がその制約 (4.4.4 を参照) を満たす。
- `F` の型パラメーターが上の手順で置き換えられている場合、その制約が満たされている。
- `F` が静的メソッドの場合、メソッドグループは、レシーバーが変数または値であることがコンパイル時に確認される *member-access* に基づいていない。
- `F` がインスタンス メソッドの場合、メソッドグループは、レシーバーが型であることがコンパイル時に確認される *member-access* に基づいていない。

このテストに合格する候補が 1 つも存在しない場合は、コンパイル エラーが発生します。

#### 7.5.5 関数メンバーの呼び出し

ここでは、特定の関数メンバーを呼び出すために実行時に行われる処理について説明します。バインディング時の処理において、関数メンバー候補の集合にオーバーロードの解決を適用するなどして、呼び出す特定のメンバーが既に決定されているとします。

呼び出しの処理を説明するために、関数メンバーを次の 2 つのカテゴリに分けます。

- 静的関数メンバー。インスタンス コンストラクター、静的メソッド、静的プロパティ アクセサー、およびユーザー定義演算子です。静的関数メンバーは、常に非仮想です。
- インスタンス関数メンバー。インスタンス メソッド、インスタンス プロパティ アクセサー、およびインデックス アクセサーです。インスタンス関数メンバーは非仮想または仮想で、必ず特定

のインスタンスで呼び出されます。インスタンスは、インスタンス式によって計算され、関数メンバーの中では `this` (7.6.7 を参照) としてアクセス可能になります。

関数メンバー呼び出しの実行時の処理は、以下の手順で構成されます。`M` は関数メンバーで、`M` がインスタンスマネバの場合、`E` はインスタンス式です。

- `M` が静的関数メンバーの場合。
  - 引数リストは、7.5.1 で説明する方法で評価されます。
  - `M` が呼び出されます。
- `M` が、*value-type* で宣言されているインスタンスマネバの場合。
  - `E` が評価されます。この評価で例外が発生すると、これ以降の手順は実行されません。
  - `E` が変数に分類されない場合は、`E` の型でテンポラリローカル変数が作成されて、`E` の値がこの変数に代入されます。その後、`E` は、その一時ローカル変数に対する参照として分類し直されます。テンポラリ変数は、`M` の中では `this` としてアクセスできますが、他の方法ではアクセスできません。したがって、`E` が本当の変数のときにだけ、呼び出し側は、`M` が `this` に対して行う変更を観察できます。
  - 引数リストは、7.5.1 で説明する方法で評価されます。
  - `M` が呼び出されます。`E` で参照されている変数が、`this` で参照される変数になります。
- `M` が、*reference-type* で宣言されているインスタンスマネバの場合。
  - `E` が評価されます。この評価で例外が発生すると、これ以降の手順は実行されません。
  - 引数リストは、7.5.1 で説明する方法で評価されます。
  - `E` の型が *value-type* の場合は、`E` を `object` 型に変換するボックス化変換 (4.3.1 を参照) が実行されて、以下の手順では、`E` は `object` 型であると見なされます。この場合、`M` は単に `System.Object` のメンバになります。
  - `E` の値が有効であることをチェックします。`E` の値が `null` の場合、`System.NullReferenceException` がスローされ、それ以上の手順は実行されません。
  - 呼び出す関数メンバの実装を決定します。
    - `E` のバインディング時の型がインターフェイス型の場合、呼び出す関数メンバは、`E` が参照するインスタンスの実行時の型が提供する `M` の実装です。この関数メンバを決定するには、`E` が参照するインスタンスの実行時の型が提供する `M` の実装を決定するインターフェイスマップ規則 (13.4.4 を参照) を適用します。
    - それ以外の場合、`M` が仮想関数メンバのときに、呼び出す関数メンバは、`E` が参照するインスタンスの実行時の型が提供する `M` の実装です。この関数メンバを決定するには、`E` が参照するインスタンスの実行時の型に関する `M` の最派生実装 (10.6.3 を参照) を決定するための規則を適用します。
    - それ以外の場合、`M` は非仮想関数メンバであり、呼び出す関数メンバは `M` 自体です。
  - 上の手順で決定された関数メンバの実装が呼び出されます。`E` で参照されているオブジェクトが、`this` で参照されるオブジェクトになります。

### 7.5.5.1 ボックス化されたインスタンスでの呼び出し

*value-type* で実装された関数メンバーは、以下の状況では、その *value-type* のボックス化されたインスタンスを通じて呼び出すことができます。

- 関数メンバーが、`object` 型から継承されたメソッドの `override` で、`object` 型のインスタンス式を通じて呼び出されるとき。
- 関数メンバーが、インターフェイス関数メンバーの実装で、*interface-type* のインスタンス式を通じて呼び出されるとき。
- 関数メンバーが、デリゲートを通じて呼び出されるとき。

これらの状況では、ボックス化されたインスタンスは *value-type* の変数を含むものと見なされ、この変数が、関数メンバー呼び出しの中の `this` によって参照される変数になります。特にこのことは、ボックス化されたインスタンスで呼び出された関数メンバーは、ボックス化されたインスタンスに含まれる値を変更できることを意味します。

## 7.6 基本式

基本式では、式の最も簡単な形式が使われています。

```

primary-expression:
  primary-no-array-creation-expression
  array-creation-expression

primary-no-array-creation-expression:
  literal
  simple-name
  parenthesized-expression
  member-access
  invocation-expression
  element-access
  this-access
  base-access
  post-increment-expression
  post-decrement-expression
  object-creation-expression
  delegate-creation-expression
  anonymous-object-creation-expression
  typeof-expression
  checked-expression
  unchecked-expression
  default-value-expression
  anonymous-method-expression

```

基本式は、*array-creation-expressions* と *primary-no-array-creation-expression* に分けられます。*array-creation-expression* をこのように扱うことで、他の単純な式の形式と共に記述する場合と比べて、次のような場合にコードが混乱する可能性のある文法表現を防ぐことができます。

```
object o = new int[3][1];
```

一緒に記述すると、上のコードは次のように解釈される可能性があります。

```
object o = (new int[3])[1];
```

## 7.6.1 リテラル

*literal* (2.4.4 を参照) で構成される *primary-expression* は、値として分類されます。

## 7.6.2 簡易名

*simple-name* は、次のように、識別子とそれに続くオプションの型引数リストで構成されます。

*simple-name*:  
  *identifier* *type-argument-list*<sub>opt</sub>

*simple-name* が形式 *I* または形式 *I<A<sub>1</sub>, ..., A<sub>K</sub>>* です。ここで、*I* は単一の識別子、*<A<sub>1</sub>, ..., A<sub>K</sub>>* はオプションの *type-argument-list* です。*type-argument-list* が指定されていない場合、*K* はゼロと見なされます。*simple-name* は、次のように評価および分類されます。

- *K* がゼロで、*simple-name* が *block* の中で使われていて、*block* (または外側の複数の *block*) のローカル変数宣言空間 (3.3 を参照) に名前 *I* を持つローカル変数、パラメーター、または定数が含まれている場合、*simple-name* はそのローカル変数、パラメーター、または定数を示し、変数または値に分類されます。
- *K* がゼロで、*simple-name* がジェネリック メソッド宣言の本体内に存在する場合で、その宣言に、名前 *I* を持つ型パラメーターが含まれる場合、*simple-name* はその型パラメーターを参照します。
- それ以外の場合、すぐ外側の型宣言のインスタンス型を開始点として、外側のクラス宣言または構造体宣言 (存在する場合) を順番にたどりながら、それぞれのインスタンス型 *T* (10.3.1 を参照) について次の評価が行われます。
  - *K* がゼロで、*T* の宣言に名前 *I* を持つ型パラメーターが含まれる場合は、*simple-name* はその型パラメーターを参照します。
  - それ以外の場合、*K* 個の型引数を持つ *T* について *I* でメンバー検索 (7.4 を参照) が実行され、一致が見つかると次のように処理されます。
    - *T* がすぐ外側のクラス型または構造体型のインスタンス型で、検索において 1 つ以上のメソッドが識別された場合、結果は、関連付けられたインスタンス式を持つ *this* のメソッド グループです。型引数リストが指定された場合は、ジェネリック メソッドの呼び出しに使用されます (7.6.5.1 を参照)。
    - それ以外の場合で、*T* がすぐ外側のクラス型または構造体型のインスタンス型であり、検索によってインスタンス メンバーが 1 つ識別され、参照の出現位置がインスタンス コンストラクター、インスタンス メソッド、またはインスタンス アクセサーの *block* 内である場合、結果 (7.6.4 を参照) は *this.I* の形式のメンバー アクセスと同じになります。これは、*K* がゼロの場合にのみ発生します。
    - それ以外の場合、結果 (7.6.4 を参照) は *T.I* または *T.I<A<sub>1</sub>, ..., A<sub>K</sub>>*。この場合、*simple-name* がインスタンス メンバーを参照するとバインディング エラーになります。
- それ以外の場合、*simple-name* が存在する名前空間から始めて、外側の名前空間 (存在する場合) を順番にたどり、グローバル名前空間に達するまでの各名前空間 *N* について、エンティティが見つかるまで次の手順が評価されます。
  - *K* がゼロで、*I* が *N* における名前空間の名前の場合は、次の規則に従います。
    - *simple-name* の存在する場所が *N* の名前空間宣言で囲まれており、その名前空間宣言に含まれる *extern-alias-directive* または *using-alias-directive* によって名前 *I* が名前空間または型

と関連付けられている場合、この *simple-name* はあいまいであり、コンパイル エラーになります。

- それ以外の場合、*simple-name* は *N* における *I* という名前の名前空間を参照します。
- それ以外の場合で、名前が *I* で *K* 個の型パラメーターを持つアクセス可能な型が *N* に含まれている場合は、次の規則に従います。
  - *K* がゼロで、*simple-name* の存在する場所が *N* の名前空間宣言で囲まれており、その名前空間宣言に含まれる *extern-alias-directive* または *using-alias-directive* によって名前 *I* が名前空間または型と関連付けられている場合、この *simple-name* はあいまいであり、コンパイル エラーになります。
  - それ以外の場合、*namespace-or-type-name* は指定された型引数によって構築された型を参照します。
- それ以外の場合で、*simple-name* の存在する場所が *N* の名前空間宣言に囲まれている場合は、次の規則に従います。
  - *K* がゼロで、名前空間の宣言に含まれる *extern-alias-directive* または *using-alias-directive* によって名前 *I* とインポートされた名前空間または型が関連付けられている場合、*simple-name* はその名前空間または型を参照します。
  - それ以外の場合で、名前空間の宣言の *using-namespace-directive* によってインポートされる名前空間に、名前が *I* で *K* 個の型パラメーターを持つ型が 1 つだけ含まれている場合、*simple-name* は指定された型引数で構築されたその型を参照します。
  - それ以外の場合で、名前空間の宣言の *using-namespace-directive* によってインポートされる名前空間に、名前が *I* で *K* 個の型パラメーターを持つ型が 2 つ以上含まれている場合は、*simple-name* はあいまいであり、エラーになります。

この手順は、*namespace-or-type-name* の処理の対応する手順 (3.8 を参照) と同じです。

- 以上のいずれにも該当しない場合、*simple-name* は未定義になり、コンパイル エラーが発生します。

#### 7.6.2.1 複数のブロック内での意味の一致

式または宣言子で完全な *simple-name* (型引数リストなし) として指定される各識別子について、その出現箇所を囲むすぐ外側のローカル変数宣言空間 (§ 3.3 を参照) の内部では、式または宣言子で完全な *simple-name* として指定される同じ識別子が他にも出現する場合、すべてが同じエンティティを参照している必要があります。この規則によって、特定のブロック、switch ブロック、for ステートメント、foreach ステートメント、using ステートメント、または匿名関数の内部で、名前の意味が常に同一になります。

次の例を参照してください。

```
class Test
{
    double x;
```

```
void F(bool b) {
    x = 1.0;
    if (b) {
        int x;
        x = 1;
    }
}
```

この例では、外側のブロックの範囲に `if` ステートメントで入れ子にされたブロックが含まれています。外側のブロックから見て、`x` は異なる複数のエンティティを参照しているため、コンパイルエラーが発生します。これとは対照的な例を示します。

```
class Test
{
    double x;
    void F(bool b) {
        if (b) {
            x = 1.0;
        }
        else {
            int x;
            x = 1;
        }
    }
}
```

この場合は、外側のブロックで `x` という名前が使われていないため、エラーにはなりません。

意味の一致の規則は、簡易名にのみ適用されることに注意してください。同じ識別子が、簡易名として 1 つの意味を持ち、メンバーアクセス (7.6.4 を参照) の右オペランドとして別の意味を持っていても、まったく問題ありません。次に例を示します。

```
struct Point
{
    int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

上の例は、インスタンス コンストラクターのパラメータ名としてフィールドの名前を使用する、よく使われるパターンを示しています。この例では、簡易名 `x` と `y` はパラメーターを示していますが、これにより、メンバーアクセス式 `this.x` および `this.y` によるフィールドへのアクセスが妨げられることはできません。

### 7.6.3 かっこで囲まれた式

*parenthesized-expression* は、かっこで囲まれた *expression* によって構成されます。

*parenthesized-expression*:  
 ( *expression* )

*parenthesized-expression* の評価は、かっこ内の *expression* に対して行われます。かっこ内の *expression* が名前空間または型を表している場合は、コンパイルエラーが発生します。それ以外の場合、*parenthesized-expression* の結果は、かっこ内の *expression* を評価した結果になります。

## 7.6.4 メンバー アクセス

*member-access* は、*primary-expression*、*predefined-type*、*qualified-alias-member* のいずれかの後ろに、`.` トークン、*identifier*、オプションの *type-argument-list* という順番で構成されます。

*member-access*:

```
primary-expression . identifier type-argument-listopt
predefined-type . identifier type-argument-listopt
qualified-alias-member . identifier type-argument-listopt
```

*predefined-type*: one of

bool	byte	char	decimal	double	float	int	long
object	sbyte	short	string	uint	ulong	ushort	

*qualified-alias-member* の生成については、9.7 で定義しています。

*member-access* が形式 `E.I` または形式 `E.I<A1, ..., AK>` です。ここで、`E` は基本式、`I` は单一の識別子、`<A1 ..., , AK>` はオプションの *type-argument-list* です。*type-argument-list* が指定されていない場合、`K` はゼロと見なされます。

`dynamic` 型の *primary-expression* を含む *member-access* は、動的にバインドされます(7.2.2 を参照)。この場合、コンパイラはメンバー アクセスを `dynamic` 型のプロパティ アクセスに分類します。その後、*member-access* の意味を判断する以下の規則が実行時に適用されます。これには、*primary-expression* のコンパイル時の型ではなく実行時の型が使用されます。この実行時の分類がメソッド グループになる場合、メンバー アクセスは *invocation-expression* の *primary-expression* である必要があります。

*member-access* は、次のように評価および分類されます。

- `K` がゼロで、`E` が名前空間であり、`E` が `I` という名前の入れ子になった名前空間を含む場合、結果はその名前空間になります。
- それ以外の場合で、`E` が名前空間で、`E` に名前が `I` で `K` 個の型パラメーターを持つアクセス可能な型が含まれる場合、結果は指定された型引数によって構築された型になります。
- `E` が型として分類される *predefined-type* または *primary-expression* であり、かつ `E` が型パラメーターでない場合は、`K` 個の型パラメーターを持つ `E` について、`I` でメンバー検索(7.4 を参照)が実行されます。一致が見つかると、`E.I` が次のように評価および分類されます。
  - `I` が型を示している場合、結果は指定した型引数で構築された型になります。
  - `I` が 1 つ以上のメソッドと一致する場合、結果は対応するインスタンス式を持たないメソッド グループとなります。型引数リストが指定された場合は、ジェネリック メソッドの呼び出しに使用されます(7.6.5.1 を参照)。
  - `I` が `static` プロパティを示している場合は、結果はインスタンス式が関連付けられていないプロパティ アクセスです。
  - `I` が `static` フィールドを示している場合は、次のとおりです。
    - フィールドが `readonly` で、フィールドが宣言されているクラスまたは構造体の静的コンストラクターの外部で参照が行われている場合、結果は値、つまり `E` の中の静的フィールド `I` の値になります。
    - それ以外の場合、結果は変数、つまり `E` の中の静的フィールド `I` になります。

- **I** が `static` イベントを示している場合は、次のとおりです。
  - イベントが宣言されているクラスまたは構造体の中で参照が行われていて、イベントが *event-accessor-declarations* (10.8 を参照) を使わずに宣言されている場合、**E.I** は **I** が静的フィールドである場合とまったく同様に処理されます。
  - それ以外の場合は、結果はインスタンス式が関連付けられていないイベントアクセスです。
- **I** が定数を示している場合は、結果は値、つまりその定数の値です。
- **I** が列挙体のメンバーを示している場合は、結果は値、つまりその列挙体メンバーの値です。
- それ以外の場合、**E.I** は無効なメンバー参照であり、コンパイル エラーが発生します。
- **E** がプロパティ アクセス、インデクサー アクセス、変数、または値で、その型が **T** である場合は、**K** 個の型引数を持つ **T** について、**I** でメンバー検索 (7.4 を参照) が実行されます。一致が見つかると、**E.I** は次のように評価および分類されます。
  - **E** がプロパティ アクセスまたはインデクサー アクセスである場合は、そのプロパティ アクセスまたはインデクサー アクセスの値が取得され (7.1.1 を参照)、**E** は値として再分類されます。
  - **I** が 1 つ以上のメソッドと一致する場合、結果は **E** の対応するインスタンス式を持つメソッドグループとなります。型引数リストが指定された場合は、ジェネリック メソッドの呼び出しに使用されます (7.6.5.1 を参照)。
  - **I** がインスタンス プロパティを示している場合、結果は **E** のインスタンス式が関連付けられているプロパティ アクセスになります。
  - **T** が *class-type* で、**I** がその *class-type* のインスタンス フィールドを示している場合は、次のとおりです。
    - **E** の値が `null` の場合、`System.NullReferenceException` がスローされます。
    - フィールドが `readonly` で、フィールドが宣言されているクラスのインスタンス コンストラクターの外部で参照が行われている場合、結果は値、つまり **E** によって参照されているオブジェクト内のフィールド **I** の値になります。
    - それ以外の場合、結果は変数、つまり **E** によって参照されているオブジェクト内のフィールド **I** になります。
  - **T** が *struct-type* で、**I** がその *struct-type* のインスタンス フィールドを示している場合は、次のとおりです。
    - **E** が値の場合、またはフィールドが `readonly` で、フィールドが宣言されている構造体のインスタンス コンストラクターの外部で参照が行われている場合、結果は値、つまり **E** によって指定されている構造体インスタンス内のフィールド **I** の値になります。
    - それ以外の場合、結果は変数、つまり **E** によって指定されている構造体インスタンス内のフィールド **I** になります。
  - **I** がインスタンス イベントを示している場合は、次のとおりです。
    - イベントが宣言されているクラスまたは構造体の中で参照が行われ、イベントが *event-accessor-declarations* (10.8 を参照) を使わずに宣言されていて、その参照が `+=` 演算子また

は $=$ 演算子の左側で行われない場合、 $E.I$ は $I$ がインスタンスフィールドである場合とまったく同様に処理されます。

- それ以外の場合、結果は $E$ のインスタンス式が関連付けられているイベントアクセスになります。
- それ以外の場合は、 $E.I$ を拡張メソッド呼び出し(7.6.5.2を参照)として処理する試みが行われます。この処理に失敗した場合、 $E.I$ は無効なメンバー参照であり、バインディングエラーが発生します。

#### 7.6.4.1 同一の簡易名と型名

$E.I$ の形式のメンバーアクセスにおいて、 $E$ が単一の識別子であり、*simple-name*(7.6.2を参照)としての $E$ の意味が*type-name*(3.8を参照)としての $E$ と同じ型を持つ定数、フィールド、プロパティ、ローカル変数、またはパラメーターである場合は、 $E$ に対して可能性のある両方の意味が認められます。いずれの場合も $I$ は必然的に型 $E$ のメンバーである必要があるため、 $E.I$ に対して可能性のある2つの意味があいまいになることはありません。つまり、他の場合はコンパイルエラーが発生する状況であっても、 $E$ の静的メンバーおよび入れ子になった型に対するアクセスだけは認められています。次に例を示します。

```
struct Color
{
    public static readonly Color White = new Color(...);
    public static readonly Color Black = new Color(...);
    public Color Complement() {...}
}

class A
{
    public Color color; // Field color of type Color
    void F()
    {
        Color = color.Black; // References Color.Black static member
        Color = color.Complement(); // Invokes Complement() on Color field
    }
    static void G()
    {
        Color c = color.White; // References Color.White static member
    }
}
```

Aクラスの中で、Color識別子がColor型を参照している場合は下線を付けて示し、Colorフィールドを参照している場合は下線を付けないで示しています。

#### 7.6.4.2 文法のあいまいさ

*simple-name*(7.6.2を参照)および*member-access*(7.6.4を参照)が作成されたことにより、式の文法があいまいになる場合があります。次に例を示します。

```
F(G<A,B>(7));
```

これは、2つの引数 $G < A$ と $B >$ (7)を持つ $F$ の呼び出しと解釈できます。また、1つの引数を持つ $F$ の呼び出しであり、その引数が2つの型引数と1つの通常の引数を持つジェネリックメソッド $G$ の呼び出しであると解釈することもできます。

トークンのシーケンスが、*type-argument-list*(7.6.2を参照)で終わる*simple-name*(7.6.4を参照)、*member-access*(4.4.1を参照)、または*pointer-member-access*(18.5.2を参照)として(コンテキスト内で)

解析できる場合は、閉じる > トークンの直後にあるトークンが調べられます。そのトークンが次のいずれかである場合、

( ) ] } : ; , . ? == != | ^

*type-argument-list* は *simple-name*、*member-access*、または *pointer-member-access* の一部と見なされ、トークンのシーケンスに対してそれ以外に考えられる解析は破棄されます。それ以外の場合は、トークンのシーケンスに対して他に考えられる解析がない場合でも、*type-argument-list* は *simple-name*、*member-access*、または *pointer-member-access* の一部とは見なされません。これらの規則は、*namespace-or-type-name* の *type-argument-list* の解析には適用されません (3.8 を参照)。次のようなステートメントがあります。

F(G<A,B>(7));

これは、先ほどの規則に従って、1つの引数を持つ F の呼び出しであり、その引数は、2つの型引数と 1 つの通常の引数を持つジェネリック メソッド G の呼び出しであると解釈されます。次のようなステートメントがあります。

F(G < A, B > 7);  
F(G < A, B >> 7);

これは、どちらも 2 つの引数を持つ F の呼び出しと解釈されます。次のようなステートメントがあります。

x = F < A > +y;

このステートメントは、*type-argument-list* 付きの *simple-name* の後ろに二項プラス演算子が付いたものではなく、x = (F < A) > (+y) と記述された場合と同様に、小なり演算子、大なり演算子、および単項プラス演算子と解釈されます。次のようなステートメントがあります。

x = y is C<T> + z;

ここでは、トークン C<T> は *type-argument-list* 付きの *namespace-or-type-name* と解釈されます。

## 7.6.5 呼び出し式

*invocation-expression* は、メソッドを呼び出すために使われます。

*invocation-expression*:  
    *primary-expression* ( *argument-list<sub>opt</sub>* )

次のいずれかに該当する場合、*invocation-expression* は動的にバインドされます (7.2.2 を参照)。

- *primary-expression* のコンパイル時の型が **dynamic** である。
- 省略可能な *argument-list* の少なくとも 1 つの引数のコンパイル時の型が **dynamic** であり、*primary-expression* がデリゲート型ではない。

この場合、コンパイラは *invocation-expression* を **dynamic** 型の値に分類します。*invocation-expression* の意味を特定するための次の規則が、*primary-expression* のコンパイル時の型ではなく実行時の型と、コンパイル時の型が **dynamic** である引数を使用して、実行時に適用されます。*primary-expression* のコンパイル時の型が **dynamic** ではない場合、メソッド呼び出しには 7.5.4 で説明されているコンパイル時の限られたチェックが行われます。

*invocation-expression* の *primary-expression* には、メソッド グループまたは *delegate-type* の値を指定する必要があります。*primary-expression* がメソッド グループの場合、*invocation-expression* はメソッド呼び出し (7.6.5.1 を参照) です。*primary-expression* が *delegate-type* の値の場合、*invocation-expression*

はデリゲート呼び出し (7.6.5.3 を参照) です。*primary-expression* がメソッドグループでも *delegate-type* の値でもない場合は、バインディング エラーになります。

省略可能な *argument-list* (7.5.1 を参照) では、メソッドのパラメーターに対する値または変数参照を指定します。

*invocation-expression* の評価結果は、次のように分類されます。

- *invocation-expression* が `void` を返すメソッドまたはデリゲートを呼び出す場合、結果は "なし" です。"なし" に分類される式は、*statement-expression* (8.6 を参照) のコンテキストまたは *lambda-expression* (7.15 を参照) の本体においてのみ有効です。それ以外の場合は、バインディング エラーが発生します。
- それ以外の場合、結果は、メソッドまたはデリゲートで返される型の値です。

#### 7.6.5.1 メソッドの呼び出し

メソッドの呼び出しの場合、*invocation-expression* の *primary-expression* にはメソッドグループを指定する必要があります。メソッドグループは、呼び出す 1 つのメソッド、または呼び出す特定のメソッドを選択する基になるオーバーロードされたメソッドの集合を示します。後者の場合、呼び出す特定のメソッドの決定は、*argument-list* の引数の型で指定されるコンテキストに基づいて行われます。

$M(A)$  の形式 ( $M$  はメソッドグループで多くの場合 *type-argument-list* を含み、 $A$  は省略可能な *argument-list*) によるメソッド呼び出しに対するバインディング時の処理は、次の手順で行われます。

- メソッド呼び出しに対する候補のメソッドの集合を作成します。メソッドグループ  $M$  に関連付けられた各メソッド  $F$  は、次のように処理されます。
  - $F$  が非ジェネリックの場合、 $F$  は次の場合に候補になります。
    - $M$  に型引数リストがない。かつ、
    - $F$  が  $A$  に関して適用可能である (7.5.3.1 を参照)。
  - $F$  がジェネリックで  $M$  が型引数リストを持たない場合、 $F$  は次の場合に候補になります。
    - 型推論 (7.5.2 を参照) が成功し、呼び出しの型引数のリストが推論される。かつ、
    - 推論された型引数が対応するメソッドの型引数に置き換えられ、 $F$  のパラメーター リストのすべての構築された型がその制約 (4.4.4 を参照) を満たし、 $F$  のパラメーター リストが  $A$  に関して適用可能である (7.5.3.1 を参照)。
  - $F$  がジェネリックで  $M$  が型引数リストを含む場合、 $F$  は次の場合に候補になります。
    - $F$  のメソッド型パラメーターの数が、型引数リストに指定された数と同じである。かつ、
    - 対応するメソッド型パラメーターが型引数に置き換えられ、 $F$  のパラメーター リストのすべての構築された型が制約 (4.4.4 を参照) を満たし、 $F$  のパラメーター リストが  $A$  に関して適用可能である (7.5.3.1 を参照)。
- 候補メソッドの集合を、最派生型のメソッドのみを含むように縮小します。具体的には、集合の各メソッド  $C.F$  ( $C$  はメソッド  $F$  が宣言されている型) について、 $C$  の基本型で宣言されているすべてのメソッドを集合から削除します。さらに、 $C$  が `object` 以外のクラス型である場合は、インターフェイス型に宣言されたすべてのメソッドが集合から削除されます。この後者の規則は、

メソッドグループが、**object** 以外の実質的な基底クラスと空でない有効なインターフェイスセットを持つ型パラメーターに対してメンバー検索を実行した結果である場合にのみ有効です。

- 結果の候補メソッドの集合が空の場合は、次の手順による以降の処理は破棄され、代わりに呼び出しを拡張メソッド呼び出し (7.6.5.2 を参照) として処理することが試みられます。この処理が失敗した場合、適用可能なメソッドは存在せず、バインディング エラーが発生します。
- 一連の候補メソッドのうち、最適なメソッドはオーバーロードの解決規則 (7.5.3 を参照) を使って特定されます。単一の最適なメソッドが識別されなかった場合、メソッドの呼び出しはあいまいであり、バインディング エラーが発生します。オーバーロード解決を実行するとき、ジェネリック メソッドのパラメーターは、対応するメソッド型パラメーターの型引数 (指定または推論による) を代入した後に考慮されます。
- 選択した最適なメソッドの最終的な検証は、次のように行われます。
  - メソッドをメソッド グループのコンテキストで検証します。最適なメソッドが静的メソッドの場合、メソッドグループは、*simple-name* から、または型を通じて *member-access* から得られたものである必要があります。最適なメソッドがインスタンス メソッドの場合、メソッド グループは、*simple-name* から、変数または値を通じて *member-access* から、あるいは *base-access* から得られたものである必要があります。いずれの条件も満たされない場合は、バインディング エラーが発生します。
  - 最適なメソッドがジェネリック メソッドの場合は、型引数 (指定または推論による) を、ジェネリック メソッドに宣言されている制約 (4.4.4 を参照) に照らしてチェックします。型引数の対応する制約を満たさない型引数がある場合は、バインディング エラーになります。

上記の手順によってバインディング時にメソッドが選択されて検証されると、実行時の実際の呼び出しは、7.5.4 で説明した関数メンバー呼び出しの規則に従って処理されます。

上で説明した解決規則の結果をまとめると、次のようにになります。メソッド呼び出しで呼び出される特定のメソッドを探し出すには、メソッド呼び出しで示されている型から始めて、適用可能で、アクセス可能で、オーバーライドではないメソッドの宣言が少なくとも 1 つ見つかるまで、継承チェーンを上にたどります。次に、その型で宣言されている適用可能で、アクセス可能で、オーバーライドではないメソッドの集合に対して型推論とオーバーロードの解決を実施し、このようにして選択されたメソッドを呼び出します。メソッドが見つからない場合は、呼び出しを拡張メソッド呼び出しとして処理することを試みます。

### 7.6.5.2 拡張メソッド呼び出し

次の形式のメソッド呼び出し (7.5.5.1 を参照) の 1 つを例にとります。

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

呼び出しの標準の処理で、適用可能なメソッドが見つからない場合は、構造体を拡張メソッド呼び出しとして処理することが試みられます。 *expr* またはいずれかの *args* のコンパイル時の型が **dynamic** の場合、拡張メソッドは適用されません。

目的は、最適な *type-name* *C* を見つけて対応する次の静的メソッド呼び出しを実行できるようにすることです。

```
C . identifier ( expr )
C . identifier ( expr , args )
C . identifier < typeargs > ( expr )
C . identifier < typeargs > ( expr , args )
```

拡張メソッド *C<sub>i</sub>.M<sub>j</sub>* は、次の条件が満たされる場合に候補となります。

- *C<sub>i</sub>* がジェネリッククラスではなく、入れ子になったクラスでもない
- *M<sub>j</sub>* の名前が *identifier* である
- *M<sub>j</sub>* を上記の静的メソッドとして引数に適用すると、アクセス可能で適用可能になる
- *expr* から *M<sub>j</sub>.* の最初のパラメーターの型への、暗黙の恒等変換、参照変換、またはボックス化変換が存在する

*C* の検索は次のように処理されます。

- 最も近い外側の名前空間宣言から開始し、外側にある各名前空間宣言を経て、外側のコンパイル単位まで、拡張メソッドの候補の集合が連続的に検索されます。
  - 特定の名前空間またはコンパイル単位に、適合する拡張メソッド *M<sub>j</sub>* を持つ非ジェネリック型の宣言 *C<sub>i</sub>* が直接含まれる場合は、それらの拡張メソッドの集合が候補の集合になります。
  - 特定の名前空間またはコンパイル単位にある、名前空間ディレクティブを使用してインポートされた名前空間に、適合する拡張メソッド *M<sub>j</sub>* を持つ非ジェネリック型の宣言 *C<sub>i</sub>* が直接含まれる場合は、それらの拡張メソッドの集合が候補の集合になります。
- 外側の名前空間宣言にもコンパイル単位にも候補の集合が見つからない場合は、コンパイルエラーが発生します。
- それ以外の場合は、7.5.3 で説明したとおり、オーバーロードの解決が候補の集合に適用されます。最適なメソッドが 1 つも見つからない場合は、コンパイルエラーが発生します。
- 型 *C* の中では、最適なメソッドが拡張メソッドとして宣言されます。

*C* をターゲットとして使用し、メソッド呼び出しへ静的メソッド呼び出し(7.5.4 を参照)として処理されます。

前述の規則は、インスタンスメソッドは拡張メソッドよりも優先されること、内側の名前空間宣言にある拡張メソッドは外側の名前空間宣言にある拡張メソッドよりも優先されること、そして、名前空間内で直接宣言された拡張メソッドは、using namespace ディレクティブによりそれと同じ名前空間にインポートされた拡張メソッドよりも優先されることを意味します。次に例を示します。

```
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)
    }
}

```

この例では、**B** のメソッドは、最初の拡張メソッドよりも優先され、**C** のメソッドは両方の拡張メソッドよりも優先されます。

```

public static class C
{
    public static void F(this int i) { Console.WriteLine("C.F({0})", i); }
    public static void G(this int i) { Console.WriteLine("C.G({0})", i); }
    public static void H(this int i) { Console.WriteLine("C.H({0})", i); }
}

namespace N1
{
    public static class D
    {
        public static void F(this int i) { Console.WriteLine("D.F({0})", i); }
        public static void G(this int i) { Console.WriteLine("D.G({0})", i); }
    }
}

namespace N2
{
    using N1;
    public static class E
    {
        public static void F(this int i) { Console.WriteLine("E.F({0})", i); }
    }

    class Test
    {
        static void Main(string[] args)
        {
            1.F();
            2.G();
            3.H();
        }
    }
}

```

この例の出力は次のとおりです。

```

E.F(1)
D.G(2)
C.H(3)

```

`D.G` は `C.G` よりも優先され、`E.F` は `D.F` と `C.F` の両方よりも優先されます。

### 7.6.5.3 デリゲートの呼び出し

デリゲートの呼び出しの場合、*invocation-expression* の *primary-expression* には *delegate-type* の値を指定する必要があります。さらに、*delegate-type* と同じパラメーターリストを持つ関数メンバーであることを考えると、*delegate-type* は、*invocation-expression* の *argument-list* に対して適用可能 (7.5.3.1 を参照) である必要があります。

`D(A)` の形式 (`D` は *delegate-type* の *primary-expression*、`A` は省略可能な *argument-list*) によるデリゲート呼び出しに対する実行時の処理は、次の手順で行われます。

- `D` が評価されます。この評価で例外が発生すると、それ以上の手順は実行されません。
- `D` の値が有効であることをチェックします。`D` の値が `null` の場合、`System.NullReferenceException` がスローされ、それ以上の手順は実行されません。
- それ以外の場合、`D` はデリゲートインスタンスに対する参照です。関数メンバーの呼び出し (7.5.4 を参照) は、デリゲートの呼び出リスト内の各呼び出し可能エンティティで実行されます。呼び出し可能エンティティがインスタンスとインスタンスマソッドで構成されている場合、呼び出すインスタンスは、呼び出し可能エンティティに含まれているインスタンスです。

### 7.6.6 要素へのアクセス

*element-access* は、順に、*primary-no-array-creation-expression*、"[" トークン、*argument-list*、"]" トークンで構成されます。*argument-list* は、コンマで区切られた 1 つ以上の *argument* で構成されます。

*element-access*:

*primary-no-array-creation-expression* [ *argument-list* ]

*element-access* の *argument-list* は `ref` または `out` 引数を含むことはできません。

次のいずれかに該当する場合、*element-access* は動的にバインドされます (7.2.2 を参照)。

- *primary-no-array-creation-expression* のコンパイル時の型が `dynamic` である。
- *argument-list* の少なくとも 1 つの式のコンパイル時の型が `dynamic` であり、*primary-no-array-creation-expression* が配列型ではない。

この場合、コンパイラは *element-access* を `dynamic` 型の値に分類します。*element-access* の意味を特定するための次の規則が、*primary-no-array-creation-expression* のコンパイル時の型ではなく実行時の型と、コンパイル時の型が `dynamic` である *argument-list* 式を使用して、実行時に適用されます。

*primary-no-array-creation-expression* のコンパイル時の型が `dynamic` ではない場合、要素アクセスには 7.5.4 で説明されているコンパイル時の限られたチェックが行われます。

*element-access* の *primary-no-array-creation-expression* が *array-type* の値である場合、*element-access* は配列アクセス (7.6.6.1 を参照) です。それ以外の場合、*primary-no-array-creation-expression* は、1 つ以上のインデクサー メンバーを持つクラス型、構造体型、またはインターフェイス型の変数または値である必要があり、この場合の *element-access* はインデクサー アクセス (7.6.6.2 を参照) です。

### 7.6.6.1 配列アクセス

配列アクセスの場合、*element-access* の *primary-no-array-creation-expression* には *array-type* の値を指定する必要があります。さらに、配列アクセスの *argument-list* は名前付き引数を含むことはできません。

*argument-list* の式の数は *array-type* のランクと同じである必要があり、各式は、`int`、`uint`、`long`、`ulong` のいずれかの型、またはこれらの型の 1 つ以上に暗黙で変換できる型である必要があります。

配列アクセスを評価した結果は、配列の要素型の変数、つまり *argument-list* の式の値によって選択された配列要素です。

$P[A]$  の形式 ( $P$  は *array-type* の *primary-no-array-creation-expression*、 $A$  は *argument-list*) による配列アクセスに対する実行時の処理は、次の手順で行われます。

- $P$  が評価されます。この評価で例外が発生すると、それ以上の手順は実行されません。
- *argument-list* のインデックス式は、左から右に評価されます。各インデックス式の評価に続いて、`int`、`uint`、`long`、`ulong` のいずれかの型に対する暗黙の型変換 (0 を参照) が行われます。このリストの中で暗黙の変換が存在する最初の型が選択されます。たとえば、インデックス式の型が `short` の場合は、`short` から `int` へ、または `short` から `long` への暗黙的な型変換が可能であるため、`int` への暗黙的な型変換が実行されます。インデックス式の評価、またはそれに続く暗黙の変換で例外が発生した場合は、以降のインデックス式は評価されず、これ以降の手順は実行されません。
- $P$  の値が有効であることをチェックします。 $P$  の値が `null` の場合、`System.NullReferenceException` がスローされ、それ以上の手順は実行されません。
- *argument-list* の各式の値を、 $P$  で参照されている配列インスタンスの各次元の実際の範囲に照らしてチェックします。範囲外の値がある場合は `System.IndexOutOfRangeException` がスローされて、以降の手順は実行されません。
- インデックス式で指定された配列要素の位置を計算し、この位置が配列アクセスの結果になります。

### 7.6.6.2 インデクサー アクセス

インデクサー アクセスの場合、*element-access* の *primary-no-array-creation-expression* は、クラス型、構造体型、またはインターフェイス型の変数または値である必要があり、この型は、*element-access* の *argument-list* に関して適用可能なインデクサーを 1 つ以上実装している必要があります。

$P[A]$  の形式 ( $P$  はクラス型、構造体型、またはインターフェイス型である  $T$  の *primary-no-array-creation-expression*、 $A$  は *argument-list*) によるインデクサー アクセスに対するバインディング時の処理は、次の手順で行われます。

- $T$  に含まれるインデクサーの集合を作成します。この集合は、 $T$  または  $T$  の基本型で宣言されていて、`override` 宣言がなく、現在のコンテキストでアクセス可能な (3.5 を参照) すべてのインデクサーで構成されます。
- 集合を絞り込んで、他のインデクサーによって隠ぺいされていない適用可能なインデクサーだけにします。集合内の各インデクサー  $S.I$  ( $S$  は、インデクサー  $I$  が宣言されている型) に対し、次の規則を適用します。
  - $I$  が  $A$  に関して適用可能でない場合は (7.5.3.1 を参照)、 $I$  を集合から削除します。
  - $I$  が  $A$  に関して適用可能である場合は (7.5.3.1 を参照)、 $S$  の基本型で宣言されているすべてのインデクサーを集合から削除します。
  - $I$  が  $A$  に関して適用可能であり (7.5.3.1 を参照)、 $S$  が `object` 以外のクラス型である場合は、インターフェイスで宣言されたすべてのインデクサーを集合から削除します。

- 候補インデクサーの結果セットが空の場合は、適用可能なインデクサーが存在しないため、バインディングエラーになります。
- 一連の候補インデクサーのうち、最適なインデクサーはオーバーロードの解決規則(7.5.3を参照)を使って特定されます。単一の最適なインデクサーが識別されなかった場合、インデクサーアクセスはあいまいであり、バインディングエラーになります。
- argument-list* のインデックス式は、左から右に評価されます。インデクサーアクセスを処理すると、結果はインデクサーアクセスとして分類される式になります。インデクサーアクセス式は、上記の手順で決定されたインデクサーを示し、**P** のインスタンス式と **A** の引数リストが関連付けられています。

使用されるコンテキストに基づいて、インデクサーアクセスによってインデクサーの *get-accessor* または *set-accessor* が呼び出されます。インデクサーアクセスが代入の対象である場合は、*set-accessor* が呼び出されて、新しい値を代入します(7.17.1を参照)。それ以外の場合は、*get-accessor* を呼び出して現在の値を取得します(7.1.1を参照)。

### 7.6.7 this-access

*this-access* は、予約語 **this** で構成されます。

```
this-access:  
    this
```

*this-access* を使用できるのは、インスタンス コンストラクター、インスタンス メソッド、またはインスタンス アクセサーの *block* の中だけです。*this-access* の意味は、次のいずれかです。

- クラスのインスタンス コンストラクターの *primary-expression* で使われる場合、**this** は値として分類されます。値の型は **this** が使用されているクラスのインスタンス型(10.3.1を参照)で、値は生成されているオブジェクトに対する参照です。
- クラスのインスタンス メソッドまたはインスタンス アクセサーの *primary-expression* で使われる場合、**this** は値として分類されます。値の型は **this** が使用されているクラスのインスタンス型(10.3.1を参照)で、値はメソッドまたはアクセサーの呼び出しの対象になっているオブジェクトに対する参照です。
- 構造体のインスタンス コンストラクターの *primary-expression* で使われる場合、**this** は変数として分類されます。変数の型は **this** が使用されている構造体のインスタンス型(10.3.1を参照)で、変数は生成されている構造体を表しています。構造体のインスタンス コンストラクターの **this** 変数の働きは、構造型の **out** パラメーターとまったく同じです。つまり、インスタンス コンストラクターのすべての実行パスで、変数に確実に代入する必要があります。
- 構造体のインスタンス メソッドまたはインスタンス アクセサーの *primary-expression* で使われる場合、**this** は変数として分類されます。変数の型は、これが使用される構造体のインスタンス型(10.3.1を参照)です。
  - メソッドまたはアクセサーが反復子(10.14を参照)ではない場合、**this** 変数はメソッドまたはアクセサーの呼び出しの対象の構造体を表し、構造型の **ref** パラメーターと同じ働きをします。
  - メソッドまたはアクセサーが反復子である場合、**this** 変数はメソッドまたはアクセサーの呼び出しの対象の構造体の *copy* を表し、構造型の *value* パラメーターと同じ働きをします。

上記以外のコンテキストの *primary-expression* で **this** を使用すると、コンパイル エラーになります。特に、静的メソッド、静的プロパティ アクセサー、またはフィールド宣言の *variable-initializer* では、**this** を参照できません。

### 7.6.8 base-access

*base-access* は、予約語 **base** の後に、"."*"* トークンと識別子、または角かっこで囲んだ *argument-list* を続けて構成します。

*base-access:*

```
base . identifier  
base [ argument-list ]
```

*base-access* は、現在のクラスまたは構造体の同じ名前のメンバーによって隠ぺいされている基底クラスのメンバーにアクセスするために使用します。*base-access* を使用できるのは、インスタンス コンストラクター、インスタンスマソッド、またはインスタンスアクセサーの *block* の中だけです。クラスまたは構造体で **base.I** を使用するときの **I** は、そのクラスまたは構造体の基底クラスのメンバーを表している必要があります。同様に、クラスで **base[E]** を使用するときは、基底クラスに適切なインデクサーが存在している必要があります。

**base.I** および **base[E]** の形式の *base-access* 式は、バインディング時には、**((B)this).I** および **((B)this)[E]** という記述とまったく同じように評価されます。ここで、**B** は、生成が行われているクラスまたは構造体の基底クラスです。したがって、**this** が基底クラスのインスタンスとして扱われる場合を除き、**base.I** および **base[E]** は、**this.I** および **this[E]** に対応しています。

*base-access* が仮想関数メンバー(メソッド、プロパティ、またはインデクサー)を参照する場合は、実行時に呼び出す関数メンバーの決定(7.5.4 を参照)が変わります。呼び出す関数メンバーを決定するには、**B** に関する(通常の非ベース アクセスでは、**this** ではなく **B** の実行時の型に関する)関数メンバーの最派生実装(10.6.3 を参照)を見つけます。したがって、**virtual** 関数メンバーの **override** の中では、*base-access* を使うことで、関数メンバーの継承された実装を呼び出すことができます。*base-access* で参照されている関数メンバーが抽象メンバーである場合は、バインディング エラーになります。

### 7.6.9 後置インクリメント演算子と後置デクリメント演算子

*post-increment-expression:*

```
primary-expression ++
```

*post-decrement-expression:*

```
primary-expression --
```

後置インクリメント演算子または後置デクリメント演算子のオペランドには、変数、プロパティ アクセス、またはインデクサー アクセスに分類される式を指定する必要があります。演算の結果は、オペランドと同じ型になります。

*primary-expression* のコンパイル時の型が **dynamic** で、演算子が動的にバインドされる場合(7.2.2 を参照)、*post-increment-expression* または *post-decrement-expression* のコンパイル時の型は **dynamic** であり、次の規則が *primary-expression* の実行時の型を使用して実行時に適用されます。

後置インクリメントまたはデクリメント演算子のオペランドがプロパティ アクセスまたはインデクサー アクセスの場合、そのプロパティまたはインデクサーには **get** アクセサーと **set** アクセサーの両方が必要です。そうでない場合は、バインディング エラーが発生します。

単項演算子のオーバーロードの解決 (7.3.3 を参照) を適用して、演算子の特定の実装が選択されます。`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal` の各型およびすべての列挙型には、定義済みの `++` 演算子と `--` 演算子があります。定義済みの `++` 演算子は、オペランドに 1 を加えた値を返し、定義済みの `--` 演算子は、オペランドから 1 を引いた値を返します。`checked` コンテキストでは、この加算または減算の結果が結果の型の範囲を超える場合、`System.OverflowException` がスローされます。

`x++` または `x--` の形式の後置インクリメントまたはデクリメント演算子の実行時の処理は、次の手順で構成されています。

- `x` が変数の場合。
  - `x` を評価して変数を生成します。
  - `x` の値を保存します。
  - 保存した `x` の値を引数として、選択された演算子を呼び出します。
  - 演算子から返される値を `x` の評価によって得られる場所に格納します。
  - 保存された `x` の値が、演算の結果になります。
- `x` がプロパティ アクセスまたはインデクサー アクセスの場合。
  - `x` に関連付けられたインスタンス式 (`x` が `static` でない場合) および引数リスト (`x` がインデクサー アクセスの場合) を評価し、後で行う `get` アクセサーおよび `set` アクセサーの呼び出しでは、その結果を使用します。
  - `x` の `get` アクセサーを呼び出し、返される値を保存します。
  - 保存した `x` の値を引数として、選択された演算子を呼び出します。
  - 演算子から返される値を `value` 引数として、`x` の `set` アクセサーを呼び出します。
  - 保存された `x` の値が、演算の結果になります。

`++` 演算子と `--` 演算子は前置形式 (7.7.5 を参照) もサポートしています。一般的に、`x++` または `x--` の結果が演算を行う "前" の `x` の値であるのに対し、`++x` または `--x` の結果は演算を行った "後" の `x` の値です。いずれの場合も、演算後の `x` 自体の値は同じになります。

`operator ++` または `operator --` の実装は、後置表記または前置表記のいずれを使っても呼び出すことができます。2つの表記に対して演算子を個別に実装することはできません。

### 7.6.10 new 演算子

`new` 演算子は、型の新しいインスタンスを作成するために使用します。

`new` 式には、次の 3 つの形式があります。

- クラス型および値型の新しいインスタンスを作成するには、オブジェクト作成式を使用します。
- 配列型の新しいインスタンスを作成するには、配列作成式を使用します。
- デリゲート型の新しいインスタンスを作成するには、デリゲート作成式を使用します。

`new` 演算子は、型のインスタンスの作成を意味しますが、メモリの動的な割り当てが行われるとは限りません。特に、値型のインスタンスでは、その値型が存在している変数以外に追加メモリは必要なないので、`new` を使って値型のインスタンスを作成しても、動的な割り当ては行われません。

### 7.6.10.1 オブジェクト作成式

*object-creation-expression* を使用して、*class-type* または *value-type* の新しいインスタンスを作成します。

*object-creation-expression:*

new *type* ( *argument-list*<sub>opt</sub> ) *object-or-collection-initializer*<sub>opt</sub>  
new *type* *object-or-collection-initializer*

*object-or-collection-initializer:*

*object-initializer*  
*collection-initializer*

*object-creation-expression* の *type* には、*class-type*、*value-type*、または *type-parameter* を指定する必要があります。*type* に **abstract class-type** は指定できません。

オプションの *argument-list* (7.5.1 を参照) は、*type* が *class-type* または *struct-type* の場合にだけ使用できます。

オブジェクト初期化子またはコレクション初期化子が含まれていれば、オブジェクト作成式では、コンストラクター引数リストと外側のかっこを省略できます。コンストラクター引数リストと外側の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を含むオブジェクト作成式の処理では、まず、インスタンス コンストラクターの処理、次にオブジェクト初期化子 (7.6.10.2 を参照) またはコレクション初期化子 (7.6.10.3 を参照) によって指定されたメンバーまたは要素の初期化の処理を行います。

省略可能な *argument-list* のいずれかの引数のコンパイル時の型が **dynamic** の場合、*object-creation-expression* は動的にバインドされ (7.2.2 を参照)、*argument-list* のコンパイル時の型が **dynamic** である引数の実行時の型を使用して、次の規則が実行時に適用されます。一方、オブジェクトの作成では 7.5.4 で説明されている限られたコンパイル時チェックが行われます。

**new T(A)** 形式 (*T* は *class-type* または *value-type*、*A* は省略可能な *argument-list*) による *object-creation-expression* に対するバインディング時の処理は、次の手順で行われます。

- *T* が *value-type* で、*A* が指定されていない場合
  - *object-creation-expression* は、既定のコンストラクターの呼び出しです。*object-creation-expression* の結果は、型 *T* の値、つまり 4.1.1 で定義されている *T* に対する既定値です。
- それ以外の場合、*T* が *type-parameter* で、*A* が指定されていない場合
  - *T* に値型の制約またはコンストラクター制約 (10.1.5 を参照) が指定されていない場合は、バインディング エラーが発生します。
  - *object-creation-expression* の結果は、型パラメーターがバインドされているランタイム型の値、つまりその型の既定のコンストラクターを呼び出した結果です。実行時の型には、参照型と値型があります。
- *T* が *class-type* または *struct-type* の場合。
  - *T* が **abstract class-type** の場合は、コンパイル時のエラーになります。
  - 呼び出すインスタンス コンストラクターは、オーバーロードの解決規則 (7.5.3 を参照) を使って特定されます。候補のインスタンス コンストラクターの集合は、*T* に関して適用可能な (7.5.3.1 を参照)、*A* で宣言されているすべてのアクセス可能なインスタンス コンストラクターで構成されます。候補のインスタンス コンストラクターの集合が空である場合、または 1 つ

の最適なインスタンス コンストラクターを識別できなかった場合は、バインディング エラーが発生します。

- *object-creation-expression* の結果は、型  $T$  の値、つまり上記の手順で決定されたインスタンス コンストラクターを呼び出して生成された値です。
- 以上のいずれにも該当しない場合は、*object-creation-expression* が無効であるため、バインディング エラーになります。

*object-creation-expression* が動的にバインドされる場合でも、コンパイル時の型は  $T$  です。

`new T(A)` 形式 ( $T$  は *class-type* または *struct-type*、 $A$  は省略可能な *argument-list*) による *object-creation-expression* に対する実行時の処理は、次の手順で行われます。

- $T$  が *class-type* の場合:
  - $T$  クラスの新しいインスタンスを割り当てます。新しいインスタンスを割り当てるための十分なメモリがない場合は、`System.OutOfMemoryException` がスローされ、以降の手順は実行されません。
  - 新しいインスタンスのすべてのフィールドを既定値で初期化します (5.2 を参照)。
  - 関数メンバー呼び出しの規則 (7.5.4 を参照) に従って、インスタンス コンストラクターを呼び出します。新しく割り当てたインスタンスへの参照がインスタンス コンストラクターに自動的に渡されるため、コンストラクターの中からは `this` としてインスタンスにアクセスできます。
- $T$  が *struct-type* の場合:
  - 一時ローカル変数を割り当てることで、 $T$  型のインスタンスを作成します。*struct-type* のインスタンス コンストラクターは、作成しているインスタンスの各フィールドに値を確実に代入する必要があるため、テンポラリ変数の初期化は必要ありません。
  - 関数メンバー呼び出しの規則 (7.5.4 を参照) に従って、インスタンス コンストラクターを呼び出します。新しく割り当てたインスタンスへの参照がインスタンス コンストラクターに自動的に渡されるため、コンストラクターの中からは `this` としてインスタンスにアクセスできます。

#### 7.6.10.2 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 0 個以上のフィールドまたはプロパティの値を指定します。

```

object-initializer:
  { member-initializer-listopt }
  { 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、"{" トークンと "}" トークン内のコンマで区切った一連のメンバー初期化子から構成されます。各メンバー初期化子では、初期化対象の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を指定し、その後に等号と式、また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する必要があります。オブジェクト初期化子に、同じフィールドまたはプロパティに対する複数のメンバー初期化子を含めるのは誤りです。オブジェクト初期化子は、初期化対象の新しく作成したオブジェクトを参照することはできません。

等号の後に式を指定するメンバー初期化子は、フィールドまたはプロパティへの代入 (7.17.1 を参照) と同様に処理されます。

等号の後にオブジェクト初期化子を指定するメンバー初期化子は、「入れ子になったオブジェクト初期化子」、つまり埋め込みオブジェクトの初期化です。フィールドまたはプロパティに新しい値を代入する代わりに、入れ子になったオブジェクト初期化子の代入が、フィールドまたはプロパティのメンバーへの代入として処理されます。入れ子になったオブジェクト初期化子は、値型のプロパティ、または値型の読み取り専用のフィールドには適用できません。

等号の後にコレクション初期化子を指定するメンバー初期化子は、埋め込みコレクションの初期化です。フィールドまたはプロパティに新しいコレクションを代入する代わりに、初期化子で指定された要素が、フィールドまたはプロパティによって参照されるコレクションに追加されます。フィールドまたはプロパティは、7.6.10.3 で指定された要件を満たすコレクション型である必要があります。

次のクラスは 2 つの座標により 1 つの点を表します。

```
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して作成し、初期化できます。

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

上記は以下と同じ効果があります。

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

ここで、\_\_a は参照もアクセスもできない一時変数です。次のクラスは、2 つの点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して作成し、初期化できます。

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

上記は以下と同じ効果があります。

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

ここで、`__r`、`__p1`、および`__p2`は、参照もアクセスもできない一時変数です。

`Rectangle`'s のコンストラクターで、次のように 2 つの埋め込み `Point` インスタンスが割り当てられた場合

```
public class Rectangle
{
    Point p1 = new Point();
    Point p2 = new Point();
    public Point P1 { get { return p1; } }
    public Point P2 { get { return p2; } }
}
```

新しいインスタンスを代入する代わりに、次の構造体を使用して埋め込み `Point` インスタンスを初期化できます。

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

上記は以下と同じ効果があります。

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

### 7.6.10.3 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 }
{ element-initializer-list , }
```

  

```
element-initializer-list:
element-initializer
element-initializer-list , element-initializer
```

```

element-initializer:
  non-assignment-expression
  { expression-list }

expression-list:
  expression
  expression-list , expression

```

コレクション初期化子は、"{" トークンと "}" トークン内のコンマで区切った要素初期化子のシーケンスから構成されます。各要素初期化子は、初期化対象のコレクションオブジェクトに追加する要素を指定し、"{" トークンと "}" トークン内のコンマで区切った式のリストから構成されます。単一式の要素初期化子は、かっこなしで作成することができますが、メンバー初期化子とのあいまい性を防ぐため、代入式にすることはできません。*non-assignment-expression* の生成は 7.18 で定義されています。

以下は、コレクション初期化子を含むオブジェクト作成式の例です。

```
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、

`System.Collections.IEnumerable` を実装する型である必要があります。この型でない場合は、コンパイルエラーが発生します。順に指定された要素ごとに、コレクション初期化子によりターゲットオブジェクトで要素初期化子の式リストを引数リストとして `Add` メソッドが呼び出され、各呼び出しに標準のオーバーロードの解決が適用されます。したがって、コレクションオブジェクトには、各要素初期化子に対する適切な `Add` メソッドが含まれている必要があります。

次のクラスは、名前、および電話番号のリストを含む連絡先を表しています。

```

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、次のようにして作成し初期化できます。

```

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" }
    }
};

```

上記は以下と同じ効果があります。

```

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

```

ここで、`__clist`、`__c1`、および`__c2`は、参照もアクセスもできない一時変数です。

#### 7.6.10.4 配列作成式

*array-creation-expression* は、*array-type* の新しいインスタンスを作成するために使用します。

*array-creation-expression*:

```

new non-array-type [ expression-list ] rank-specifiersopt array-initializeropt
new array-type array-initializer
new rank-specifier array-initializer

```

配列作成式の最初の形式は、式リストから個別の各式を削除した結果である型の配列インスタンスを割り当てます。たとえば、配列作成式 `new int[10, 20]` からは `int[,]` 型の配列インスタンスが生成されて、配列作成式 `new int[10][,]` からは `int[][]` 型の配列が生成されます。式リストの各式の型は、`int`、`uint`、`long`、`ulong` のいずれか、またはこれらの型の 1 つ以上に暗黙で変換できる型である必要があります。各式の値によって、新しく割り当てられた配列インスタンスの対応する次元の長さが決まります。配列次元の長さは負の値にできないため、式リストに負の値を持つ *constant-expression* があるとコンパイルエラーになります。

`unsafe` コンテキスト (18.1 を参照) の場合を除き、配列のレイアウトは指定されません。

第 1 の形式の配列作成式で配列初期化子を指定する場合は、式リストの各式には定数を指定し、式リストで指定するランクと次元の長さ配列初期化子のものと一致している必要があります。

2 番目または 3 番目の形式の配列作成式では、指定された配列型のランクまたはランク指定子は配列初期化子のランクと一致している必要があります。個別の次元の長さは、配列初期化子の対応する各入れ子レベルにおける要素の数から推測されます。次に例を示します。

```
new int[,] {{0, 1}, {2, 3}, {4, 5}}
```

上の式は、下の式と正確に対応しています。

```
new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}
```

3 番目の形式の配列作成式は、“暗黙に型が指定される配列作成式”と呼ばれます。これは 2 番目の形式と同様ですが、配列の要素型は明示的に指定されず、配列初期化子内の式の集合で共通した最適な型 (7.5.2.14 を参照) として決定されます。多次元配列、つまり *rank-specifier* に少なくとも 1 つのコソマが含まれる配列の場合、この集合は入れ子になった *array-initializer* にあるすべての *expression* を構成します。

配列初期子については 12.6 で詳細に説明します。

配列作成式を評価した結果は、値として分類されます。つまり、新しく作成された配列インスタンスに対する参照です。配列作成式の実行時の処理は、以下の手順で行われます。

- *expression-list* における次元の長さの式は、左から右に評価されます。各インデックス式の評価に続いて、`int`、`uint`、`long`、`ulong` のいずれかの型に対する暗黙の型変換 (0 を参照) が行われます。このリストの中で暗黙の変換が存在する最初の型が選択されます。式の評価、またはそれに続く暗黙の変換で例外が発生した場合は、以降の式は評価されず、これ以降の手順は実行されません。
- 計算された次元の長さの値を次の方法で検査します。ゼロ未満の値が 1 つでもある場合は、`System.OverflowException` をスローし、以降の手順を実行しません。
- 指定された次元の長さで配列インスタンスを割り当てます。新しいインスタンスを割り当てるための十分なメモリがない場合は、`System.OutOfMemoryException` がスローされ、以降の手順は実行されません。
- 新しい配列インスタンスのすべての要素を既定値で初期化します (5.2 を参照)。
- 配列作成式に配列初期化子が含まれている場合は、配列初期化子の各式を評価し、対応する配列要素に代入します。評価と代入は、配列初期化子で式が記述されている順序で実行します。つまり、要素の初期化は、インデックスの昇順に、右端の次元のインデックスを最初に増加させて行います。指定された式の評価、または対応する配列要素への代入で例外が発生した場合は、それ以降の要素は初期化しません。したがって、残りの要素は既定値のままになります。

配列作成式では、配列型の要素を持つ配列をインスタンス化できますが、このような配列の要素は、手動で初期化する必要があります。次に例を示します。

```
int[][] a = new int[100][];
```

このステートメントでは、`int[]` 型の要素を 100 個持つ 1 次元配列が作成されます。各要素の初期値は `null` です。同じ配列作成式でサブ配列のインスタンス化も行うことはできません。

```
int[][] a = new int[100][5]; // Error
```

このコードはコンパイルエラーになります。サブ配列のインスタンス化は、代わりに手動で行う必要があります。

```
int[][] a = new int[100][];
for (int i = 0; i < 100; i++) a[i] = new int[5];
```

配列の配列が "四角形" の形状をしている場合、つまりサブ配列がすべて同じ長さの場合は、多次元配列を使う方が効率的です。上の例では、配列の配列をインスタンス化すると、101 個のオブジェクト (外側の配列が 1 個と 100 個のサブ配列) が作成されます。次に示すのは多次元配列の例です。

```
int[,] = new int[100, 5];
```

この例では、2 次元配列のオブジェクトが 1 つ作成されるだけで、1 つのステートメントで割り当てを行うことができます。

以下は暗黙に型が指定された配列作成式の例です。

```
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" }; // Error
```

`int` も `string` も他の型に暗黙に変換できず、共通した最適な型がないため、最後の式はコンパイルエラーになります。この場合、明示的に型が指定された配列作成式を使用する必要があります。たと

えば型を `object[]` に指定します。また、要素の 1 つを共通の基本型にキャストすることもできます。これにより、この基本型が推論された要素型となります。

暗黙に型が指定された配列作成式では、匿名オブジェクト初期化子 (7.6.10.6 を参照) と統合して、匿名で型が指定されたデータ構造を作成できます。次に例を示します。

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

#### 7.6.10.5 デリゲート作成式

*delegate-creation-expression* を使用して、*delegate-type* の新しいインスタンスを作成します。

*delegate-creation-expression*:  
 new *delegate-type* ( *expression* )

デリゲート作成式の引数には、メソッドグループ、匿名関数、またはコンパイル時の型 `dynamic` または *delegate-type* の値を指定する必要があります。引数がメソッドグループの場合、引数は、デリゲートを作成する対象のメソッドとオブジェクト(インスタンスマソッドの場合)を示します。引数が匿名関数である場合、それはデリゲートターゲットのパラメータとメソッド本体を直接定義します。引数が値の場合、引数は、コピーを作成するデリゲートインスタンスを示します。

*expression* のコンパイル時の型が `dynamic` で、*delegate-creation-expression* が動的にバインド (7.2.2 を参照) される場合、*expression* の実行時の型を使用して実行時に次の規則が適用されます。それ以外の場合は、コンパイル時に規則が適用されます。

`new D(E)` の形式 (*D* は *delegate-type*、*E* は *expression*) による *delegate-creation-expression* のバインディング時の処理は、次の手順で行われます。

- *E* がメソッドグループの場合、デリゲート作成式は、*E* から *D* へのメソッドグループの変換 (6.6 を参照) と同様に処理されます。
- *E* が匿名関数の場合、デリゲート作成式は、*E* から *D* への匿名関数の変換 (0 を参照) と同様に処理されます。
- *E* が値の場合は、*E* は *D* との互換性を持っている (15.1 を参照) 必要があり、結果は、*E* と同じ呼び出しリストを参照する新しく作成された型 *D* のデリゲートへの参照になります。*E* が *D* との互換性を持たない場合は、コンパイル エラーが発生します。

`new D(E)` の形式 (*D* は *delegate-type*、*E* は *expression*) による *delegate-creation-expression* の実行時の処理は、次の手順で行われます。

- *E* がメソッドグループの場合、デリゲート作成式は、*E* から *D* へのメソッドグループの変換 (6.6 を参照) として評価されます。
- *E* が匿名関数の場合、デリゲート作成は、*E* から *D* への匿名関数の変換 (0 を参照) として評価されます。
- *E* が *delegate-type* の値の場合。

- `E` が評価されます。この評価で例外が発生すると、それ以上の手順は実行されません。
- `E` の値が `null` の場合、`System.NullReferenceException` がスローされ、それ以上の手順は実行されません。
- デリゲート型 `D` の新しいインスタンスを割り当てます。新しいインスタンスを割り当てるための十分なメモリがない場合は、`System.OutOfMemoryException` がスローされ、以降の手順は実行されません。
- 新しいデリゲートインスタンスを、`E` で指定されるデリゲートインスタンスと同じ呼び出しリストで初期化します。

デリゲートの呼び出しリストは、デリゲートがインスタンス化されると決定されて、デリゲートの有効期間の間は一定に保たれます。つまり、デリゲートの対象となる呼び出し可能なエンティティは、いったん作成された後では変更できません。2つのデリゲートが結合されるか、または1つのデリゲートが別のデリゲートから削除されると(15.1を参照)、新しいデリゲートが作成され、既存のデリゲートの内容は変更されません。

プロパティ、インデクサー、ユーザー定義演算子、インスタンスコンストラクター、デストラクター、または静的コンストラクターを示すデリゲートは作成できません。

先に説明したように、メソッドグループからデリゲートを作成すると、デリゲートの仮パラメーターリストおよび戻り値の型により、オーバーロードされたメソッドの中で選択されるものが決まります。次に例を示します。

```
delegate double DoubleFunc(double x);
class A
{
    DoubleFunc f = new DoubleFunc(Square);
    static float Square(float x) {
        return x * x;
    }
    static double Square(double x) {
        return x * x;
    }
}
```

`A.f` フィールドは、2番目の `Square` メソッドを示すデリゲートで初期化されます。これは、このメソッドが、`DoubleFunc` の仮パラメーター リストおよび戻り値の型と完全に一致するためです。2番目の `Square` メソッドがない場合は、コンパイル エラーが発生します。

#### 7.6.10.6 匿名オブジェクト作成式

*anonymous-object-creation-expression* は、匿名型のオブジェクトの作成に使用します。

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

anonymous-object-initializer:
    { member-declarator-listopt }
    { member-declarator-list , }

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

```

member-declarator:
  simple-name
  member-access
  base-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** から直接継承した名前なしのクラス型です。匿名型のメンバーは、型のインスタンスの作成に使用された匿名オブジェクト初期化子から推論された一連の読み取り専用プロパティです。次に具体的な例を示します。

```
new { p1 = e1 , p2 = e2 , ⋯ pn = en }
```

上記の形式の匿名オブジェクト初期化子は、次の形式の匿名型を宣言します。

```

class __Anonymous1
{
    private readonly T1 f1 ;
    private readonly T2 f2 ;
    ...
    private readonly Tn fn ;

    public __Anonymous1(T1 a1, T2 a2, ⋯, Tn an) {
        f1 = a1 ;
        f2 = a2 ;
        ...
        fn = an ;
    }

    public T1 p1 { get { return f1 ; } }
    public T2 p2 { get { return f2 ; } }
    ...
    public Tn pn { get { return fn ; } }

    public override bool Equals(object __o) { ⋯ }
    public override int GetHashCode() { ⋯ }
}

```

ここで、各  $T_x$  は対応する式  $e_x$  の型です。*member-declarator* で使用される式には型が必要です。したがって、*member-declarator* 内の式が **null** または匿名関数である場合、コンパイル エラーになります。さらに、式の型が **unsafe** である場合もコンパイル エラーになります。

**Equals** メソッドに対する匿名型およびパラメーターの名前は、コンパイラによって自動的に作成されるため、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、名前およびコンパイル時の型が同じ一連のプロパティを同じ順序で指定する 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。

次に例を示します。

```

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

```

最後の行の代入は、**p1** と **p2** が同じ匿名型であるために、許可されています。

匿名型の `Equals` メソッドと `GetHashCode` メソッドは、`object` から継承されたメソッドをオーバーライドし、プロパティの `Equals` および `GetHashCode` に関連して定義されるので、同じ匿名型の 2 つのインスタンスは、それらのすべてのプロパティが等しい場合にのみ、等しくなります。

メンバー宣言子は、簡易名 (7.5.2 を参照)、メンバーアクセス (7.5.4 を参照)、またはベースアクセス (7.6.8 を参照) として省略することができます。これは、"予測初期化子" と呼ばれ、同じ名前のプロパティへの宣言および代入の省略表現です。次に具体的な例を示します。

*identifier*   *expr . identifier*

上記の形式のメンバー宣言子は、それぞれ以下と同等です。

*identifier = identifier*                                   *identifier = expr . identifier*

したがって、予測初期化子では、*identifier* が値、および値の代入先であるフィールドまたはプロパティの両方を選択します。直感的には、予測初期化子は値のみではなく、値の名前も予測します。

### 7.6.11 `typeof` 演算子

`typeof` 演算子を使用して型の `System.Type` オブジェクトを取得します。

```

typeof-expression:
  typeof ( type )
  typeof ( unbound-type-name )
  typeof ( void )

unbound-type-name:
  identifier generic-dimension-specifieropt
  identifier :: identifier generic-dimension-specifieropt
  unbound-type-name . identifier generic-dimension-specifieropt

generic-dimension-specifier:
  < commasopt >

commas:
  ,
  commas ,

```

第 1 の形式の `typeof-expression` は、`typeof` キーワードと、かっこで囲んだ `type` です。この形式の式の結果は、指定した型に対する `System.Type` オブジェクトです。特定の型に対する `System.Type` オブジェクトは、1 つだけです。つまり、型 `T` に関して `typeof(T) == typeof(T)` が常に成り立ちます。`type` を `dynamic` にすることはできません。

2 番目の形式の `typeof-expression` は、`typeof` キーワードと、かっこで囲んだ `unbound-type-name` です。`unbound-type-name` は `type-name` (3.8 を参照) とよく似ていますが、`unbound-type-name` は `generic-dimension-specifier` を含み、`type-name` は `type-argument-list` を含むという点が異なります。`typeof-expression` のオペラントがトークンのシーケンスであり、`unbound-type-name` と `type-name` の両方の文法を満たす場合、つまり `typeof-expression` が `generic-dimension-specifier` も `type-argument-list` も含まない場合、そのトークンのシーケンスは `type-name` と見なされます。`unbound-type-name` の意味は、次の規則に従って決定されます。

- 各 `generic-dimension-specifier` を各 `type-argument` と同数のコンマとキーワード `object` を含む `type-argument-list` で置き換えることにより、一連のトークンを `type-name` に変換します。
- 得られた `type-name` を評価し、すべての型パラメーターの制約は無視します。

- 得られた構築された型に関連付けられている非バインド ジェネリック型に *unbound-type-name* を解決します (4.4.3 を参照)。

*typeof-expression* の結果は、得られた非バインド ジェネリック型の `System.Type` オブジェクトです。

3 番目の形式の *typeof-expression* は、`typeof` キーワードと、かつて囲んだ `void` キーワードです。この形式の式の結果は、型が存在しないことを表す `System.Type` オブジェクトです。`typeof(void)` から返される型オブジェクトは、いずれかの型に対して返される型オブジェクトとは異なります。この特殊な型オブジェクトは、言語のメソッドでリフレクションを使用できるようにするクラス ライブラリにおいて役に立ちます。このようなメソッドでは、`void` メソッドを含む任意のメソッドの戻り値の型を `System.Type` のインスタンスで表す手段が必要です。

`typeof` 演算子は型パラメーターに対して使用できます。結果は、型パラメーターにバインドされた実行時の型の `System.Type` オブジェクトになります。また `typeof` 演算子は、構築された型または非バインド ジェネリック型に対しても使用できます (4.4.3 を参照)。非バインド ジェネリック型の `System.Type` オブジェクトは、インスタンス型の `System.Type` オブジェクトとは異なります。インスタンス型は実行時には常にクローズ構築型であるため、`System.Type` オブジェクトは使用される実行時の型引数に依存しますが、非バインド ジェネリック型には型引数がありません。

次の例を参照してください。

```
using System;
class X<T>
{
    public static void PrintTypes() {
        Type[] t = {
            typeof(int),
            typeof(System.Int32),
            typeof(string),
            typeof(double[]),
            typeof(void),
            typeof(T),
            typeof(X<T>),
            typeof(X<X<T>>),
            typeof(X<>)
        };
        for (int i = 0; i < t.Length; i++) {
            Console.WriteLine(t[i]);
        }
    }
}
class Test
{
    static void Main() {
        X<int>.PrintTypes();
    }
}
```

このプログラムは次のように出力されます。

```
System.Int32
System.Int32
System.String
System.Double[]
System.Void
System.Int32
X`1[System.Int32]
X`1[X`1[System.Int32]]
X`1[T]
```

`int` と `System.Int32` が同じ型であることに注意してください。

また、`typeof(x<>)` の結果は型引数に依存しませんが、`typeof(x<T>)` の結果は依存します。

### 7.6.12 checked 演算子と unchecked 演算子

`checked` 演算子と `unchecked` 演算子は、整数型の算術演算および変換に対する "オーバーフロー チェック コンテキスト" を制御するために使用します。

```
checked-expression:  
  checked ( expression )  
  
unchecked-expression:  
  unchecked ( expression )
```

`checked` 演算子は、かっこ内の式を `checked` (チェックあり) のコンテキストで評価し、`unchecked` 演算子は、かっこ内の式を `unchecked` (チェックなし) のコンテキストで評価します。`checked-expression` または `unchecked-expression` は、`parenthesized-expression` (7.6.3 を参照) に厳密に対応していますが、かっこ内の式が所定のオーバーフロー チェック コンテキストで評価される点が異なっています。

オーバーフロー チェック コンテキストは、`checked` ステートメントおよび `unchecked` ステートメント (8.11 を参照) を使って制御することもできます。

以下の演算は、`checked` と `unchecked` の演算子およびステートメントで設定されたオーバーフロー チェック コンテキストによって影響を受けます。

- 定義済みの `++` および `--` 単項演算子 (7.6.9 および 7.7.5 を参照)。オペランドが整数型の場合。
- 定義済みの `-` 単項演算子 (7.7.2 を参照)。オペランドが整数型の場合。
- 定義済みの `+`、`-`、`*`、および `/` 二項演算子 (7.8 を参照)。両方のオペランドとも整数型の場合。
- ある整数型から別の整数型へ、または `float` か `double` から整数型への明示的な数値変換 (6.2.1 を参照)。

上記の演算において生成された値が、結果を格納する型に対して大きすぎると、演算が実行されたコンテキストによって結果の扱いが制御されます。

- `checked` コンテキストでは、演算が定数式 (7.19 を参照) であるとコンパイル エラーが発生します。それ以外の場合は、実行時に演算が実行された時点で、`System.OverflowException` がスローされます。
- `unchecked` コンテキストでは、結果の格納される型に収まらない上位ビットが破棄されて、結果が切り詰められます。

非定数式 (実行時に評価される式) が `checked` または `unchecked` の演算子またはステートメントで囲まれていない場合は、外部要因 (コンパイラ スイッチや実行環境の構成など) が `checked` での評価を要求していない限り、既定のオーバーフロー チェック コンテキストは `unchecked` です。

定数式 (コンパイル時に完全に評価できる式) の場合は、既定のオーバーフロー チェック コンテキストは常に `checked` です。定数式が明示的に `unchecked` コンテキストの中に置かれていなければ、式のコンパイル時評価においてオーバーフローが発生すると、常にコンパイル エラーになります。

匿名関数の本体は、匿名関数がある `checked` コンテキストにも `unchecked` コンテキストにも依存しません。

次に例を示します。

```
class Test
{
    static readonly int x = 1000000;
    static readonly int y = 1000000;

    static int F() {
        return checked(x * y);      // Throws OverflowException
    }

    static int G() {
        return unchecked(x * y);   // Returns -727379968
    }

    static int H() {
        return x * y;              // Depends on default
    }
}
```

この例では、いずれの式もコンパイル時には評価できないため、コンパイルエラーは報告されません。実行時には、**F** メソッドは `System.OverflowException` をスローし、**G** メソッドは `-727379968` (範囲外の結果の下位 32 ビット) を返します。**H** メソッドの動作はコンパイルに対する既定のオーバーフロー チェック コンテキストによって異なりますが、**F** または **G** と同じになります。

次に例を示します。

```
class Test
{
    const int x = 1000000;
    const int y = 1000000;

    static int F() {
        return checked(x * y);      // Compile error, overflow
    }

    static int G() {
        return unchecked(x * y);   // Returns -727379968
    }

    static int H() {
        return x * y;              // Compile error, overflow
    }
}
```

この例では、式が `checked` コンテキストで評価されるため、**F** および **H** の定数式を評価したときに発生するオーバーフローは、コンパイルエラーとして報告されます。**G** の定数式の評価でもオーバーフローが発生しますが、`unchecked` コンテキストで評価が行われるため、オーバーフローは報告されません。

`checked` 演算子と `unchecked` 演算子は、"( " トークンと ")" トークンの内側に記述されている演算に対するオーバーフロー チェック コンテキストに対してだけ、影響を与えます。対象となる式を評価した結果として呼び出される関数メンバーには影響を与えません。次に例を示します。

```
class Test
{
    static int Multiply(int x, int y) {
        return x * y;
    }

    static int F() {
        return checked(Multiply(1000000, 1000000));
    }
}
```

F の中で使われている `checked` は、`Multiply` の `x * y` の評価には影響を与えないため、`x * y` は既定のオーバーフロー チェック コンテキストで評価されます。

`unchecked` 演算子は、16 進表記の中に符号付き整数型の定数を記述するときに役に立ちます。次に例を示します。

```
class Test
{
    public const int AllBits = unchecked((int)0xFFFFFFFF);
    public const int HighBit = unchecked((int)0x80000000);
}
```

上の例の 16 進定数は、いずれも `uint` 型です。定数は `int` の範囲外なので、`unchecked` 演算子がないと、`int` へのキャストでコンパイル エラーが発生します。

`checked` 演算子と `unchecked` 演算子、およびステートメントにより、数値計算の一部を制御できます。ただし、一部の数値演算子の動作は、オペランドのデータ型に依存します。たとえば、2 つの `decimal` 値を乗算した結果は、明示的に宣言された `unchecked` 構造内でも、常にオーバーフローによる例外が発生します。一方、2 つの `float` 値を乗算した結果は、明示的に宣言された `checked` 構造内でも、オーバーフローによる例外は発生しません。さらに、他の演算子は、既定であるか明示的であるかにかかわらず、チェックのモードに影響されることはありません。

### 7.6.13 既定値の式

既定値の式を使用して、型の既定値 (5.2 を参照) を取得できます。一般的に、既定値の式は型パラメーターに対して使用されます。これは、型パラメーターが値型か参照型かどうかがわからない場合があるためです。型パラメーターが参照型とわかっている場合を除き、`null` リテラルから型パラメーターへの変換は存在しません。

*default-value-expression:*  
*default* ( *type* )

*default-value-expression* の *type* が実行時に参照型に評価される場合、結果はその型に変換された `null` になります。*default-value-expression* の *type* が実行時に値型に評価される場合、結果は *value-type* の既定値になります (4.1.2 を参照)。

*default-value-expression* は、型が参照型または参照型とわかっている型パラメーター (7.19 を参照) の場合は定数式 (10.1.5 を参照) になります。また、*default-value-expression* は、型が値型 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal`、`bool`、または任意の列挙型の場合は、定数式になります。

### 7.6.14 匿名メソッドの式

*anonymous-method-expression* は、匿名関数を定義する 2 つの方法のいずれかです。これらの方針については、7.15 で詳しく説明します。

## 7.7 単項演算子

`+`、`-`、`!`、`~`、`++`、`--`、キャストおよび `await` 演算子は、単項演算子と呼ばれます。

```

unary-expression:
  primary-expression
  + unary-expression
  - unary-expression
  ! unary-expression
  ~ unary-expression
  pre-increment-expression
  pre-decrement-expression
  cast-expression
  await-expression

```

*unary-expression* のオペランドのコンパイル時の型が `dynamic` の場合は、動的にバインドされます (7.2.2 を参照)。この場合、*unary-expression* のコンパイル時の型は `dynamic` であり、以下で説明する解決はオペランドの実行時の型を使用して実行時に行われます。

### 7.7.1 単項プラス演算子

`+x` 形式の演算では、単項演算子のオーバーロードの解決 (7.3.3 を参照) を適用して、演算子の実装が選択されます。オペランドが、選択した演算子のパラメーターの型に変換され、その結果の型が演算子の戻り値の型になります。定義済みの単項プラス演算子は次のとおりです。

```

int operator +(int x);
uint operator +(uint x);
long operator +(long x);
ulong operator +(ulong x);
float operator +(float x);
double operator +(double x);
decimal operator +(decimal x);

```

これらの演算子の結果はいずれも、単にオペランドの値です。

### 7.7.2 単項マイナス演算子

`-x` 形式の演算では、単項演算子のオーバーロードの解決 (7.3.3 を参照) を適用して、演算子の実装が選択されます。オペランドが、選択した演算子のパラメーターの型に変換され、その結果の型が演算子の戻り値の型になります。定義済みの否定演算子は次のとおりです。

- 整数の否定：

```

int operator -(int x);
long operator -(long x);

```

結果は、ゼロから  $x$  を引いて得られます。 $x$  の値がオペランドの型で表現できる最も小さい値である場合 (`int` の場合は  $-2^{31}$ , `long` の場合は  $-2^{63}$ )、 $x$  の算術否定はオペランドの型の中で表現できません。これが `checked` コンテキスト内で発生すると、`System.OverflowException` がスローされ、`unchecked` コンテキスト内で発生すると、オペランドの値が結果となり、オーバーフローは報告されません。

否定演算子のオペランドが `uint` 型の場合は、`long` 型に変換されて、結果の型は `long` になります。例外は、`int` 値  $-2147483648$  ( $-2^{31}$ ) を 10 進数整数リテラル (2.4.4.2 を参照) として記述できる規則です。

否定演算子のオペランドが `ulong` 型の場合は、コンパイル エラーになります。例外は、`long` 値  $-9223372036854775808$  ( $-2^{63}$ ) を 10 進数整数リテラル (2.4.4.2 を参照) として記述できる規則です。

- 浮動小数点数の否定：

```
float operator -(float x);
double operator -(double x);
```

結果は、 $x$  の値の符号を反転させたものです。 $x$  が NaN の場合は、結果も NaN です。

- 10 進数の否定。

```
decimal operator -(decimal x);
```

結果は、ゼロから  $x$  を引いて得られます。10 進数の否定は、`System.Decimal` 型の単項マイナス演算子を使う場合と等価です。

### 7.7.3 論理否定演算子

$!x$  形式の演算では、単項演算子のオーバーロードの解決 (7.3.3 を参照) を適用して、演算子の実装が選択されます。オペランドが、選択した演算子のパラメーターの型に変換され、その結果の型が演算子の戻り値の型になります。定義済みの論理否定演算子は 1 つだけです。

```
bool operator !(bool x);
```

この演算子は、オペランドの論理否定を計算します。オペランドが `true` の場合、結果は `false` になります。オペランドが `false` の場合、結果は `true` になります。

### 7.7.4 ビットごとの補数演算子

$\sim x$  形式の演算では、単項演算子のオーバーロードの解決 (7.3.3 を参照) を適用して、演算子の実装が選択されます。オペランドが、選択した演算子のパラメーターの型に変換され、その結果の型が演算子の戻り値の型になります。定義済みのビットごとの補数演算子は、次のとおりです。

```
int operator ~(int x);
uint operator ~(uint x);
long operator ~(long x);
ulong operator ~(ulong x);
```

これらの演算子の結果はいずれも、 $x$  のビットごとの補数です。

すべての列挙型 `E` には、暗黙で、次に示すビットごとの補数演算子が用意されています。

```
E operator ~(E x);
```

$\sim x$  を評価した結果は、 $(E)(\sim(U)x)$  を評価した結果とまったく同じです。ただし、`E` への変換は常に `unchecked` コンテキスト内であるかのように実行されます (§ 7.6.12 を参照)。ここで、 $x$  は、`U` 型が基になった列挙型 `E` の式です。

### 7.7.5 前置インクリメント演算子と前置デクリメント演算子

*pre-increment-expression:*  
    `++ unary-expression`

*pre-decrement-expression:*  
    `-- unary-expression`

前置インクリメント演算子または前置デクリメント演算子のオペランドには、変数、プロパティ アクセス、またはインデクサー アクセスに分類される式を指定する必要があります。演算の結果は、オペランドと同じ型の値になります。

前置インクリメントまたはデクリメント演算子のオペランドがプロパティ アクセスまたはインデクサー アクセスの場合、そのプロパティまたはインデクサーには `get` アクセサーと `set` アクセサーの両方が必要です。そうでない場合は、バインディング エラーが発生します。

単項演算子のオーバーロードの解決 (7.3.3 を参照) を適用して、演算子の特定の実装が選択されます。`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal` の各型およびすべての列挙型には、定義済みの `++` 演算子と `--` 演算子があります。定義済みの `++` 演算子は、オペランドに 1 を加えた値を返し、定義済みの `--` 演算子は、オペランドから 1 を引いた値を返します。`checked` コンテキストでは、この加算または減算の結果が結果の型の範囲を超える場合、`System.OverflowException` がスローされます。

`++x` または `--x` の形式の前置インクリメントまたはデクリメント演算子の実行時の処理は、次の手順で構成されています。

- `x` が変数の場合。
  - `x` を評価して変数を生成します。
  - `x` の値を引数として、選択された演算子を呼び出します。
  - 演算子から返される値を `x` の評価によって得られる場所に格納します。
  - 演算子から返された値が、演算の結果になります。
- `x` がプロパティ アクセスまたはインデクサー アクセスの場合。
  - `x` に関連付けられたインスタンス式 (`x` が `static` でない場合) および引数リスト (`x` がインデクサー アクセスの場合) を評価し、後で行う `get` アクセサーおよび `set` アクセサーの呼び出しでは、その結果を使用します。
  - `x` の `get` アクセサーを呼び出します。
  - `get` アクセサーから返される値を引数として、選択された演算子を呼び出します。
  - 演算子から返される値を `value` 引数として、`x` の `set` アクセサーを呼び出します。
  - 演算子から返された値が、演算の結果になります。

`++` 演算子と `--` 演算子は後置形式 (7.6.9 を参照) もサポートしています。一般的に、`x++` または `x--` の結果が演算を行う "前" の `x` の値であるのに対し、`++x` または `--x` の結果は演算を行った "後" の `x` の値です。いずれの場合も、演算後の `x` 自体の値は同じになります。

`operator ++` または `operator --` の実装は、後置表記または前置表記のいずれを使っても呼び出すことができます。2つの表記に対して演算子を個別に実装することはできません。

## 7.7.6 キャスト式

式を特定の型に明示的に変換するには、*cast-expression* を使用します。

*cast-expression*:  
   `( type ) unary-expression`

$(T)E$  の形式 ( $T$  は *type*、 $E$  は *unary-expression*) の *cast-expression* は、 $E$  の値を  $T$  型に明示的に変換します (6.2 を参照)。 $E$  から  $T$  への明示的な変換が存在していない場合は、バインディング エラーになります。存在している場合は、明示的変換によって生成される値が結果になります。 $E$  が変数を表している場合でも、結果は常に値です。

*cast-expression* の文法のために、構文にはあいまいな部分があります。たとえば、 $(x)-y$  という式は、*cast-expression* ( $x$  型に対する  $-y$  のキャスト) または *parenthesized-expression* と組み合わされた *additive-expression* (値  $x - y$  を計算する) のいずれにも解釈できます。

*cast-expression* のあいまいさを解決するために、次の規則が設けられています。かつこの中に 1 つ以上のトークン(2.3.3 を参照)のシーケンスがある場合は、以下の条件の少なくとも 1 つが満たされている場合に限り、*cast-expression* の開始と見なします。

- トークンのシーケンスが、*type* については正しい文法になっているが、*expression* については正しくない。
- トークンのシーケンスが *type* の正しい文法になっており、右かつこのすぐ後のトークンが、"~" トークン、"!" トークン、 "(" トークン、*identifier* (2.4.1 を参照)、*literal* (2.4.4 を参照)、または **as** および **is** 以外の任意の *keyword* (2.4.3 を参照) である。

上の規則で使われている "正しい文法" という表現は、トークンのシーケンスが特定の文法生成規則に準拠している必要がある、ということだけを意味します。シーケンスを構成する識別子の実際の意味については考慮しません。たとえば、*x* と *y* が識別子の場合、*x.y* は型に関して正しい文法です。*x.y* が実際に型を表しているかどうかには関係ありません。

あいまいさを排除するための規則に従うと、*x* および *y* が識別子である場合は、(*x*)*y*、(*x*)(*y*)、および (*x*)(-*y*) は *cast-expressions* であり、(*x*)-*y* は *x* が型を示していても *cast-expressions* ではありません。ただし、*x* が定義済みの型 (*int* など) を示すキーワードである場合は、このようなキーワードがそれ自体で式になることはないため、先に示した 4 つの形式はすべて *cast-expression* になります。

### 7.7.7 Await 式

`await` 演算子は、オペランドによって表される非同期操作が完了するまで外側の非同期関数の評価を中断するために使用します。

*await-expression*:  
    `await` *unary-expression*

*await-expression* は、非同期関数 (エラー! 参照元が見つかりません。 を参照) の本体内でのみ使用できます。*await-expression* はすぐ外側にある非同期関数内の次の場所では発生しません。

- 入れ子になった (非同期以外) 匿名関数内
- *try-statement* の `catch` ブロックまたは `finally` ブロック内
- *lock-statement* のブロック内
- `unsafe` コンテキスト内

*await-expression* は非同期以外のラムダ式を使用するために構文的に変換されるので、*query-expression* 内のほとんどの場所で発生しません。

非同期関数内では、`await` を識別子として使用することはできません。そのため、*await-expressions* と、識別子を含むさまざまな式の間に構文のあいまいさはありません。非同期関数の外では、`await` は通常の識別子として動作します。

*await-expression* のオペランドは、タスクと呼ばれます。このオペランドは非同期操作を表し、この操作は *await-expression* の評価時に完了している場合も、完了していない場合もあります。`await` 演算子の目的は、待機中のタスクが完了するまで外側の非同期関数の実行を中断し、その後結果を取得することです。

### 7.7.7.1 待機可能な式

`await` 式のタスクは、**待機可能**である必要があります。式  $t$  は、次のいずれかに該当する場合、待機可能です。

- $t$  is of compile time type `dynamic`
- $t$  has an accessible instance or extension method called `GetAwaiter` with no parameters and no type parameters, and a return type  $A$  for which all of the following hold:
  - $A$  がインターフェイス `System.Runtime.CompilerServices.INotifyCompletion` (説明を簡潔にするため、これ以降は `INotifyCompletion` と呼びます) を実装している
  - $A$  にアクセス可能、読み取り可能な `bool` 型のインスタンス プロパティ `IsCompleted` がある
  - $A$  にパラメーターも型パラメーターもないアクセス可能なインスタンス メソッド `GetResult` がある

`GetAwaiter` メソッドの目的は、タスクの **awaiter** を取得することです。型  $A$  は、`await` 式の **awaiter 型** と呼ばれます。

`IsCompleted` プロパティの目的は、タスクが完了しているかどうかを判断することです。完了している場合、評価を中断する必要はありません。

`INotifyCompletion.OnCompleted` メソッドの目的は、タスクへの "継続"、つまり、タスクの完了後に呼び出される、`System.Action` 型のデリゲートを登録することです。

`GetResult` メソッドの目的は、タスクが完了したらその結果を取得することです。この結果は、結果値がある可能性がある正常な完了である場合も、`GetResult` メソッドによってスローされる例外である場合もあります。

### 7.7.7.2 `await` 式の分類

式 `await t` は、式 `(t).GetAwaiter().GetResult()` と同様に分類されます。そのため、戻り値の型 `GetResult` が `void` の場合、`await-expression` は "なし" として分類されます。`void` 以外の戻り値の型  $T$  を持つ場合、`await-expression` は型  $T$  の値として分類されます。

### 7.7.7.3 `await` 式の実行時評価

実行時に、式 `await t` は次のように評価されます。

- `awaiter a` は、式 `(t).GetAwaiter()` を評価することによって取得されます。
- `bool b` は、式 `(a).IsCompleted` を評価することによって取得されます。
- `b` が `false` である場合、評価は `a` がインターフェイス `System.Runtime.CompilerServices.ICriticalNotifyCompletion` (説明を簡潔にするため、これ以降は `ICriticalNotifyCompletion` と呼びます) を実装するかどうかによって異なります。このチェックはバインディング時に行われます。つまり、`a` がコンパイル時型 `dynamic` を持っている場合は実行時に、それ以外の場合はコンパイル時に行われます。`r` で再開デリゲート (エラー! 参照元が見つかりません。を参照) を指定するとします。
  - `a` が `ICriticalNotifyCompletion` を実装しない場合、式 `((a) as INotifyCompletion).OnCompleted(r)` が評価されます。

- *a* が `ICriticalNotifyCompletion` を実装する場合、式 `((a) as ICriticalNotifyCompletion).UnsafeOnCompleted(r)` が評価されます。
- その後、評価が中断され、制御は非同期関数の現在の呼び出し元に戻されます。
- 直後 (*b* が `true` であった場合)、または後で再開デリゲートが呼び出されるとき (*b* が `false` であった場合) に式 `(a).GetResult()` が評価されます。値が返される場合、その値は `await-expression` の結果です。それ以外の場合、結果は "なし" です。

インターフェイス メソッドである `INotifyCompletion.OnCompleted` メソッドと `ICriticalNotifyCompletion.UnsafeOnCompleted` メソッドの awriter の実装によりデリゲート *r* が呼び出されるのは一度だけです。それ以外の場合、外側の非同期関数の動作は定義されていません。

## 7.8 算術演算子

\*、/、%、+、および - の各演算子は、算術演算子と呼ばれます。

*multiplicative-expression:*

*unary-expression*

*multiplicative-expression* \* *unary-expression*

*multiplicative-expression* / *unary-expression*

*multiplicative-expression* % *unary-expression*

*additive-expression:*

*multiplicative-expression*

*additive-expression* + *multiplicative-expression*

*additive-expression* - *multiplicative-expression*

算術演算子のオペランドのコンパイル時の型が `dynamic` の場合、式は動的にバインドされます (7.2.2 を参照)。この場合、式のコンパイル時の型は `dynamic` であり、以下で説明する解決は、コンパイル時の型が `dynamic` であるオペランドの実行時の型を使用して実行時に行われます。

### 7.8.1 乗算演算子

`x * y` 形式の演算では、二項演算子のオーバーロードの解決 (7.3.4 を参照) を適用して、演算子の実装が選択されます。オペランドが、選択した演算子のパラメーターの型に変換され、その結果の型が演算子の戻り値の型になります。

定義済みの乗算演算子は以下のとおりです。これらの演算子はすべて、`x` と `y` の積を計算します。

- 整数の乗算 :

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
```

`checked` コンテキストでは、積が結果の型の範囲を超える場合は、`System.OverflowException` がスローされます。`unchecked` コンテキストでは、オーバーフローは報告されず、結果の型の範囲外の有意の上位ビットが破棄されます。

- 浮動小数点数の乗算 :

```
float operator *(float x, float y);
double operator *(double x, double y);
```

積は、IEEE 754 の演算規則に従って計算されます。次に示す表は、非ゼロ有限値、ゼロ、無限大、および NaN の可能なすべての組み合わせの結果をまとめたものです。表の `x` および `y` は正の有限

値です。 $z$  は  $x * y$  の結果です。結果を格納する型に対して結果が大きすぎる場合、 $z$  は無限大になります。結果を格納する型に対して結果が小さすぎる場合、 $z$  はゼロになります。

	+y	-y	+0	-0	$+\infty$	$-\infty$	NaN
+x	+z	-z	+0	-0	$+\infty$	$-\infty$	NaN
-x	-z	+z	-0	+0	$-\infty$	$+\infty$	NaN
+0	+0	-0	+0	-0	NaN	NaN	NaN
-0	-0	+0	-0	+0	NaN	NaN	NaN
$+\infty$	$+\infty$	$-\infty$	NaN	NaN	$+\infty$	$-\infty$	NaN
$-\infty$	$-\infty$	$+\infty$	NaN	NaN	$-\infty$	$+\infty$	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 10進数の乗算：

```
decimal operator *(decimal x, decimal y);
```

結果値が大きすぎて `decimal` 形式で表すことができない場合は、`System.OverflowException` がスローされます。値が小さすぎて `decimal` 形式で表すことができない場合、結果は 0 になります。丸めを行う前の結果のスケールは、2つのオペランドのスケールの合計です。

10進数の乗算は、`System.Decimal` 型の乗算演算子を使う場合と等価です。

## 7.8.2 除算演算子

$x / y$  形式の演算では、二項演算子のオーバーロードの解決 (7.3.4 を参照) を適用して、演算子の実装が選択されます。オペランドが、選択した演算子のパラメーターの型に変換され、その結果の型が演算子の戻り値の型になります。

定義済みの除算演算子は以下のとおりです。これらの演算子はすべて、 $x$  と  $y$  の商を計算します。

- 整数の除算：

```
int operator /(int x, int y);
uint operator /(uint x, uint y);
long operator /(long x, long y);
ulong operator /(ulong x, ulong y);
```

右辺オペランドの値が 0 の場合は、`System.DivideByZeroException` がスローされます。

除算の結果は 0 に向かって丸められます。したがって、結果の絶対値は、2つのオペランドの商の絶対値を超えない最大値となります。2つのオペランドが同じ符号の場合の結果はゼロまたは正で、2つのオペランドが異なる符号の場合の結果はゼロまたは負です。

左オペランドが `int` または `long` の表現可能な最小値で右オペランドが -1 の場合は、オーバーフローが発生します。`checked` コンテキストでは、これによって `System.ArithmeticException` (またはコンテキストでのサブクラス) がスローされます。`unchecked` コンテキストでは、`System.ArithmeticException` (またはコンテキストでのサブクラス) がスローされるか、または左のオペランドを結果値としてオーバーフローを報告しないかは、実装で定義されます。

- 浮動小数点数の除算：

```
float operator /(float x, float y);
double operator /(double x, double y);
```

商は、IEEE 754 の演算規則に従って計算されます。次に示す表は、非ゼロ有限値、ゼロ、無限大、および NaN の可能なすべての組み合わせの結果をまとめたものです。表の x および y は正の有限値です。z は  $x / y$  の結果です。結果を格納する型に対して結果が大きすぎる場合、z は無限大になります。結果を格納する型に対して結果が小さすぎる場合、z はゼロになります。

	+y	-y	+0	-0	$+\infty$	$-\infty$	NaN
+x	+z	-z	$+\infty$	$-\infty$	+0	-0	NaN
-x	-z	+z	$-\infty$	$+\infty$	-0	+0	NaN
+0	+0	-0	NaN	NaN	+0	-0	NaN
-0	-0	+0	NaN	NaN	-0	+0	NaN
$+\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	NaN	NaN	NaN
$-\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$	NaN	NaN	NaN
NaN	NaN						

- 10 進数の除算 :

```
decimal operator /(decimal x, decimal y);
```

右辺オペランドの値が 0 の場合は、`System.DivideByZeroException` がスローされます。結果値が大きすぎて `decimal` 形式で表すことができない場合は、`System.OverflowException` がスローされます。値が小さすぎて `decimal` 形式で表すことができない場合、結果は 0 になります。結果のスケールは、実際の計算結果に最も近い表現可能な 10 進値を保持する最小のスケールです。

10 進数の除算は、`System.Decimal` 型の除算演算子を使う場合と等価です。

### 7.8.3 剰余演算子

$x \% y$  形式の演算では、二項演算子のオーバーロードの解決 (7.3.4 を参照) を適用して、演算子の実装が選択されます。オペランドが、選択した演算子のパラメーターの型に変換され、その結果の型が演算子の戻り値の型になります。

定義済みの剰余演算子は以下のとおりです。これらの演算子はすべて、x と y の除算の余りを計算します。

- 整数の剰余 :

```
int operator %(int x, int y);
uint operator %(uint x, uint y);
long operator %(long x, long y);
ulong operator %(ulong x, ulong y);
```

$x \% y$  の結果は、 $x - (x / y) * y$  によって得られる値です。y がゼロの場合、`System.DivideByZeroException` がスローされます。

左オペランドが `int` または `long` の最小値で右オペランドが `-1` の場合は `System.OverflowException` がスローされます。 $x / y$  が例外をスローしない場合に  $x \% y$  が例外をスローすることはありません。

- 浮動小数点数の剰余 :

```
float operator %(float x, float y);
double operator %(double x, double y);
```

次に示す表は、非ゼロ有限値、ゼロ、無限大、およびNaNの可能なすべての組み合わせの結果をまとめたものです。xおよびyは正の有限値です。zは $x \% y$ の結果で、 $x - n * y$ を計算して得られます。nは、 $x / y$ 以下または $x/y$ に等しい最大の整数です。この剰余計算方法は、整数オペランドに対して使用されるものとほぼ同じですが、IEEE 754の定義(nは $x / y$ に最も近い整数)とは異なっています。

	+y	-y	+0	-0	$+\infty$	$-\infty$	NaN
+x	+z	+z	NaN	NaN	x	x	NaN
-x	-z	-z	NaN	NaN	-x	-x	NaN
+0	+0	+0	NaN	NaN	+0	+0	NaN
-0	-0	-0	NaN	NaN	-0	-0	NaN
$+\infty$	NaN	NaN	NaN	NaN	NaN	NaN	NaN
$-\infty$	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 10進数の剰余 :

```
decimal operator %(decimal x, decimal y);
```

右辺オペランドの値が0の場合は、`System.DivideByZeroException`がスローされます。丸めを行う前の結果のスケールは、2つのオペランドのうち大きい方のスケールになり、結果の符号は、ゼロ以外の場合はxと同じです。

10進数の剰余は、`System.Decimal`型の剰余演算子を使う場合と等価です。

#### 7.8.4 加算演算子

$x + y$ 形式の演算では、二項演算子のオーバーロードの解決(7.3.4を参照)を適用して、演算子の実装が選択されます。オペランドが、選択した演算子のパラメーターの型に変換され、その結果の型が演算子の戻り値の型になります。

定義済みの加算演算子は以下のとおりです。数値型と列挙型については、定義済み加算演算子は2つのオペランドの和を計算します。オペランドの片方または両方が文字列型の場合は、オペランドの文字列表現を連結します。

- 整数の加算 :

```
int operator +(int x, int y);
uint operator +(uint x, uint y);
long operator +(long x, long y);
ulong operator +(ulong x, ulong y);
```

`checked`コンテキストでは、和が結果の型の範囲を超える場合は、`System.OverflowException`がスローされます。`unchecked`コンテキストでは、オーバーフローは報告されず、結果の型の範囲外の有意の上位ビットが破棄されます。

- 浮動小数点数の加算 :

```
float operator +(float x, float y);
double operator +(double x, double y);
```

和は、IEEE 754 の演算規則に従って計算されます。次に示す表は、非ゼロ有限値、ゼロ、無限大、および NaN の可能なすべての組み合わせの結果をまとめたものです。表の x および y はゼロではない有限値で、z は  $x + y$  の結果です。x と y の絶対値が同じで符号だけ異なる場合、z は正のゼロになります。結果を格納する型に対して  $x + y$  が大きすぎる場合、z は  $x + y$  と同じ符号の無限大になります。

	y	+0	-0	$+\infty$	$-\infty$	NaN
x	z	x	x	$+\infty$	$-\infty$	NaN
+0	y	+0	+0	$+\infty$	$-\infty$	NaN
-0	y	+0	-0	$+\infty$	$-\infty$	NaN
$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN	NaN
$-\infty$	$-\infty$	$-\infty$	$-\infty$	NaN	$-\infty$	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 10進数の加算 :

```
decimal operator +(decimal x, decimal y);
```

結果値が大きすぎて decimal 形式で表すことができない場合は、System.OverflowException がスローされます。丸めを行う前の結果のスケールは、2つのオペランドのうち大きい方のスケールになります。

10進数の加算は、System.Decimal 型の加算演算子を使う場合と等価です。

- 列挙型の加算。すべての列挙型には、暗黙で、次の定義済み演算子が用意されています。E は列挙型で、U は E の基になっている型です。

```
E operator +(E x, U y);
E operator +(U x, E y);
```

実行時には、演算子は  $(E)((U)x + (U)y)$  とまったく同様に評価されます。

- 文字列の連結 :

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

二項演算子 + のこれらのオーバーロードでは、文字列の連結が実行されます。文字列連結のオペランドが null の場合は、空の文字列に置き換えられます。文字列ではない引数は、object 型から継承された仮想の ToString メソッドを呼び出すことで、文字列形式に変換されます。

ToString が null を返す場合は、空の文字列に置き換えられます。

```
using System;
```

```

class Test
{
    static void Main() {
        string s = null;
        Console.WriteLine("s = >" + s + "<");           // displays s = ><
        int i = 1;
        Console.WriteLine("i = " + i);                     // displays i = 1
        float f = 1.2300E+15F;
        Console.WriteLine("f = " + f);                     // displays f = 1.23E+15
        decimal d = 2.900m;
        Console.WriteLine("d = " + d);                     // displays d = 2.900
    }
}

```

文字列連結演算子の結果は、左オペランドの文字の後に右オペランドの文字が付加された文字列です。文字列連結演算子から `null` 値が返ることはできません。結果の文字列を割り当てるのに十分なメモリがない場合は、`System.OutOfMemoryException` がスローされる場合があります。

- デリゲートの組み合わせ。すべてのデリゲート型には、暗黙で、次の定義済み演算子が用意されています。`D` はデリゲート型です。

```
D operator +(D x, D y);
```

二項演算子 `+` は、両方のオペランドがデリゲート型 `D` のときは、デリゲートの組み合わせを実行します。オペランドが異なるデリゲート型の場合は、バインディング エラーになります。最初のオペランドが `null` の場合、演算の結果は、2 番目のオペランドの値(2 番目のオペランドも `null` である場合でも)になります。2 番目のオペランドが `null` の場合は、1 番目のオペランドの値になります。それ以外の場合は、演算の結果は新しいデリゲートインスタンスになり、このインスタンスを呼び出すと、1 番目のオペランドが呼び出された後、2 番目のオペランドが呼び出されます。デリゲートの組み合わせの例は、7.8.5 と 15.4 を参照してください。`System.Delegate` はデリゲート型でないため、`operator +` は定義されていません。

## 7.8.5 減算演算子

`x - y` 形式の演算では、二項演算子のオーバーロードの解決(7.3.4 を参照)を適用して、演算子の実装が選択されます。オペランドが、選択した演算子のパラメーターの型に変換され、その結果の型が演算子の戻り値の型になります。

定義済みの減算演算子は以下のとおりです。演算子はすべて、`x` から `y` を引きます。

- 整数の減算：

```

int operator -(int x, int y);
uint operator -(uint x, uint y);
long operator -(long x, long y);
ulong operator -(ulong x, ulong y);

```

`checked` コンテキストでは、差が結果の型の範囲を超える場合は、`System.OverflowException` がスローされます。`unchecked` コンテキストでは、オーバーフローは報告されず、結果の型の範囲外の有意の上位ビットが破棄されます。

- 浮動小数点数の減算：

```

float operator -(float x, float y);
double operator -(double x, double y);

```

差は、IEEE 754 の演算規則に従って計算されます。次に示す表は、非ゼロ有限値、ゼロ、無限大、および NaN の可能なすべての組み合わせの結果をまとめたものです。表の  $x$  および  $y$  はゼロではない有限値で、 $z$  は  $x - y$  の結果です。 $x$  と  $y$  が等しい場合、 $z$  は正のゼロです。結果を格納する型に対して  $x - y$  が大きすぎる場合、 $z$  は  $x - y$  と同じ符号の無限大になります。

	$y$	+0	-0	$+\infty$	$-\infty$	NaN
$x$	$z$	$x$	$x$	$-\infty$	$+\infty$	NaN
+0	$-y$	+0	+0	$-\infty$	$+\infty$	NaN
-0	$-y$	-0	+0	$-\infty$	$+\infty$	NaN
$+\infty$	$+\infty$	$+\infty$	$+\infty$	NaN	$+\infty$	NaN
$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- 10進数の減算：

```
decimal operator -(decimal x, decimal y);
```

結果値が大きすぎて decimal 形式で表すことができない場合は、System.OverflowException がスローされます。丸めを行う前の結果のスケールは、2つのオペランドのうち大きい方のスケールになります。

10進数の減算は、System.Decimal 型の減算演算子を使う場合と等価です。

- 列挙型の減算すべての列挙型には、暗黙で、次の定義済み演算子が用意されています。E は列挙型で、U は E の基になっている型です。

```
U operator -(E x, E y);
```

この演算子は、 $(U)((U)x - (U)y)$  とまったく同様に評価されます。つまり、 $x$  と  $y$  の序数値の間の差を計算し、結果の型は列挙型の基になっている型です。

```
E operator -(E x, U y);
```

この演算子は、 $(E)((U)x - y)$  とまったく同様に評価されます。つまり、列挙型の基になっている型から値を引き、列挙型の値を生成します。

- デリゲートの削除。すべてのデリゲート型には、暗黙で、次の定義済み演算子が用意されています。D はデリゲート型です。

```
D operator -(D x, D y);
```

二項演算子 - は、両方のオペランドがデリゲート型 D のときは、デリゲートの削除を実行します。オペランドが異なるデリゲート型の場合は、バインディング エラーになります。第1オペランドが null の場合、演算の結果は null になります。2番目のオペランドが null の場合は、1番目のオペランドの値になります。それ以外の場合は、どちらのオペランドも1つ以上のエントリを含む呼び出しリスト (15.1 を参照) を表し、2番目のオペランドのリストが最初のオペランドのリストに基づく連続するサブリストであれば、結果は最初のオペランドのエントリから 2番目のオペランドのエントリを除いた新しい呼び出しリストです。サブリストが等しいかどうかを判断するには、対応するエントリをデリゲート等値演算子 (7.10.8 を参照) で比較します。第2オペランドのリストが1番目のリストのサブセットになっていない場合は、結果は左オペランドの値にな

ります。どちらのオペランドのリストも、処理によって変化することはありません。第2オペランドのリストが、第1オペランドのリストの連続するエントリの複数のサブリストと一致する場合は、連続するエントリの一致するサブリストのうち右端にあるものが削除されます。削除した結果が空のリストになる場合は、結果は `null` です。次に例を示します。

```

delegate void D(int x);

class C
{
    public static void M1(int i) { /* ... */ }
    public static void M2(int i) { /* ... */ }
}

class Test
{
    static void Main()
    {
        D cd1 = new D(C.M1);
        D cd2 = new D(C.M2);
        D cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
        cd3 -= cd1;                         // => M1 + M2 + M2
        cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd2;                 // => M2 + M1
        cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd2;                 // => M1 + M1
        cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd1;                 // => M1 + M2
        cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd1;                 // => M1 + M2 + M2 + M1
    }
}

```

## 7.9 シフト演算子

ビットシフト演算を行うには、`<<` 演算子と `>>` 演算子を使用します。

```

shift-expression:
  additive-expression
  shift-expression << additive-expression
  shift-expression right-shift additive-expression

```

*shift-expression* のオペランドのコンパイル時の型が `dynamic` の場合、式は動的にバインドされます (7.2.2 を参照)。この場合、式のコンパイル時の型は `dynamic` であり、以下で説明する解決は、コンパイル時の型が `dynamic` であるオペランドの実行時の型を使用して実行時に行われます。

`x << count` または `x >> count` 形式の演算では、二項演算子のオーバーロードの解決 (7.3.4 を参照) を適用して、演算子の実装が選択されます。オペランドが、選択した演算子のパラメーターの型に変換され、その結果の型が演算子の戻り値の型になります。

オーバーロードした `shift` 演算子を宣言するときは、常に、第1オペランドの型は演算子の宣言を含むクラスまたは構造体にし、第2オペランドの型は `int` にする必要があります。

定義済みのシフト演算子は次のとおりです。

- 左シフト :

```
int operator <<(int x, int count);
uint operator <<(uint x, int count);
long operator <<(long x, int count);
ulong operator <<(ulong x, int count);
```

<< 演算子は、後で説明する方法によって計算されたビット数だけ x を左にシフトします。

x の結果の型の範囲外の上位ビットは破棄され、残りのビットが左にシフトされて、空の下位ビット位置には 0 が設定されます。

- 右シフト :

```
int operator >>(int x, int count);
uint operator >>(uint x, int count);
long operator >>(long x, int count);
ulong operator >>(ulong x, int count);
```

>> 演算子は、後で説明する方法によって計算されたビット数だけ x を右にシフトします。

x が int 型または long 型の場合は、x の下位ビットが破棄され、残りのビットが右にシフトされ、空の上位ビット位置には、x が負ではない場合は 0 が、x が負の場合は 1 が設定されます。

x が uint 型または ulong 型の場合は、x の下位ビットが破棄され、残りのビットが右にシフトされて、空の上位ビット位置には 0 が設定されます。

定義済み演算子では、シフトするビットの数は次の方法で計算されます。

- x の型が int または uint のときは、count の下位 5 ビットがシフト カウントになります。つまり、シフト カウントは count & 0x1F で計算されます。
- x の型が long または ulong のときは、count の下位 6 ビットがシフト カウントになります。つまり、シフト カウントは count & 0x3F で計算されます。

得られたシフト カウントが 0 の場合は、シフト演算からは単に x の値が返ります。

シフト演算ではオーバーフローが発生することではなく、checked コンテキストと unchecked コンテキストで同じ結果になります。

>> 演算子の左オペランドが符号付き整数型の場合は、算術右シフトが行われ、オペランドの最上位ビット(符号ビット)の値が上位の空ビット位置に設定されます。>> 演算子の左オペランドが符号なし整数型の場合は、論理右シフトが行われ、上位の空ビット位置には常に 0 が設定されます。オペランドの型から推測される演算とは逆の演算を実行するには、明示的なキャストを使用できます。たとえば、x が int 型の変数の場合、unchecked((int)((uint)x >> y)) という演算では x の論理右シフトが行われます。

## 7.10 関係演算子と型検査演算子

==、!=、<、>、<=、>=、is、および as の各演算子は、関係演算子および型検査演算子と呼ばれます。

*relational-expression:*

```
shift-expression
relational-expression < shift-expression
relational-expression > shift-expression
relational-expression <= shift-expression
relational-expression >= shift-expression
relational-expression is type
relational-expression as type
```

*equality-expression:*

```
relational-expression
equality-expression == relational-expression
equality-expression != relational-expression
```

**is** 演算子は 7.10.10 で、**as** 演算子は 7.10.11 で説明しています。

**==**、**!=**、**<**、**>**、**<=**、および**>=** の各演算子は、"比較演算子" と呼ばれます。

比較演算子のオペランドのコンパイル時の型が **dynamic** の場合、式は動的にバインドされます (7.2.2 を参照)。この場合、式のコンパイル時の型は **dynamic** であり、以下で説明する解決は、コンパイル時の型が **dynamic** であるオペランドの実行時の型を使用して実行時に行われます。

**x op y** 形式 (*op* は比較演算子) の演算では、オーバーロードの解決 (7.3.4 を参照) を適用して、演算子の実装が選択されます。オペランドが、選択した演算子のパラメーターの型に変換され、その結果の型が演算子の戻り値の型になります。

以下のセクションでは、定義済みの比較演算子について説明します。次の表で示すように、定義済み比較演算子はすべて、**bool** 型の結果を返します。

演算	結果
<b>x == y</b>	<b>x</b> と <b>y</b> が等しい場合は <b>true</b> 。それ以外の場合は <b>false</b> 。
<b>x != y</b>	<b>x</b> と <b>y</b> が等しくない場合は <b>true</b> 。それ以外の場合は <b>false</b> 。
<b>x &lt; y</b>	<b>x</b> が <b>y</b> より小さい場合は <b>true</b> 、それ以外の場合は <b>false</b> 。
<b>x &gt; y</b>	<b>x</b> が <b>y</b> より大きい場合は <b>true</b> 、それ以外の場合は <b>false</b> 。
<b>x &lt;= y</b>	<b>x</b> が <b>y</b> 以下である場合は <b>true</b> 、それ以外の場合は <b>false</b> 。
<b>x &gt;= y</b>	<b>x</b> が <b>y</b> 以上である場合は <b>true</b> 、それ以外の場合は <b>false</b> 。

### 7.10.1 整数比較演算子

定義済みの整数比較演算子は次のとおりです。

```
bool operator ==(int x, int y);
bool operator ==(uint x, uint y);
bool operator ==(long x, long y);
bool operator ==(ulong x, ulong y);

bool operator !=(int x, int y);
bool operator !=(uint x, uint y);
bool operator !=(long x, long y);
bool operator !=(ulong x, ulong y);
```

```

bool operator <(int x, int y);
bool operator <(uint x, uint y);
bool operator <(long x, long y);
bool operator <(ulong x, ulong y);

bool operator >(int x, int y);
bool operator >(uint x, uint y);
bool operator >(long x, long y);
bool operator >(ulong x, ulong y);

bool operator <=(int x, int y);
bool operator <=(uint x, uint y);
bool operator <=(long x, long y);
bool operator <=(ulong x, ulong y);

bool operator >=(int x, int y);
bool operator >=(uint x, uint y);
bool operator >=(long x, long y);
bool operator >=(ulong x, ulong y);

```

これらの演算子は、2つの整数オペランドの数値を比較し、特定の関係が `true` または `false` のいずれであるかを示す `bool` 値を返します。

### 7.10.2 浮動小数点数比較演算子

定義済みの浮動小数点数比較演算子は次のとおりです。

```

bool operator ==(float x, float y);
bool operator ==(double x, double y);

bool operator !=(float x, float y);
bool operator !=(double x, double y);

bool operator <(float x, float y);
bool operator <(double x, double y);

bool operator >(float x, float y);
bool operator >(double x, double y);

bool operator <=(float x, float y);
bool operator <=(double x, double y);

bool operator >=(float x, float y);
bool operator >=(double x, double y);

```

これらの演算子は、IEEE 754 標準の規則に従ってオペランドを比較します。

- いずれかのオペランドが `Nan` の場合は、`!=` を除くすべての演算子の結果が `false` になります。`!=` の場合だけ、結果が `true` になります。任意の2つのオペランドに対して、`x != y` は常に `!(x == y)` と同じ結果を生成します。ただし、一方または両方のオペランドが `Nan` の場合、`<`、`>`、`<=`、および`>=` の各演算子は、逆の意味を持つ演算子の論理否定と同じ結果にはなりません。たとえば、`x` と `y` のいずれかが `Nan` の場合、`x < y` は `false` ですが、`!(x >= y)` は `true` です。
- オペランドのいずれも `Nan` でないときは、演算子は、2つの浮動小数点数オペランドの値を次の順序で比較します。

`-∞ < -max < ... < -min < -0.0 == +0.0 < +min < ... < +max < +∞`

`min` および `max` は、その浮動小数点形式で表現できる最小および最大の正の有限値です。この順序では次の点に注意してください。

- 負のゼロと正のゼロは等しいものと見なされます。

- 負の無限大は他のすべての値より小さいものと見なされますが、他の負の無限大とは等しいものと見なされます。
- 正の無限大は他のすべての値より大きいものと見なされますが、他の正の無限大とは等しいものと見なされます。

### 7.10.3 10 進数比較演算子

定義済みの 10 進数比較演算子は次のとおりです。

```
bool operator ==(decimal x, decimal y);
bool operator !=(decimal x, decimal y);
bool operator <(decimal x, decimal y);
bool operator >(decimal x, decimal y);
bool operator <=(decimal x, decimal y);
bool operator >=(decimal x, decimal y);
```

これらの演算子は、2つの 10 進数オペランドの数値を比較し、特定の関係が `true` または `false` のいずれであるかを示す `bool` 値を返します。10 進数の各比較は、`System.Decimal` 型の対応する関係演算子または等値演算子を使う場合と等価です。

### 7.10.4 ブール等値演算子

定義済みのブール等値演算子は次のとおりです。

```
bool operator ==(bool x, bool y);
bool operator !=(bool x, bool y);
```

`x` と `y` が両方とも `true` の場合、または `x` と `y` が両方とも `false` の場合、`==` の結果は `true` になります。それ以外の場合、結果は `false` になります。

`x` と `y` が両方とも `true` の場合、または `x` と `y` が両方とも `false` の場合、`!=` の結果は `false` になります。それ以外の場合、結果は `true` になります。オペランドが `bool` 型のとき、`!=` 演算子の結果は `^` 演算子と同じになります。

### 7.10.5 列挙比較演算子

すべての列挙型には、暗黙で、次に示す定義済み比較演算子が用意されています。

```
bool operator ==(E x, E y);
bool operator !=(E x, E y);
bool operator <(E x, E y);
bool operator >(E x, E y);
bool operator <=(E x, E y);
bool operator >=(E x, E y);
```

`x op y` を評価した結果は、`((U)x) op ((U)y)` を評価した結果とまったく同じです。ここで、`x` と `y` は、`U` 型が基になった列挙型 `E` の式で、`op` は、比較演算子のいずれかです。つまり、列挙型の比較演算子は、2つのオペランドの基になっている整数値を比較しているだけです。

### 7.10.6 参照型等値演算子

定義済みの参照型等値演算子は次のとおりです。

```
bool operator ==(object x, object y);
```

```
bool operator !=(object x, object y);
```

これらの演算子は、2つの参照が等値かどうかを比較した結果を返します。

定義済みの参照型等値演算子は、**object** 型のオペランドを受け付けるため、適切な **operator ==** メンバーおよび **operator !=** メンバーを宣言していないすべての型に適用されます。逆に、適切なユーザー定義等値演算子を宣言することで、定義済みの参照型等値演算子を完全に隠ぺいできます。

定義済みの参照型等値演算子には、次のいずれかが必要です。

- オペランドは両方とも型が *reference-type* とわかっている値か、リテラル **null** 値です。さらに、一方のオペランドの型から他方のオペランドの型への明示的な参照の変換 (6.2.4 を参照) が存在します。
- 1つのオペランドは **T** 型の値で、**T** は *type-parameter* を、もう一方のオペランドはリテラル **null** 値を示します。また、**T** には値型の制約がありません。

これらの条件の一方でも満たされていない限り、バインディング エラーが発生します。これらの規則によって、次のような重要な影響があります。

- 定義済み参照型等値演算子を使って、バインディング時に異なることがわかっている 2つの参照を比較すると、バインディング エラーになります。たとえば、オペランドのバインディング時の型が 2つのクラス型 **A** および **B** で、**A** も **B** も他方から派生していない場合は、2つのオペランドが同じオブジェクトを参照することはできません。したがって、演算はバインディング エラーと見なされます。
- 定義済み参照型等値演算子では、値型のオペランドを比較することはできません。したがって、構造型の場合は、独自の等値演算子を宣言しない限り、その構造型の値を比較できません。
- 定義済み参照型等値演算子では、ボックス化演算をオペランドで使用できません。新しく割り当てられたボックス化されたインスタンスへの参照は、必然的に他のすべての参照と異なっているため、そのようなボックス化演算を実行しても意味がありません。
- 型パラメーターの **T** 型のオペランドを **null** と比較して、**T** の実行時の型が値型だった場合、比較の結果は **false** になります。

次の例は、制約されていない型パラメーター型の引数が **null** かどうかをチェックします。

```
class C<T>
{
    void F(T x) {
        if (x == null) throw new ArgumentNullException();
        ...
    }
}
```

**x == null** という構造は **T** が値型を表す場合でも使用でき、**T** が値型のときは単に **false** という結果が定義されます。

**x == y** または **x != y** の形式の演算では、適用可能な **operator ==** または **operator !=** が存在していると、演算子オーバーロードの解決 (7.3.4) の規則において、定義済みの参照型等値演算子の代わりにその演算子が選択されます。ただし、一方または両方のオペランドを **object** 型に明示的にキャストすることで、常に定義済み参照型等値演算子を選択できます。次の例を参照してください。

```
using System;
```

```

class Test
{
    static void Main() {
        string s = "Test";
        string t = string.Copy(s);
        Console.WriteLine(s == t);
        Console.WriteLine((object)s == t);
        Console.WriteLine(s == (object)t);
        Console.WriteLine((object)s == (object)t);
    }
}

```

この例では、次のように出力されます。

```

True
False
False
False

```

変数の `s` と `t` は、同じ文字列を含む 2 つの異なる `string` インスタンスを参照しています。両方のオペランドが `True` 型のときは、定義済みの文字列等値演算子 (7.10.7 を参照) が選択されるため、最初の比較では `string` が output されます。一方または両方のオペランドが `object` 型のときは、定義済みの参照型等値演算子が選択されるため、残りの比較ではすべて `False` が output されます。

上の技法は値型では意味がないことに注意してください。次の例を参照してください。

```

class Test
{
    static void Main() {
        int i = 123;
        int j = 123;
        System.Console.WriteLine((object)i == (object)j);
    }
}

```

この例では、キャストによって、ボックス化された `int` 値の 2 つの異なるインスタンスに対する参照が作成されるため、`False` が output されます。

## 7.10.7 文字列等値演算子

定義済みの文字列等値演算子は次のとおりです。

```

bool operator ==(string x, string y);
bool operator !=(string x, string y);

```

次のいずれかの条件が満たされていると、2 つの `string` 値は等しいものと見なされます。

- 両方の値が `null` である。
- 両方の値が `null` ではなく、長さが等しくて各文字位置の文字が等しい文字列インスタンスを参照している。

文字列等値演算子は、文字列の “参照” ではなく文字列の “値” を比較します。2 つの異なる文字列インスタンスにまったく同じ文字列のシーケンスが含まれている場合、文字列の値は同じですが、参照は異なります。7.10.6 で説明したように、参照型の等値演算子は、文字列値ではなく文字列参照を比較するために使用できます。

## 7.10.8 デリゲート等値演算子

定義済みのデリゲート等値演算子は次のとおりです。

```

bool operator ==(System.Delegate x, System.Delegate y);

```

```
bool operator !=(System.Delegate x, System.Delegate y);
```

2つのデリゲートインスタンスは、次の場合に等しいと見なされます。

- どちらかのデリゲートインスタンスが `null` の場合は、両方とも `null` の場合にだけ等しいものと見なされます。
- デリゲートの実行時の型が異なる場合、デリゲートは等値ではありません。
- 両方のデリゲートインスタンスに呼び出しリスト (15.1 を参照) がある場合は、両方の呼び出しリストが同じ長さで、一方の呼び出しリストの各エントリがもう一方の呼び出しリストの同じ順序で対応するエントリと等しい (下記を参照) 場合に限り、等しいものと見なされます。

呼び出しリストのエントリの等値性は、次の規則に従います。

- 2つの呼び出しリストのエントリが両方とも同じ静的メソッドを示す場合、エントリは同じものと見なされます。
- 2つの呼び出しリストのエントリが両方とも同じ対象オブジェクト上の同じ非静的メソッドを示す場合 (参照型等値演算子で定義)、エントリは同じものと見なされます。
- キャプチャされた外部変数インスタンスの集合が同じで (多くの場合は空の集合)、意味が同じ *anonymous-function-expressions* が評価された場合、生成される呼び出しリストエントリは等値であることが許可されています (必須ではありません)。

### 7.10.9 等値演算子と `null`

`==` 演算子および `!=` 演算子では、演算を表す定義済み演算子またはユーザー一定義演算子 (リフトされていない形式またはリフト形式) が存在しない場合でも、1つのオペランドを `null` 許容型の値とし、もう1つのオペランドを `null` リテラルとすることが可能です。

次のいずれかの形式の演算があるとします。

```
x == null      null == x      x != null      null != x
```

ここで、`x` は `null` 許容型の式です。演算子のオーバーロード解決 (7.2.4 を参照) で適用可能な演算子が見つからない場合は、代わりに `x` の `HasValue` プロパティから結果が計算されます。具体的には、最初の2つの形式は `!x.HasValue` に変換され、最後の2つの形式は `x.HasValue` に変換されます。

### 7.10.10 `is` 演算子

`is` 演算子を使うと、オブジェクトの実行時の型が指定した型と互換性があるかどうかを動的に確認できます。演算 `E is T` (`E` は式で `T` は型) の結果はブール値で、参照変換、ボックス化変換、またはボックス化解除変換により `E` を `T` 型に変換できるかどうかを示しています。この演算は、すべての型パラメーターの型引数が置き換えられた後で、次のように評価されます。

- `E` が匿名関数の場合は、コンパイル エラーになります。
- `E` がメソッド グループ、または `null` リテラルであるか、`E` の型が参照型または `null` 非許容型で `E` の値が `null` だった場合、結果は `false` になります。
- それ以外の場合は、次のように、`D` は `E` の動的な型を表します。
  - `E` の型が参照型の場合、`D` は `E` によるインスタンス参照の実行時の型になります。
  - `E` の型が `null` 許容型の場合、`D` はその `null` 許容型の基になる型になります。
  - `E` の型が `null` 非許容の値型の場合、`D` は `E` の型になります。

- 処理の結果は、次のように  $D$  および  $T$  に依存します。
  - $T$  が参照型の場合、 $D$  と  $T$  が同じ型であるか、 $D$  が参照型で  $D$  から  $T$  への暗黙の参照変換が存在するか、 $D$  が値型で  $D$  から  $T$  へのボックス化変換が存在するならば、結果は `true` です。
  - $T$  が `null` 許容型の場合、 $D$  が  $T$  の基になる型であれば、結果は `true` です。
  - $T$  が `null` 非許容の値型の場合、 $D$  と  $T$  が同じ型であれば、結果は `true` です。
  - それ以外の場合、結果は `false` です。

ユーザ一定義変換は、`is` 演算子では考慮されません。

### 7.10.11 as 演算子

値を特定の参照型または `null` 許容型に明示的に変換するには、`as` 演算子を使用します。キャスト式(7.7.6 を参照)とは異なり、`as` 演算子は例外をスローしません。指定された変換を実行できない場合は、結果の値が `null` になります。

形式  $E \text{ as } T$  の演算では、 $E$  は式、 $T$  は参照型、参照型であることがわかっている型パラメーター、または `null` 許容型である必要があります。また、少なくとも次のいずれかが成り立つ必要があります。そうでない場合はコンパイルエラーになります。

- $E$  から  $T$  への変換には、恒等変換(6.1.1 を参照)、暗黙の `null` 許容変換(6.1.4 を参照)、暗黙の参照変換(6.1.6 を参照)、ボックス化変換(6.1.7 を参照)、明示的な `null` 許容変換(6.2.3 を参照)、明示的な参照変換(6.2.4 を参照)、またはボックス化解除(6.2.5 を参照)変換が存在します。
- $E$  または  $T$  の型がオープン型であること。
- $E$  が `null` リテラルであること。

$E$  のコンパイル時の型が `dynamic` ではない場合、演算  $E \text{ as } T$  の結果は次の式と同じになります。

$E \text{ is } T ? (T)(E) : (T)\text{null}$

ただし、 $E$  が評価されるのは 1 回だけです。 $E \text{ as } T$  は、動的な型チェックが最大でも 1 回だけ実行されるように、コンパイラによって最適化されます。これに対して上の展開形式では、2 回の動的な型チェックが暗黙的に実行されます。

$E$  のコンパイル時の型が `dynamic` の場合は、キャスト演算子とは異なり、`as` 演算子は動的にバインドされません(7.2.2 を参照)。そのため、この場合の展開は次のようになります。

$E \text{ is } T ? (T)(\text{object})(E) : (T)\text{null}$

ユーザ一定義変換などの一部の変換は、`as` 演算子では実行できず、キャスト式を使って行う必要があります。

次に例を示します。

```
class X
{
    public string F(object o) {
        return o as string;           // OK, string is a reference type
    }

    public T G<T>(object o) where T: Attribute {
        return o as T;              // Ok, T has a class constraint
    }
}
```

```

public U H<U>(object o) {
    return o as U;           // Error, U is unconstrained
}
}

```

`T` の `G` 型パラメーターは、クラス制約を持つため参照型であることがわかります。`H` の `U` 型パラメーターは参照型であることがわからないため、`H` では `as` 演算子が許可されません。

## 7.11 論理演算子

`&` `^` `|` の各演算子は、論理演算子と呼ばれます。

*and-expression:*

```

equality-expression
and-expression & equality-expression

```

*exclusive-or-expression:*

```

and-expression
exclusive-or-expression ^ and-expression

```

*inclusive-or-expression:*

```

exclusive-or-expression
inclusive-or-expression | exclusive-or-expression

```

論理演算子のオペランドのコンパイル時の型が `dynamic` の場合、式は動的にバインドされます (7.2.2 を参照)。この場合、式のコンパイル時の型は `dynamic` であり、以下で説明する解決は、コンパイル時の型が `dynamic` であるオペランドの実行時の型を使用して実行時に行われます。

`x op y` の形式の演算 (`op` は論理演算子のいずれか) では、オーバーロードの解決規則 (7.3.4 を参照) が適用されて、演算子の特定の実装が選択されます。オペランドが、選択した演算子のパラメーターの型に変換され、その結果の型が演算子の戻り値の型になります。

以下のセクションでは、定義済みの論理演算子について説明します。

### 7.11.1 整数論理演算子

定義済みの整数論理演算子は次のとおりです。

```

int operator &(int x, int y);
uint operator &(uint x, uint y);
long operator &(long x, long y);
ulong operator &(ulong x, ulong y);

int operator |(int x, int y);
uint operator |(uint x, uint y);
long operator |(long x, long y);
ulong operator |(ulong x, ulong y);

int operator ^(int x, int y);
uint operator ^(uint x, uint y);
long operator ^(long x, long y);
ulong operator ^(ulong x, ulong y);

```

`&` 演算子は、2つのオペランドに対してビットごとの論理 AND を計算します。`|` 演算子は、2つのオペランドに対してビットごとの論理 OR を計算します。`^` 演算子は、2つのオペランドに対してビットごとの論理排他的 OR を計算します。これらの演算ではオーバーフローは発生しません。

### 7.11.2 列挙論理演算子

すべての列挙型 `E` には、暗黙で、次に示す定義済み論理演算子が用意されています。

```
E operator &(E x, E y);
E operator |(E x, E y);
E operator ^(E x, E y);
```

$x \ op \ y$  を評価した結果は、 $(E)((U)x \ op \ (U)y)$  を評価した結果とまったく同じです。ここで、 $x$  と  $y$  は、 $U$  型が基になった列挙型  $E$  の式で、 $op$  は、論理演算子のいずれかです。つまり、列挙型の論理演算子は、2つのオペランドの基になっている型に対して論理演算を実行するだけです。

### 7.11.3 ブール論理演算子

定義済みのブール論理演算子は、次のとおりです。

```
bool operator &(bool x, bool y);
bool operator |(bool x, bool y);
bool operator ^(bool x, bool y);
```

$x$  と  $y$  の両方が `true` の場合、 $x \ & \ y$  の結果は `true` になります。それ以外の場合、結果は `false` になります。

$x$  または  $y$  のどちらかが `true` の場合、 $x \ | \ y$  の結果は `true` となります。それ以外の場合、結果は `false` になります。

$x$  が `true` で  $y$  が `false` の場合、または  $x$  が `false` で  $y$  が `true` の場合、 $x \ ^ \ y$  の結果は、`true` になります。それ以外の場合、結果は `false` になります。オペランドが `bool` 型のとき、 $\wedge$  演算子の結果は  $\neq$  演算子と同じになります。

### 7.11.4 null 許容ブール論理演算子

null 許容ブール型である `bool?` は、`true`、`false`、および `null` の3つの値を表すことができ、SQL のブール型の式で使用される3つの値で構成される型と概念的によく似ています。`bool?` に  $\&$  演算子および  $|$  演算子を使用することによって生成される結果と、SQL の3つの値で構成される論理との間で整合性を確保するために、次のような定義済み演算子があります。

```
bool? operator &(bool? x, bool? y);
bool? operator |(bool? x, bool? y);
```

次の表は、`true`、`false`、および `null` という3つの値のすべての組み合わせに対して、これらの演算子によって生成される結果を示します。

$x$	$y$	$x \ & \ y$	$x \   \ y$
<code>true</code>	<code>true</code>	<code>true</code>	<code>true</code>
<code>true</code>	<code>false</code>	<code>false</code>	<code>true</code>
<code>true</code>	<code>null</code>	<code>null</code>	<code>true</code>
<code>false</code>	<code>true</code>	<code>false</code>	<code>true</code>
<code>false</code>	<code>false</code>	<code>false</code>	<code>false</code>
<code>false</code>	<code>null</code>	<code>false</code>	<code>null</code>
<code>null</code>	<code>true</code>	<code>null</code>	<code>true</code>
<code>null</code>	<code>false</code>	<code>false</code>	<code>null</code>
<code>null</code>	<code>null</code>	<code>null</code>	<code>null</code>

## 7.12 条件論理演算子

`&&` と `||` の各演算子は、条件論理演算子と呼ばれます。"ショートサーキット" 論理演算子と呼ばれることもあります。

*conditional-and-expression:*

*inclusive-or-expression*

*conditional-and-expression && inclusive-or-expression*

*conditional-or-expression:*

*conditional-and-expression*

*conditional-or-expression || conditional-and-expression*

`&&` 演算子および `||` 演算子は、`&` 演算子および `|` 演算子の条件付きバージョンです。

- 演算 `x && y` は演算 `x & y` に対応していますが、`x` が `false` でない場合にだけ `y` が評価される点が異なります。
- 演算 `x || y` は演算 `x | y` に対応していますが、`x` が `true` でない場合にだけ `y` が評価される点が異なります。

条件論理演算子のオペランドのコンパイル時の型が `dynamic` の場合、式は動的にバインドされます (7.2.2 を参照)。この場合、式のコンパイル時の型は `dynamic` であり、以下で説明する解決は、コンパイル時の型が `dynamic` であるオペランドの実行時の型を使用して実行時に行われます。

`x && y` または `x || y` という形式の演算は、`x & y` または `x | y` という形式の演算と同じように、オーバーロードの解決規則 (7.3.4 を参照) を適用して処理されます。処理の方法は、次のとおりです。

- オーバーロードの解決によって单一の最適な演算子を見つけられない場合、またはオーバーロードの解決によって定義済み整数論理演算子の 1 つが選択された場合は、バインディング エラーになります。
- 選択された演算子が定義済みブール論理演算子 (7.11.3 を参照) の 1 つであるか、`null` 許容ブール論理演算子 (7.11.4 を参照) である場合、演算は 7.12.1 で説明されているように処理されます。
- 選択された演算子がユーザー定義演算子の場合は、演算は 7.12.2 で説明されているように処理されます。

条件論理演算子を直接オーバーロードすることはできません。ただし、条件論理演算子は標準の論理演算子によって評価されるため、若干の制限はありますが、標準の論理演算子のオーバーロードは、条件論理演算子のオーバーロードと見なすこともできます。これは 7.12.2 で詳細に説明します。

### 7.12.1 ブール条件論理演算子

`&&` または `||` のオペランドが `bool` 型のとき、または適切な `operator &` または `operator |` が定義されていなくても `bool` への暗黙の変換が定義されている型がオペランドになっているときは、演算は次のように処理されます。

- 演算 `x && y` は `x ? y : false` のように評価されます。つまり、`x` を最初に評価し、`bool` 型に変換します。`x` が `true` の場合は、`y` を評価して `bool` 型に変換し、これが演算の結果になります。それ以外の場合は、演算の結果は `false` です。
- 演算 `x || y` は `x ? true : y` のように評価されます。つまり、`x` を最初に評価し、`bool` 型に変換します。`x` が `true` の場合は、演算の結果は `true` になります。それ以外の場合は、`y` を評価して `bool` 型に変換し、これが演算の結果になります。

### 7.12.2 ユーザー定義の条件論理演算子

`&&` または `||` のオペランドが、ユーザー定義の適用可能な `operator &` または `operator |` を宣言している型の場合は、次の両方の条件が満たされている必要があります。`T` は、選択された演算子が宣言されている型です。

- 選択された演算子の戻り値の型および各パラメーターの型は、`T` である必要があります。つまり、演算子は、`T` 型の 2 つのオペランドの論理 `AND` または論理 `OR` を計算し、`T` 型の結果を返す必要があります。
  - `T` は、`operator true` および `operator false` の宣言を含んでいる必要があります。
- これらの要件のいずれかが満たされていないと、バインディング エラーになります。それ以外の場合、`&&` 演算または `||` 演算は、ユーザー定義の `operator true` または `operator false` と選択されたユーザー定義演算子を組み合わせて評価されます。
- 演算 `x && y` は、`T.false(x) ? x : T.&(x, y)` として評価されます。`T.false(x)` は `T` で宣言されている `operator false` の呼び出しで、`T.&(x, y)` は選択された `operator &` の呼び出します。つまり、`x` を最初に評価し、結果に対して `operator false` を呼び出して、`x` が確実に `false` かどうかを決定します。`x` が確実に `false` の場合は、演算の結果は `x` に対して先に計算した値です。それ以外の場合は、`y` を評価し、`x` に対して先に計算した値と `y` に対して計算した値で、選択された `operator &` を呼び出して、演算の結果を生成します。
  - 演算 `x || y` は、`T.true(x) ? x : T.|(x, y)` として評価されます。`T.true(x)` は `T` で宣言されている `operator true` の呼び出しで、`T.|(x, y)` は選択された `operator |` の呼び出します。つまり、`x` を最初に評価し、結果に対して `operator true` を呼び出して、`x` が確実に `true` かどうかを決定します。`x` が確実に `true` の場合は、演算の結果は `x` に対して先に計算した値です。それ以外の場合は、`y` を評価し、`x` に対して先に計算した値と `y` に対して計算した値で、選択された `operator |` を呼び出して、演算の結果を生成します。

いずれの演算でも、`x` によって指定される式は 1 回だけ評価され、`y` によって指定される式は、まったく評価されないか 1 回だけ評価されます。

`operator true` および `operator false` を実装する型の例については、11.4.2 を参照してください。

### 7.13 null 合体演算子

`??` 演算子は、null 合体演算子と呼ばれます。

```
null-coalescing-expression:
conditional-or-expression
conditional-or-expression ?? null-coalescing-expression
```

`a ?? b` という形式の null 合体式では、`a` は null 許容型または参照型である必要があります。`a` が非 null の場合、`a ?? b` の結果は `a` です。それ以外の場合、結果は `b` です。この演算は、`a` が null の場合にのみ `b` を評価します。

null 合体演算子の結合規則は "右から左" です。つまり、演算は右から左にグループ化されます。たとえば、`a ?? b ?? c` という形式の式は、`a ?? (b ?? c)` として評価されます。一般的には、`E1 ?? E2 ?? ... ?? EN` という形式の式は非 null の最初のオペランドを返し、すべてのオペランドが null の場合は null を返します。

式 `a ?? b` の型は、オペランドで使用できる暗黙の型変換によって決まります。優先順位に基づいて、`a ?? b` の型は  $A_0$ 、`A`、または `B` になります。ここで `A` は `a` の型 (`a` に型がある場合)、`B` は `b` の型 (`b` に型がある場合) を表します。 $A_0$  は、`A` が `null` 許容型の場合は `A` の基になる型を表し、それ以外の場合は `A` を表します。具体的に、`a ?? b` は次のように処理されます。

- `A` が存在し、`null` 許容型でない場合または参照型の場合は、コンパイルエラーが発生します。
- `b` が動的な式の場合、結果の型は `dynamic` です。実行時には、`a` が最初に評価されます。`a` が `null` でない場合は、`a` は動的に変換され、これが結果になります。そうでない場合は、`b` を評価して、これが結果になります。
- それ以外の場合、`A` が存在し、`null` 許容型で、`b` から  $A_0$  への暗黙の型変換が存在する場合、結果の型は  $A_0$  になります。実行時には、`a` が最初に評価されます。`a` が `null` でない場合は、`a` を型  $A_0$  にラップ解除し、これが結果になります。それ以外の場合は、`b` を評価して  $A_0$  型に変換し、これが結果になります。
- それ以外の場合で、`A` が存在し、`b` から `A` への暗黙の型変換が存在する場合は、結果の型は `A` になります。実行時には、`a` が最初に評価されます。`a` が `null` でない場合は、`a` が結果になります。それ以外の場合は、`b` を評価して `A` 型に変換し、これが結果になります。
- それ以外の場合、`A` が存在し、`null` 許容型で、`b` が型 `B` を持ち、 $A_0$  から `B` への暗黙の型変換が存在する場合、結果の型は `B` になります。実行時には、`a` が最初に評価されます。`a` が `null` でない場合は、`a` が型  $A_0$  にラップ解除され、型 `B` に変換されます。これが、結果になります。そうでない場合は、`b` を評価して、これが結果になります。
- それ以外の場合で、`b` が型 `B` で、`a` から `B` への暗黙の型変換が存在する場合は、結果の型は `B` になります。実行時には、`a` が最初に評価されます。`a` が `null` でない場合、`a` は型 `B` に変換され、これが結果になります。そうでない場合は、`b` を評価して、これが結果になります。
- それ以外の場合は、`a` と `b` は互換性がなく、コンパイルエラーになります。

## 7.14 条件演算子

`?:` 演算子は、条件演算子と呼ばれます。三項演算子と呼ばれることもあります。

```
conditional-expression:
    null-coalescing-expression
    null-coalescing-expression ? expression : expression
```

`b ?: x : y` の形式の条件式では、最初に条件 `b` が評価されます。`b` が `true` の場合は、`x` が評価されて、演算の結果になります。そうでない場合は、`y` が評価されて、演算の結果になります。条件式では、`x` と `y` が両方とも評価されることはありません。

条件演算子の結合規則は "右から左" です。つまり、演算は右から左にグループ化されます。たとえば、`a ?: b : c ?: d : e` という形式の式は、`a ?: b : (c ?: d : e)` として評価されます。

`?:` 演算子の最初のオペランドには、`bool` に暗黙で変換できる式、または `operator true` を実装する型の式を指定する必要があります。いずれの条件も満たされない場合は、コンパイルエラーが発生します。

`?:` 演算子の 2 番目と 3 番目のオペランド (`x` および `y`) は、条件式の型を制御します。

- `x` に型 `X` があり、`y` に型 `Y` がある場合は、次のようになります。

- $X$  から  $Y$  への暗黙の型変換 ( $O$  を参照) は存在するが、 $Y$  から  $X$  へは存在しない場合は、 $Y$  が条件式の型になります。
  - $Y$  から  $X$  への暗黙の型変換 ( $O$  を参照) は存在するが、 $X$  から  $Y$  へは存在しない場合は、 $X$  が条件式の型になります。
  - 上記以外の場合は、式の型を決定できず、コンパイル エラーになります。
  - $x$  および  $y$  の 1 つだけに型があり、 $x$  と  $y$  の両方が暗黙的にその型に変換できる場合、それが条件式の型になります。
  - 上記以外の場合は、式の型を決定できず、コンパイル エラーになります。
- $b ? x : y$  の形式による条件式の実行時の処理は、次の手順で構成されています。
- 最初に、 $b$  を評価し、 $b$  の `bool` 値を決定します。
    - $b$  の型から `bool` への暗黙の変換が存在している場合は、この暗黙の変換を実行して `bool` 値を生成します。
    - それ以外の場合は、 $b$  の型で定義されている `operator true` を呼び出して、`bool` 値を生成します。
  - 上の手順で生成した `bool` 値が `true` の場合は、 $x$  を評価して条件式の型に変換し、これが条件式の結果になります。
  - それ以外の場合は、 $y$  を評価して条件式の型に変換し、これが条件式の結果になります。

## 7.15 匿名関数の式

"匿名関数" は、"インライン" メソッド定義を表す式です。匿名関数自体は値または型を持ちませんが、互換性のあるデリゲート型または式ツリー型に変換できます。匿名関数変換の評価は、変換先の型によって異なります。デリゲート型の場合、変換は匿名関数が定義するメソッドを参照するデリゲート値を評価します。式ツリー型の場合、変換はオブジェクト構造としてのメソッドの構造を表す式ツリーを評価します。

匿名関数には、過去の経緯から、*lambda-expression* と *anonymous-method-expression* という 2 種類の構文があります。ほとんどの目的では、後方互換性のためにこの言語に残されている *anonymous-method-expression* よりも *lambda-expression* の方が簡潔で表現性に優れています。

```

lambda-expression:
  asyncopt anonymous-function-signature => anonymous-function-body

anonymous-method-expression:
  asyncopt delegate explicit-anonymous-function-signatureopt block

anonymous-function-signature:
  explicit-anonymous-function-signature
  implicit-anonymous-function-signature

explicit-anonymous-function-signature:
  ( explicit-anonymous-function-parameter-listopt )

explicit-anonymous-function-parameter-list:
  explicit-anonymous-function-parameter
  explicit-anonymous-function-parameter-list , explicit-anonymous-function-parameter

```

```

explicit-anonymous-function-parameter:
  anonymous-function-parameter-modifieropt type identifier

anonymous-function-parameter-modifier:
  ref
  out

implicit-anonymous-function-signature:
  ( implicit-anonymous-function-parameter-listopt )
  implicit-anonymous-function-parameter

implicit-anonymous-function-parameter-list:
  implicit-anonymous-function-parameter
  implicit-anonymous-function-parameter-list , implicit-anonymous-function-parameter

implicit-anonymous-function-parameter:
  identifier

anonymous-function-body:
  expression
  block

```

=> 演算子と代入 (=) は優先順位が同じで、結合規則が右から左です。

`async` 修飾子を持つ匿名関数は、非同期関数で、エラー! 参照元が見つかりません。で説明する規則に従います。

*lambda-expression* 形式の匿名関数のパラメーターは、明示的または暗黙的に型指定できます。明示的に型指定したパラメーター リストでは、各パラメーターの型は明示的に宣言されます。暗黙に型指定されたパラメーター リストの場合、パラメーターの型は匿名関数が発生したコンテキストから推論されます。特に匿名関数を互換性のあるデリゲート型または式ツリー型に変換すると、パラメーター型が提供されます (0 を参照)。

1 つの暗黙に型指定されたパラメーターを持つ匿名関数では、かつてをパラメーター リストから省略できます。たとえば、次の形式の匿名関数があるとします。

( param ) => expr

これは次のように省略できます。

param => expr

*anonymous-method-expression* の形式の匿名関数のパラメーター リストは省略可能です。指定する場合は、パラメーターを明示的に型指定する必要があります。指定しない場合、匿名関数は `out` パラメーターを含まないパラメーター リストを持つデリゲートに変換できます。

匿名関数の `block` 本体は、匿名関数が到達不能ステートメント内に発生しない限り、到達可能です (§ 8.1 を参照)。

匿名関数の例をいくつか示します。

x => x + 1	// Implicitly typed, expression body
x => { return x + 1; }	// Implicitly typed, statement body
(int x) => x + 1	// Explicitly typed, expression body
(int x) => { return x + 1; }	// Explicitly typed, statement body
(x, y) => x * y	// Multiple parameters

```

() => Console.WriteLine()           // No parameters
async (t1,t2) => await t1 + await t2 // Async
delegate (int x) { return x + 1; }    // Anonymous method expression
delegate { return 1 + 1; }             // Parameter list omitted

```

*lambda-expression* と *anonymous-method-expression* の動作は同じですが、次の点が異なります。

- *anonymous-method-expression* ではパラメーター リストを全部省略できるので、値パラメーターの任意のリストをデリゲート型に変換できます。
- *lambda-expression* ではパラメーター型を省略して推論できるのに対し、*anonymous-method-expression* ではパラメーター型を明示的に宣言する必要があります。
- *lambda-expression* の本体には式またはステートメントブロックを使用できるのに対し、*anonymous-method-expression* の本体はステートメントブロックである必要があります。
- *lambda-expression* にのみ、互換性のある式ツリー型への変換があります (§ 4.6 を参照)。

### 7.15.1 匿名関数のシグネチャ

匿名関数の省略可能な *anonymous-function-signature* は、匿名関数の名前と、オプションで匿名関数の仮パラメーターの型を定義します。匿名関数のパラメーターのスコープは、*anonymous-function-body* です (3.7 を参照)。*anonymous-method-body* は、指定されたパラメーター リストがあれば、一緒に宣言空間を作成します (3.3 を参照)。したがって、匿名関数のパラメーターの名前が、スコープに *anonymous-method-expression* または *lambda-expression* を含むローカル変数、ローカル定数、またはパラメーターの名前と一致すると、コンパイル エラーになります。

匿名関数が *explicit-anonymous-function-signature* を持つ場合、互換性のあるデリゲート型および式ツリー型の集合は、同じパラメーター型と修飾子を同じ順序で持つデリゲート型に制限されます。メソッド グループの変換 (6.6 を参照) とは対照的に、匿名関数のパラメーター型では反変性はサポートされていません。匿名関数が *anonymous-function-signature* を持たない場合、互換性のあるデリゲート型および式ツリー型の集合は、*out* パラメーターを持たない型に制限されます。

*anonymous-function-signature* は、属性またはパラメーター配列を含むことはできません。しかし、*anonymous-function-signature* は、パラメーター リストにパラメーター配列を含むデリゲート型と互換性がある場合があります。

式ツリー型への変換は、互換性があっても、コンパイル時に失敗する可能性があります (4.6 を参照)。

### 7.15.2 匿名関数の本体

匿名関数の本体 (*expression* または *block*) は、次の規則に従います。

- 匿名関数がシグネチャを含む場合、シグネチャに指定されたパラメーターは本体で使用できます。匿名関数にシグネチャがない場合は、匿名関数を、パラメーターを持つデリゲート型または式型に変換できます (0 を参照) が、本体内でパラメーターにアクセスできません。
- 最も近い外側の匿名関数のシグネチャに指定されている *ref* パラメーターまたは *out* パラメーター (存在する場合) を除き、本体が *ref* パラメーターまたは *out* パラメーターにアクセスするとコンパイル エラーになります。
- *this* の型が構造体型の場合は、本体が *this* にアクセスするとコンパイル エラーになります。これは、アクセスが明示的 (*this.x* の場合) か暗黙 (*x* を構造体のインスタンス メンバーとする *x* の

場合) かに関係なく発生します。この規則は、単にこのようなアクセスを禁止するものであり、メンバー検索の結果が構造体のメンバーかどうかには影響しません。

- 本体は匿名関数の外部変数 (7.15.5 を参照) にアクセスします。外部変数のアクセスでは、*lambda-expression* または *anonymous-method-expression* が評価された時点でのアクティブな変数のインスタンスが参照されます (7.15.6 を参照)。
- 本体内の `goto` ステートメント、`break` ステートメント、または `continue` ステートメントのターゲットがその本体の外側、または含まれる匿名関数の本体内に存在すると、コンパイルエラーになります。
- 本体の `return` ステートメントは、外側の関数メンバーからではなく、最も近い外側の匿名関数の呼び出しから制御を返します。`return` ステートメントに指定する式は、最も近い外側の *lambda-expression* または *anonymous-method-expression* が変換された後のデリゲート型または式ツリー型の戻り値の型に暗黙的に変換できる必要があります (0 を参照)。

*lambda-expression* または *anonymous-method-expression* の評価および呼び出しを使用する以外に匿名関数の `block` を実行する方法があるかどうかは、明示的に指定されていません。特に、コンパイラは、1つ以上の名前付きメソッドまたは型を同期することによって匿名関数の実装を選択する場合があります。このような同期された要素の名前は、コンパイラのために予約された形式である必要があります。

### 7.15.3 オーバーロードの解決法

引数リストの匿名関数は、型推論とオーバーロードの解決に関与しています。厳密な規則については、7.5.2 および 7.5.3 を参照してください。

オーバーロードの解決における匿名関数の効果を次の例に示します。

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

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

`ItemList<T>` クラスには、`Sum` メソッドが 2 つあります。それぞれのメソッドは、リスト項目から値を抽出して合計する引数 `selector` を使用します。抽出される値には `int` または `double` のいずれかを使用でき、結果として得られる合計も同様に `int` または `double` のいずれかになります。

たとえば、`Sum` メソッドを使用して、明細行のリストの合計を順番に計算できます。

```
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、匿名関数 `d => d.UnitCount` が `Func<Detail, int>` および `Func<Detail, double>` の両方と互換性があるため、`Sum` メソッドは両方とも適切です。ただし、`Func<Detail, int>` への変換の方が `Func<Detail, double>` への変換よりも適切であるため、オーバーロード解決では最初の `Sum` メソッドが選択されます。

`orderDetails.Sum` の 2 番目の呼び出しでは、匿名関数の `d => d.UnitPrice * d.UnitCount` で `double` の値型が生成されるため、2 番目の `Sum` メソッドだけが適切です。したがって、オーバーロード解決では 2 番目の `Sum` メソッドがその呼び出しのために選択されます。

#### 7.15.4 匿名関数と動的バインディング

匿名関数は、動的にバインドされる操作のレシーバー、引数、またはオペランドにはできません。

#### 7.15.5 外部変数

スコープに *lambda-expression* または *anonymous-method-expression* を含むローカル変数、値パラメーター、またはパラメーター配列は、匿名関数の "外部変数" と呼ばれます。クラスのインスタンス関数メンバーでは、`this` 値は値パラメーターと見なされ、関数メンバー内に含まれる匿名関数の外部変数です。

##### 7.15.5.1 キャプチャされた外部変数

外部変数が匿名関数によって参照されることを、外部変数が匿名関数によって "キャプチャされた" と表現します。通常、ローカル変数の有効期間は、関連付けられているブロックまたはステートメントの実行中に限定されています(5.1.7 を参照)。しかし、キャプチャされた外部変数の有効期間は、少なくとも、匿名関数から作成されたデリゲートまたは式ツリーがガベージコレクションの対象になるまで延長されます。

次に例を示します。

```

using System;
delegate int D();
class Test
{
    static D F() {
        int x = 0;
        D result = () => ++x;
        return result;
    }
    static void Main() {
        D d = F();
        Console.WriteLine(d());
        Console.WriteLine(d());
        Console.WriteLine(d());
    }
}

```

ここでは、ローカル変数 `x` が匿名関数によってキャプチャされ、`x` の有効期間は、少なくとも `F` が返すデリゲートがガベージコレクションの対象になる(これはプログラムの最後まで発生しません)ま

で延長されます。匿名関数のすべての呼び出しは `x` の同じインスタンスで行われるため、この例の出力は次のようになります。

```
1
2
3
```

ローカル変数または値パラメーターが匿名関数によってキャプチャされると、そのローカル変数またはパラメーターは固定変数 (18.3 を参照) ではなく移動可能変数と見なされるようになります。したがって、キャプチャされた外部変数のアドレスを使用する `unsafe` コードは、まず `fixed` ステートメントを使用して変数を固定する必要があります。

キャプチャされていない変数とは異なり、キャプチャされたローカル変数は複数の実行スレッドに同時に公開できることに注意してください。

#### 7.15.5.2 ローカル変数のインスタンス化

ローカル変数は、実行が変数のスコープ内に入ると、"インスタンス化された" と見なされます。たとえば、次のメソッドが呼び出されると、ローカル変数 `x` は、ループが反復処理されるたびに 1 回、合計で 3 回インスタンス化および初期化されます。

```
static void F() {
    for (int i = 0; i < 3; i++) {
        int x = i * 2 + 1;
        ...
    }
}
```

しかし、`x` の宣言をループの外側に移動すると、`x` のインスタンス化は 1 回しか行われません。

```
static void F() {
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        ...
    }
}
```

キャプチャされていない場合、ローカル変数が何回インスタンス化されるかを厳密に判断する方法はありません。これは、インスタンス化の有効期間の間には関連性がなく、各インスタンス化が同じ格納場所を使用する可能性があるためです。しかし、匿名関数がローカル変数をキャプチャする場合は、インスタンス化の効果が明らかになります。

次の例を参照してください。

```
using System;
delegate void D();
class Test
{
    static D[] F() {
        D[] result = new D[3];
        for (int i = 0; i < 3; i++) {
            int x = i * 2 + 1;
            result[i] = () => { Console.WriteLine(x); };
        }
        return result;
    }
}
```

```

    static void Main() {
        foreach (D d in F())
            d();
    }
}

```

この例では、次のように出力されます。

```

1
3
5

```

しかし、`x` の宣言をループの外側に移動すると、次のようにになります。

```

static D[] F() {
    D[] result = new D[3];
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        result[i] = () => { Console.WriteLine(x); };
    }
    return result;
}

```

この出力は次のとおりです。

```

5
5
5

```

`for` ループで反復変数が宣言されると、その変数そのものがループの外で宣言されたと見なされます。したがって、反復変数自体をキャプチャするように例を変更すると、次のようにになります。

```

static D[] F() {
    D[] result = new D[3];
    for (int i = 0; i < 3; i++) {
        result[i] = () => { Console.WriteLine(i); };
    }
    return result;
}

```

反復変数の 1 つのインスタンスだけがキャプチャされ、次のような出力が生成されます。

```

3
3
3

```

匿名関数のデリゲートは、いくつかのキャプチャされた変数を共有しながら、他の個別のインスタンスも持つことができます。たとえば、`F` が次のように変更されたとします。

```

static D[] F() {
    D[] result = new D[3];
    int x = 0;
    for (int i = 0; i < 3; i++) {
        int y = 0;
        result[i] = () => { Console.WriteLine("{0} {1}", ++x, ++y); };
    }
    return result;
}

```

ここで、3 つのデリゲートは `x` の同じインスタンスと `y` の個別のインスタンスをキャプチャし、出力は次のようにになります。

```

1 1
2 1
3 1

```

複数の異なる匿名関数が外部変数の同じインスタンスをキャプチャできます。次に例を示します。

```
using System;
delegate void Setter(int value);
delegate int Getter();
class Test
{
    static void Main() {
        int x = 0;
        Setter s = (int value) => { x = value; };
        Getter g = () => { return x; };
        s(5);
        Console.WriteLine(g());
        s(10);
        Console.WriteLine(g());
    }
}
```

ここでは、2つの匿名関数がローカル変数 `x` の同じインスタンスをキャプチャし、それにより、その変数を通じて "コミュニケーション" できます。出力例は次のとおりです。

5  
10

### 7.15.6 匿名関数式の評価

匿名関数 `F` は常に、直接、またはデリゲート作成式 `new D(F)` を実行して、デリゲート型 `D` か式ツリー型 `E` に変換する必要があります。この変換によって、0で説明するように匿名関数の結果が決定されます。

## 7.16 クエリ式

"クエリ式" は、SQL や XQuery のような関係クエリ言語や階層型クエリ言語に似た統合言語構文をクエリに提供します。

```
query-expression:
    from-clause query-body

from-clause:
    from typeopt identifier in expression

query-body:
    query-body-clausesopt select-or-group-clause query-continuationopt

query-body-clauses:
    query-body-clause
    query-body-clauses query-body-clause

query-body-clause:
    from-clause
    let-clause
    where-clause
    join-clause
    join-into-clause
    orderby-clause

let-clause:
    let identifier = expression
```

```

where-clause:
  where boolean-expression

join-clause:
  join typeopt identifier in expression on expression equals expression

join-into-clause:
  join typeopt identifier in expression on expression equals expression into identifier

orderby-clause:
  orderby orderings

orderings:
  ordering
  orderings , ordering

ordering:
  expression ordering-directionopt

ordering-direction:
  ascending
  descending

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

select-clause:
  select expression

group-clause:
  group expression by expression

query-continuation:
  into identifier query-body

```

クエリ式は **from** 句で始まり、**select** 句か **group** 句で終わります。最初の **from** 句には、0 個以上の **from**、**let**、**where**、**join**、または **orderby** 句を続けて記述できます。各 **from** 句は、"シーケンス" の要素にわたる "範囲変数" を導入するジェネレーターです。各 **let** 句は、前の範囲変数を使用して計算された値を表す範囲変数を導入します。各 **where** 句は、結果から項目を除外するフィルターです。各 **join** 句は、ソース シーケンスの指定のキーを別のシーケンスのキーと比較して、一致するペアを生成します。各 **orderby** 句は、指定された基準に従って項目を並べ替えます。最後の **select** または **group** 句は、範囲変数の結果の形状を指定します。最後に **into** 句は、1 つのクエリの結果をその後のクエリのジェネレーターとして扱うことで、クエリの "連結" に使用することができます。

### 7.16.1 クエリ式のあいまいさ

クエリ式には、所定のコンテキストで特別な意味を持つ識別子など、多数の "コンテキストキーワード" が含まれています。具体的には **from**、**where**、**join**、**on**、**equals**、**into**、**let**、**orderby**、**ascending**、**descending**、**select**、**group**、および **by** があります。これらの識別子をキーワード、または簡易名として混合使用したことによるクエリ式のあいまいさを避けるため、これらの識別子はクエリ式のどの場所で使用されてもキーワードと見なされます。

このため、クエリ式は "**from identifier**" で始まる任意の式で、その後に ";"、"="、"," を除く任意のトークンが続きます。

これらの語をクエリ式で識別子として使用するには、前に "@" を付けます (2.4.2 を参照)。

## 7.16.2 クエリ式の書き換え

C# 言語では、クエリ式の実行の意味を指定しません。クエリ式は、クエリ式パターンに準拠するメソッドの呼び出しに書き換えられます(7.16.3 を参照)。具体的には、クエリ式は、`Where`、`Select`、`SelectMany`、`Join`、`GroupJoin`、`OrderBy`、`OrderByDescending`、`ThenBy`、`ThenByDescending`、`GroupBy`、`Cast` という名前のメソッドの呼び出しに書き換えられます。これらのメソッドは、7.16.3 で説明しているように、特定のシグネチャと結果型を持ちます。これらのメソッドには、クエリ対象のオブジェクトのインスタンス メソッドか、オブジェクトの外部の拡張メソッドを使用できるほか、これらのメソッドでクエリの実際の実行が実装されます。

クエリ式からメソッド呼び出しへの書き換えは、型の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行われる構文マップです。書き換えは構文的に正しいことが保証されていますが、意味的に正しい C# コードが生成されるかどうかは保証されていません。クエリ式の書き換えに続いて得られたメソッド呼び出しは、通常のメソッド呼び出しとして処理されます。これによって、メソッドが存在しない、引数の型が正しくない、メソッドがジェネリックで型推論が失敗する、などのエラーが明らかになることがあります。

クエリ式は、それ以上縮小が不可能になるまで、次の書き換えを反復的に適用することによって処理されます。書き換えは適用された順にリストされます。各セクションは、前のセクションの書き換えが完全に実行されたことを前提とするため、終了すると、同じクエリ式の処理のために後で再参照されることはありません。

クエリ式では範囲変数に割り当てることができません。ただし、ここに示した構文変換の方法では C# の実装が不可能な場合があるため、そのような場合にはこの制限を適用しなくともかまいません。

特定の書き換えは、\* で表される透過的識別子を範囲変数に挿入します。透過的識別子の特殊プロパティについては、詳しく説明します(7.16.2.7)。

### 7.16.2.1 連続した select 句および groupby 句

連続したクエリ式の例を示します。

```
from ... into x ...
```

これは次のように書き換えられます。

```
from x in ( from ... ) ...
```

次のセクションの書き換えでは、クエリに `into` の連続がないと仮定しています。

次の例を参照してください。

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

これは次のように書き換えられます。

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

最後の書き換えは次のようになります。

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

### 7.16.2.2 明示的な範囲変数型

範囲変数型を明示的に指定する `from` 句があります。

```
from T x in e
```

これは次のように書き換えられます。

```
from x in (e) . Cast <T> ()
```

範囲変数型を明示的に指定する `join` 句があります。

```
join T x in e on k1 equals k2
```

これは次のように書き換えられます。

```
join x in (e) . Cast <T> () on k1 equals k2
```

次のセクションの書き換えでは、クエリに明示的な範囲変数型がないと仮定しています。

次の例を参照してください。

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

これは次のように書き換えられます。

```
from c in customers.Cast<Customer>()
where c.City == "London"
select c
```

最後の書き換えは次のようになります。

```
customers.
Cast<Customer>().
where(c => c.City == "London")
```

明示的な範囲変数の型は、非ジェネリックな `IEnumerable` インターフェイスを実装するコレクションのクエリには便利ですが、ジェネリックな `IEnumerable<T>` インターフェイスを実装する際には役に立ちません。前の例では、`customers` が型 `ArrayList` であればそのような状況になります。

### 7.16.2.3 変質クエリ式

次の形式のクエリ式があります。

```
from x in e select x
```

これは次のように書き換えられます。

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

次の例を参照してください。

```
from c in customers
select c
```

これは次のように書き換えられます。

```
customers.Select(c => c)
```

変質クエリ式は、ソースの要素を普通に選択する式です。書き換えの以降のフェーズになると、他の書き換え手順によって導入された変質クエリは、ソースと置き換えることによって削除されます。しかし、クエリのクライアントにソースの型と ID を明らかにするため、クエリ式の結果がソースオブ

ジェクトそのものではないことが重要です。そのためこの手順は、ソースで明示的に **Select** を呼び出すことにより、ソースコードに直接書き込まれた変質クエリを保護します。したがって、**Select** の導入者、およびその他のクエリのオペレーターの責任で、これらのメソッドがソースオブジェクトそのものを返さないように確認します。

#### 7.16.2.4 from 句、let 句、where 句、join 句、orderby 句

2番目の **from** 句に、**select** 句が続くクエリ式を示します。

```
from x1 in e1
from x2 in e2
select v
```

これは次のように書き換えられます。

```
( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => v )
```

2番目の **from** 句に **select** 以外の句が続くクエリ式を示します。

```
from x1 in e1
from x2 in e2
...
```

これは次のように書き換えられます。

```
from * in ( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => new { x1 , x2 } )
```

**let** 句を使用したクエリ式を示します。

```
from x in e
let y = f
...
```

これは次のように書き換えられます。

```
from * in ( e ) . Select ( x => new { x , y = f } )
```

**where** 句を使用したクエリ式を示します。

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

これは次のように書き換えられます。

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

**join** 句を **into** なしで使用し、**select** 句が続くクエリ式を示します。

```
from x1 in e1
join x2 in e2 on k1 equals k2
select v
```

これは次のように書き換えられます。

```
( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => v )
```

`join` 句を `into` なしで使用し、`select` 以外の句が続くクエリ式を示します。

```
from x1 in e1
join x2 in e2 on k1 equals k2
...
```

これは次のように書き換えられます。

```
from * in ( e1 ) . Join(
    e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => new { x1 , x2 })
...
```

`join` 句を `into` と共に使用し、`select` 句が続くクエリ式を示します。

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
select v
```

これは次のように書き換えられます。

```
( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => v )
```

`join` 句を `into` と共に使用し、`select` 以外の句が続くクエリ式を示します。

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
...
```

これは次のように書き換えられます。

```
from * in ( e1 ) . GroupJoin(
    e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => new { x1 , g })
...
```

`orderby` 句を使用したクエリ式を示します。

```
from x in e
orderby k1 , k2 , ... , kn
...
```

これは次のように書き換えられます。

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

順序を指定する句が `descending` 方向インジケーターを示している場合は、`OrderByDescending` または `ThenByDescending` の呼び出しが生成されます。

次の書き換えは、`let`、`where`、`join`、`orderby` 句がなく、それぞれのクエリ式の最初の `from` 句が 1 個だけであると仮定しています。

次の例を参照してください。

**Chapter** エラー! [ホーム] タブを使用して、ここに表示する文字列に Heading 1 を適用してください。 エラー! [ホーム] タブ

```
from c in customers
from o in c.Orders
select new { c.Name, o.orderID, o.Total }
```

これは次のように書き換えられます。

```
customers.
SelectMany(c => c.Orders,
    (c,o) => new { c.Name, o.OrderID, o.Total })
)
```

次の例を参照してください。

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

これは次のように書き換えられます。

```
from * in customers.
    SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

最後の書き換えは次のようになります。

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })
```

xはコンパイラによって生成された識別子で、参照もアクセスもできません。

次の例を参照してください。

```
from o in orders
let t = o.Details.Sum(d => d.UnitPrice * d.Quantity)
where t >= 1000
select new { o.OrderID, Total = t }
```

これは次のように書き換えられます。

```
from * in orders.
    Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) }).
where t >= 1000
select new { o.OrderID, Total = t }
```

最後の書き換えは次のようになります。

```
orders.
Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) }).
Where(x => x.t >= 1000).
Select(x => new { x.o.OrderID, Total = x.t })
```

xはコンパイラによって生成された識別子で、参照もアクセスもできません。

次の例を参照してください。

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
select new { c.Name, o.OrderDate, o.Total }
```

これは次のように書き換えられます。

```
customers.Join(orders, c => c.CustomerID, o => o.CustomerID,
    (c, o) => new { c.Name, o.OrderDate, o.Total })
```

次の例を参照してください。

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID into co
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

これは次のように書き換えられます。

```
from * in customers.
    GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
              (c, co) => new { c, co })
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

最後の書き換えは次のようになります。

```
customers.
GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
          (c, co) => new { c, co }).
Select(x => new { x, n = x.co.Count() }).
Where(y => y.n >= 10).
Select(y => new { y.x.c.Name, OrderCount = y.n })
```

x および y はコンパイラによって生成された識別子で、参照もアクセスもできません。

次の例を参照してください。

```
from o in orders
orderby o.Customer.Name, o.Total descending
select o
```

最後の書き換えは次のようになります。

```
orders.
OrderBy(o => o.Customer.Name).
ThenByDescending(o => o.Total)
```

### 7.16.2.5 Select 句

次の形式のクエリ式があります。

```
from x in e select v
```

これは次のように書き換えられます。

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

ただし、v が x 識別子の場合の書き換えは単に次のようになります。

```
( e )
```

次に例を示します。

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

これは単純に次のように書き換えられます。

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

### 7.16.2.6 Groupby 句

次の形式のクエリ式があります。

```
from x in e group v by k
```

これは次のように書き換えられます。

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

ただし、v が x 識別子の場合の書き換えは次のようにになります。

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

次の例を参照してください。

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

これは次のように書き換えられます。

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

### 7.16.2.7 透過的識別子

特定の書き換えは、\* で表される透過的識別子を範囲変数に挿入します。透過的識別子は厳密には言語機能ではありません。クエリ式の書き換えプロセスの中間手順として存在するだけです。

クエリ書き換えに透過的識別子が挿入されると、以降の書き換え手順は透過的識別子を匿名関数および匿名オブジェクト初期化子に伝達します。そのようなコンテキストでは、透過的識別子は次のように動作します。

- 透過的識別子が匿名関数のパラメーターとして発生すると、関連する匿名型のメンバーは、自動的に匿名関数の本体のスコープに入ります。
- 透過的識別子を持つメンバーがスコープ内にある場合、そのメンバーのメンバーもスコープ内になります。
- 透過的識別子が匿名オブジェクト初期化子のメンバー宣言子として発生すると、透過的識別子を持つメンバーが導入されます。

上記の書き換え手順では、透過的識別子は、1つのオブジェクトに複数の範囲変数をキャプチャするために、常に匿名型と一緒に導入されます。C# の実装では、匿名型とは異なる機構を使用して複数の範囲変数をグループ化することができます。次の書き換え例は匿名型が使用されていると仮定して、透過的識別子がどのように書き換えられるかを示します。

次の例を参照してください。

```
from c in customers  
from o in c.Orders  
orderby o.Total descending  
select new { c.Name, o.Total }
```

これは次のように書き換えられます。

```
from * in customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o })  
orderby o.Total descending  
select new { c.Name, o.Total }
```

これはさらに次のように書き換えられます。

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(* => o.Total).
Select(* => new { c.Name, o.Total })
```

透過的識別子が消去されると、次と同じになります。

```
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.Total })
```

xはコンパイラによって生成された識別子で、参照もアクセスもできません。

次の例を参照してください。

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

これは次のように書き換えられます。

```
from * in customers.
Join(orders, c => c.CustomerID, o => o.CustomerID,
     (c, o) => new { c, o }).
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

これはさらに次のように縮小されます。

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o }).
Join(details, * => o.OrderID, d => d.OrderID, (*, d) => new { *, d }).
Join(products, * => d.ProductID, p => p.ProductID, (*, p) => new { *, p }).
Select(* => new { c.Name, o.OrderDate, p.ProductName })
```

最後の書き換えは次のようになります。

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID,
     (c, o) => new { c, o }).
Join(details, x => x.o.OrderID, d => d.OrderID,
     (x, d) => new { x, d }).
Join(products, y => y.d.ProductID, p => p.ProductID,
     (y, p) => new { y, p }).
Select(z => new { z.y.x.c.Name, z.y.x.o.OrderDate, z.p.ProductName })
```

x、yおよびzはコンパイラによって生成された識別子で、参照もアクセスもできません。

### 7.16.3 クエリ式パターン

“クエリ式パターン”は、型で実装してクエリ式をサポートできるメソッドのパターンを確立します。構文マップを使用してクエリ式をメソッドの呼び出しに書き換えると、型によるクエリ式パターンの実装方法に大幅な柔軟性がもたらされます。たとえば、パターンのメソッドをインスタンスメソッド、または拡張メソッドとして実装することができます。これは、この2つの呼び出し構文が同じで、匿名関数を両方に変換でき、デリゲート型も式のツリー型も要求できるためです。

クエリ式パターンをサポートするジェネリック型 `C<T>` の推奨形式を次に示します。ジェネリック型はパラメーターと結果型の適切な関係を示すために使用されますが、非ジェネリック型のパターンも実装することができます。

```

delegate R Func<T1,R>(T1 arg1);
delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);
class C
{
    public C<T> Cast<T>();
}
class C<T> : C
{
    public C<T> Where(Func<T,bool> predicate);
    public C<U> Select<U>(Func<T,U> selector);
    public C<V> SelectMany<U,V>(Func<T,C<U>> selector,
        Func<T,U,V> resultSelector);
    public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);
    public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,C<U>,V> resultSelector);
    public O<T> OrderBy<K>(Func<T,K> keySelector);
    public O<T> OrderByDescending<K>(Func<T,K> keySelector);
    public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);
    public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
        Func<T,E> elementSelector);
}
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> : C<T>
{
    public K Key { get; }
}

```

上記のメソッドは `Func<T1, R>` および `Func<T1, T2, R>` のジェネリック デリゲート型を使用しますが、パラメーターおよび結果型の同じ関係で他のデリゲート型または式ツリー型を同じように使用することもできます。

`C<T>` と `O<T>` の間の推奨関係に注意してください。この関係により、`ThenBy` メソッドと `ThenByDescending` メソッドが、`OrderBy` または `OrderByDescending` の結果でのみ使用できることが保証されます。また、`GroupBy` の結果で推奨される結果にも注意してください。シーケンスのシーケンスでは内部シーケンスに追加の `Key` プロパティがあります。

`System.Linq` 名前空間は、`System.Collections.Generic.IEnumerable<T>` インターフェイスを実装するあらゆる型のクエリ演算子パターンを実装します。

## 7.17 代入演算子

代入演算子は、変数、プロパティ、イベント、またはインデクサー要素に新しい値を代入します。

*assignment:*  
*unary-expression assignment-operator expression*

*assignment-operator:*

```
=  
+=  
-=  
*=  
/=  
%=  
&=  
|=  
^=  
<<=  
right-shift-assignment
```

代入の左辺のオペランドには、変数、プロパティ アクセス、インデクサー アクセス、またはイベント アクセスに分類される式を指定する必要があります。

= 演算子は "**単純代入演算子**" と呼ばれます。単純代入演算子は、右オペランドの値を左オペランドで指定されている変数、プロパティ、またはインデクサー要素に代入します。単純代入演算子の左オペランドには、イベントアクセスを指定できません(10.8.1 で示している場合を除く)。単純代入演算子については、7.17.1 で説明します。

= 演算子以外の代入演算子は、**複合代入演算子**と呼ばれます。これらの演算子は、2つの演算子に対して指定された演算を実行し、左オペランドで指定されている変数、プロパティ、またはインデクサー要素に結果の値を代入します。複合代入演算子については、7.17.2 で説明します。

イベントアクセス式で左オペランドに使用される **+=** 演算子および **-=** 演算子は、イベント代入演算子と呼ばれます。左オペランドがイベントアクセスの場合、他に有効な代入演算子はありません。イベント代入演算子については、7.17.3 で説明します。

代入演算子の結合規則は "右から左" です。つまり、演算は右から左にグループ化されます。たとえば、**a = b = c** という形式の式は、**a = (b = c)** として評価されます。

### 7.17.1 単純代入

= 演算子は "単純代入演算子" と呼ばれます。

単純代入の左オペランドが **E.P** または **E[E<sub>i</sub>]** という形式で、**E** のコンパイル時の型が **dynamic** の場合、代入は動的にバインドされます(7.2.2 を参照)。この場合、代入式のコンパイル時の型は **dynamic** であり、以下で説明する解決は、**E** の実行時の型に基づいて実行時に行われます。

単純代入では、右オペランドとして、左オペランドの型に暗黙に変換できる式を指定する必要があります。演算は、右オペランドの値を左オペランドで指定されている変数、プロパティ、またはインデクサー要素に代入します。

単純代入式の結果は、左オペランドに代入される値です。結果は左オペランドと同じ型で、常に値として分類されます。

左オペランドがプロパティまたはインデクサー アクセスの場合、プロパティまたはインデクサーには **set** アクセサーが必要です。そうでない場合は、バインディング エラーが発生します。

**x = y** の形式による単純代入の実行時の処理は、次の手順で構成されています。

- **x** が変数の場合。

- **x** を評価して変数を生成します。

- *y* を評価し、必要に応じて、暗黙の型変換 (0 を参照) を行って *x* の型に変換します。
- *x* で指定されている変数が *reference-type* の配列要素の場合は、ランタイム チェックを実施し、*y* に対して計算された値と *x* が要素である配列インスタンスとの間に互換性があることを確認します。*y* が `null` の場合、または *y* によって参照されているインスタンスの実際の型から *x* を含む配列インスタンスの実際の要素型に対する暗黙の参照変換 (6.1.6 を参照) が存在する場合、チェックは成功します。それ以外の場合は、`System.ArrayTypeMismatchException` がスローされます。
- *y* の評価と変換から得られた値を *x* の評価によって得られる場所に格納します。
- *x* がプロパティ アクセスまたはインデクサー アクセスの場合。
  - *x* に関連付けられたインスタンス式 (*x* が `static` でない場合) および引数リスト (*x* がインデクサー アクセスの場合) を評価し、後で行う `set` アクセサーの呼び出しでは、その結果を使用します。
  - *y* を評価し、必要に応じて、暗黙の型変換 (0 を参照) を行って *x* の型に変換します。
  - *y* に対して計算された値を引数 `value` として、*x* の `set` アクセサーを呼び出します。

配列の共変性規則 (12.5 を参照) により、配列型の `A[]` から `B[]` に対する暗黙の参照変換が存在している場合は、配列型 `B` の値を配列型 `A` のインスタンスへの参照にできます。このような規則のため、*reference-type* の配列要素への代入では、ランタイム チェックを実施して、代入される値と配列インスタンスとの間に互換性があることを確認する必要があります。次に例を示します。

```
string[] sa = new string[10];
object[] oa = sa;
oa[0] = null;           // Ok
oa[1] = "Hello";        // Ok
oa[2] = new ArrayList(); // ArrayTypeMismatchException
```

この例で、`ArrayList` のインスタンスは `string[]` の要素に格納できないため、最後の代入によって `System.ArrayTypeMismatchException` がスローされます。

*struct-type* で宣言されているプロパティまたはインデクサーが代入の対象である場合、プロパティ アクセスまたはインデクサー アクセスと関連付けられているインスタンス式は、変数として分類される必要があります。インスタンス式が値に分類される場合は、バインディング エラーが発生します。7.6.4 により、フィールドにも同じ規則が適用されます。

例のような宣言について考えます。

```
struct Point
{
    int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int x {
        get { return x; }
        set { x = value; }
    }
}
```

```

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

struct Rectangle
{
    Point a, b;
    public Rectangle(Point a, Point b) {
        this.a = a;
        this.b = b;
    }
    public Point A {
        get { return a; }
        set { a = value; }
    }
    public Point B {
        get { return b; }
        set { b = value; }
    }
}

```

次に例を示します。

```

Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;

```

`p` および `r` は変数であるため、`p.X`、`p.Y`、`r.A`、および `r.B` への代入は認められます。これとは別の例を示します。

```

Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;

```

この例では、`r.A` および `r.B` は変数ではないため、代入はすべて無効です。

## 7.17.2 複合代入

複合代入の左オペランドが `E.P` または `E[Ei]` という形式で、`E` のコンパイル時の型が `dynamic` の場合、代入は動的にバインドされます (7.2.2 を参照)。この場合、代入式のコンパイル時の型は `dynamic` であり、以下で説明する解決は、`E` の実行時の型に基づいて実行時に行われます。

`x op=y` という形式の演算は、`(x) op y` という形式の演算と同じように、二項演算子オーバーロードの解決規則 (7.3.4 を参照) を適用して処理されます。`R` を選択された演算子の戻り値の型、`T` を型 `x` とします。処理の方法は、次のとおりです。

- 型 `R` から型 `T` への式の暗黙的な変換が存在する場合、演算は `x = (T)((x) op y)` として評価されます。ただし、`x` は 1 回だけ評価されます。
- 選択された演算子が定義済みの演算子で、`R` を "明示的に" `T` に変換でき、`y` を "暗黙的に" `T` に変換できるか演算子がシフト演算子である場合は、演算は `x = (T)((x) op y)` として評価されます。ただし、`x` は 1 回だけ評価されます。

- それ以外の場合、その複合代入は無効であり、バインディング エラーが発生します。

"1回だけ評価される" とは、 $x \ op \ y$  の評価において、 $x$  を構成するすべての式の結果は一時的に保存されて、 $x$ への代入を実行するときに再利用されることを意味します。たとえば、`int[]` を返すメソッド A と `int` を返すメソッド B および C で構成される `A() [B()] += C()` という代入では、メソッドは A、B、C という順序で 1 回だけ呼び出されます。

複合代入の左オペランドがプロパティ アクセスまたはインデクサー アクセスの場合、そのプロパティまたはインデクサーには `get` アクセサーと `set` アクセサーの両方が必要です。そうでない場合は、バインディング エラーが発生します。

前の 2 番目の規則では、特定のコンテキストにおいて  $x \ op= y$  を  $x = (T)((x) \ op \ y)$  として評価することが認められています。この規則は、左オペランドが型 `sbyte`、`byte`、`short`、`ushort`、または `char` のときに定義済み演算子を複合演算子として使用できるように、設けられています。両方の引数がこれらの型のいずれかであるときにも、7.3.6.2 で示されているように、定義済み演算子は `int` 型の結果を生成します。したがって、キャストを行わないと、結果を左オペランドに代入できません。

定義済み演算子に対する規則による影響は、 $x \ op \ y$  と  $(x) = y$  の両方が認められる場合は  $x \ op= y$  が認められることだけです。次に例を示します。

```
byte b = 0;
char ch = '\0';
int i = 0;

b += 1;           // Ok
b += 1000;        // Error, b = 1000 not permitted
b += i;           // Error, b = i not permitted
b += (byte)i;    // Ok

ch += 1;          // Error, ch = 1 not permitted
ch += (char)1;    // Ok
```

各エラーの理由は、対応する単純代入にもエラーがあるためです。

これは、複合代入演算でリフト演算が使用できることも意味します。次に例を示します。

```
int? i = 0;
i += 1;           // Ok
```

リフト演算子 `+(int?, int?)` が使用されています。

### 7.17.3 イベント代入

`+=` 演算子または `-=` 演算子の左オペランドがイベント アクセスとして分類される場合、式は次のように評価されます。

- イベント アクセスのインスタンス式が存在する場合は、インスタンス式を評価します。
- `+=` 演算子または `-=` 演算子の右オペランドを評価し、必要に応じて暗黙の型変換 (0 を参照) を行って左オペランドの型に変換します。
- 右オペランドを評価し、必要に応じて変換を行った後、右オペランドで構成される引数リストを使用してイベントのイベント アクセサーを呼び出します。演算子に `+=` が使用されていた場合、`add` アクセサーが呼び出され、演算子に `-=` が使用されていた場合は `remove` アクセサーが呼び出されます。

イベント代入式は、値を生成しません。このため、イベント代入式は、*statement-expression* (8.6 を参照) のコンテキストにおいてのみ有効です。

## 7.18 式

*expression* は、*non-assignment-expression* または *assignment* です。

```

expression:
  non-assignment-expression
  assignment

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

## 7.19 定数式

*constant-expression* は、コンパイル時に完全に評価できる式です。

```

constant-expression:
  expression
```

定数式は、`null` リテラルか、`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal`、`bool`、`object`、`string`、または任意の列挙型を持つ値である必要があります。定数式では、次の構成要素のみを使用できます。

- リテラル (`null` リテラルを含む)
- クラス型および構造体型の `const` メンバーに対する参照
- 列挙型のメンバーに対する参照
- `const` パラメーターまたはローカル変数に対する参照
- それ自体が定数式である、かっこで囲まれた部分式
- キャスト式、ただし変換先の型が上記の型のいずれかであること
- `checked` 式と `unchecked` 式
- 既定値の式
- 定義済み単項演算子 `+`、`-`、`!`、および `~`
- 定義済み二項演算子 `+`、`-`、`*`、`/`、`%`、`<<`、`>>`、`&`、`|`、`^`、`&&`、`||`、`==`、`!=`、`<`、`>`、`<=`、および `>=` (ただし各オペランドが上記の型のいずれかであること)
- 条件演算子 `?:`

定数式では、次の変換を使用できます。

- 恒等変換
- 数値変換
- 列挙値変換
- 定数式変換
- 暗黙および明示的な参照変換 (変換のソースが `null` 値を評価する定数式の場合)

ボックス化、ボックス化解除、および非 null 値の暗黙の参照変換などは定数式では使用できません。次に例を示します。

```
class C {  
    const object i = 5; // error: boxing conversion not permitted  
    const object str = "hello"; // error: implicit reference conversion  
}
```

`i` の初期化は、ボックス化変換が必要なため、エラーになります。`str` の初期化は、非 null 値からの暗黙の参照変換が必要なため、エラーになります。

上記の条件を満たすたびに、式はコンパイル時に評価されます。定数ではない構造を含む大きな式のサブ式である式についても、このことが成り立ちます。

定数式のコンパイル時の評価では、非定数式の実行時の評価と同じ規則を使用しますが、実行時の評価が例外をスローすると、コンパイル時の評価ではコンパイル エラーが発生する点が異なります。

定数式が `unchecked` コンテキストの中に明示的に置かれていない限り、式のコンパイル時評価の過程で、整数型の算術演算および型変換に関してオーバーフローが発生すると、常にコンパイル エラーになります (7.19 を参照)。

定数式は、以下のコンテキストで使用されます。これらのコンテキストでは、コンパイル時に式を完全に評価できないと、コンパイル エラーが発生します。

- 定数宣言 (10.4 を参照)
- 列挙メンバー宣言 (14.3 を参照)
- 仮パラメーター リストの既定引数 (§ 10.6.1 を参照)
- `case` ステートメントの `switch` ラベル (8.7.2 を参照)
- `goto case` ステートメント (8.9.3 を参照)
- 初期化子を含む配列作成式における次元の長さ (7.6.10.4 を参照)
- 属性 (17 を参照)

暗黙的な定数式変換 (6.1.9 を参照) では、定数式の値が変換先の型の範囲内である場合、`int` 型の定数式を `sbyte`、`byte`、`short`、`ushort`、`uint`、または `ulong` に変換できます。

## 7.20 Boolean 式

`boolean-expression` の結果は `bool` 型となります。結果が直接示されるか、次に示すように特定のコンテキストの `operator true` を適用することで示されます。

*boolean-expression:*  
    *expression*

`if-statement` (8.7.1 を参照)、`while-statement` (8.8.1 を参照)、`do-statement` (8.8.2 を参照)、または `for-statement` (8.8.3 を参照) の制御条件式は `boolean-expression` です。`? :` 演算子 (7.14 を参照) の制御条件式は `boolean-expression` と同じ規則に従いますが、演算子の優先順位の理由から、`conditional-or-expression` として分類されます。

`boolean-expression E` は、次のように、`bool` 型の値を生成できるようにする必要があります。

- `E` は `bool` に暗黙的に変換される場合、実行時にその暗黙的な変換が適用されます。

- それ以外の場合、単項演算子のオーバーロードの解決(§ 7.3.3 を参照)が `E` で演算子 `true` の一意の最適実装を検索するために使用され、その実装が実行時に適用されます。
- 演算子が見つからない場合、バインディング時のエラーになります。

`operator true` および `operator false` を実装する型の例については、11.4.2 の `DBBool` 構造体型を参照してください。

## 8. ステートメント

C# にはさまざまなステートメントが用意されています。これらの多くは、C および C++ でプログラムを作成している開発者にとって理解しやすいステートメントです。

```
statement:  
    labeled-statement  
    declaration-statement  
    embedded-statement  
  
embedded-statement:  
    block  
    empty-statement  
    expression-statement  
    selection-statement  
    iteration-statement  
    jump-statement  
    try-statement  
    checked-statement  
    unchecked-statement  
    lock-statement  
    using-statement  
    yield-statement
```

*embedded-statement* 非終端記号は、ステートメントを他のステートメント内に記述する場合に使用します。 *statement* ではなく *embedded-statement* を使用すると、これらのコンテキストで宣言ステートメントとラベル付きステートメントを使用できなくなります。次の例を参照してください。

```
void F(bool b) {  
    if (b)  
        int i = 44;  
}
```

*if* ステートメントは、*if* 分岐用に *statement* ではなく *embedded-statement* を必要とするため、コンパイルエラーになります。このコードが許可された場合、変数 *i* は宣言されますが、使用することはできません。ただし、ブロック内で *i* を宣言しているため、このコード例は有効です。

### 8.1 終了点と到達可能性

すべてのステートメントには終了点があります。わかりやすく言うと、ステートメントの終了点とは、そのステートメント直後の場所です。複合ステートメント(埋め込みステートメントを含むステートメント)の実行規則により、制御が埋め込みステートメントの終了点に達したときに実行されるアクションが指定されます。たとえば、制御がブロック内のステートメントの終了点に到達すると、制御はブロック内の次のステートメントに移ります。

実行によりステートメントに到達できる場合、そのステートメントは到達可能と呼ばれます。逆に、ステートメントが実行される可能性がない場合、そのステートメントは到達不可能と呼ばれます。

次に例を示します。

```
void F() {
    Console.WriteLine("reachable");
    goto Label;
    Console.WriteLine("unreachable");
Label:
    Console.WriteLine("reachable");
}
```

`Console.WriteLine` の 2 回目の呼び出しは、このステートメントが実行される可能性がないので到達不可能です。

ステートメントが到達不可能であるとコンパイラが判断すると、警告が報告されます。ステートメントが到達不可能でもエラーになるわけではありません。

特定のステートメントまたは終了点が到達可能かどうかを判別するために、コンパイラは、各ステートメントに定義された到達可能性規則に従って、フロー解析を実行します。フロー解析では、ステートメントの動作を制御する定数式 (7.19 を参照) の値を考慮に入れますが、非定数式が取ることのできる値は考慮されません。つまり、制御フロー解析の目的では、指定された型の非定数式は、その型が取ることのできる値を持つと見なされます。

次に例を示します。

```
void F() {
    const int i = 1;
    if (i == 2) Console.WriteLine("unreachable");
}
```

`if` ステートメントのブール型の式は、`==` 演算子の両側のオペランドが定数なので定数式です。定数式はコンパイル時に評価され、値 `false` が生成されるので、`Console.WriteLine` の呼び出しは到達不可能と見なされます。ただし、`i` がローカル変数に変更されると次のようになります。

```
void F() {
    int i = 1;
    if (i == 2) Console.WriteLine("reachable");
}
```

実際には実行されなくても、`Console.WriteLine` の呼び出しは到達可能と見なされます。

関数メンバーの `block` は常に到達可能と見なされます。ブロック内の各ステートメントの到達可能性規則を連続して評価することで、任意のステートメントの到達可能性を判定できます。

次に例を示します。

```
void F(int x) {
    Console.WriteLine("start");
    if (x < 0) Console.WriteLine("negative");
}
```

2 番目の `Console.WriteLine` の到達可能性は次のように判定されます。

- `F` メソッドのブロックは到達可能であるため、最初の `Console.WriteLine` 式ステートメントは到達可能です。
- そのステートメントは到達可能であるため、最初の `Console.WriteLine` 式ステートメントの終了点は到達可能です。
- 最初の `Console.WriteLine` 式ステートメントの終了点は到達可能であるため、`if` ステートメントは到達可能です。

- `if` ステートメントのプール型の式の値は定数値 `false` ではないので、2番目の `Console.WriteLine` 式ステートメントは到達可能です。

ステートメントの最後の位置に到達するとコンパイルエラーが発生するのは、次の2つのような場合です。

- `switch` ステートメントでは、ある `switch` セクションから次の `switch` セクションへ、連続して実行することはできません。ある `switch` セクションから次の `switch` セクションに連続して実行すると、ステートメントリストの最後に到達した時点で、コンパイルエラーになります。このエラーは、通常、`break` ステートメントを挿入し忘れた場合に発生します。
- 値を計算する関数メンバーのブロックの終了点が到達可能になると、コンパイルエラーになります。このエラーは、通常、`return` ステートメントを挿入し忘れた場合に発生します。

## 8.2 ブロック

`block` を使うと、1つのステートメントが許容されるコンテキストに複数のステートメントを記述できます。

*block:*  
  { *statement-list<sub>opt</sub>* }

`block` は、中かっこ ({} ) で囲まれた省略可能な *statement-list* (8.2.1 を参照) から構成されます。ステートメントリストを省略すると、ブロックは空になります。

ブロックは宣言ステートメント (8.5 を参照) を含むことができます。ブロック内で宣言されたローカル変数または定数のスコープは、そのブロックです。

同一ブロック内では、式のコンテキストで使用される名前の意味は常に同じである必要があります (7.6.2.1 を参照)。

ブロックは次のように実行されます。

- ブロックが空の場合、制御はブロックの終了点に移ります。
- ブロックが空でない場合、制御はステートメントに移ります。制御がステートメントリストの終了点に達すると、そのブロックの終了点に制御が移ります。

ブロック自体が到達可能な場合、ブロックのステートメントリストは到達可能です。

ブロックが空であるか、またはステートメントリストの終了点が到達可能な場合、ブロックの終了点は到達可能です。

1つ以上の `yield` ステートメント (8.14 を参照) を含むブロックは、反復子ブロックと呼ばれます。反復子ブロックは、関数メンバーを反復子として実装する場合に使用します (10.14 を参照)。反復子ブロックには追加の制限が適用されます。

- `return` ステートメントを反復子ブロックで実行するとコンパイル時のエラーになります (`yield return` ステートメントは許可されます)。
- 反復子ブロックに `unsafe` コンテキスト (18.1 を参照) が含まれると、コンパイルエラーになります。反復子ブロックは、宣言が `unsafe` コンテキストに入れ子になっている場合でも、常に `safe` コンテキストを定義します。

### 8.2.1 ステートメントリスト

ステートメントリストは、順に記述された1つ以上のステートメントで構成されます。ステートメントリストは、*block* (8.2を参照)内、および*switch-block* (8.7.2を参照)内に記述できます。

```
statement-list:
  statement
  statement-list statement
```

ステートメントリストは、その最初のステートメントに制御を移すことで実行されます。制御がステートメントの終了点に達すると、次のステートメントに制御が移ります。制御が最後のステートメントの終了点に達すると、そのステートメントリストの終了点に制御が移ります。

ステートメントリスト内のステートメントは、次のいずれか1つでも当てはまれば到達可能です。

- ステートメントが最初のステートメントであり、ステートメントリスト自体が到達可能である場合
- 前のステートメントの終了点が到達可能である場合
- ステートメントがラベル付きステートメントであり、到達可能な `goto` ステートメントによってそのラベルが参照される場合

ステートメントリストの最後のステートメントの終了点が到達可能な場合は、ステートメントリストの終了点は到達可能です。

### 8.3 空のステートメント

*empty-statement* は何も実行しません。

```
empty-statement:
  ;
```

空のステートメントは、ステートメントが必要となるコンテキストで、実行する操作がない場合に使用されます。

空のステートメントの実行では、ステートメントの終了点に制御が移るだけです。このため、空のステートメントが到達可能な場合、空のステートメントの終了点は到達可能です。

空のステートメントは、次のように本体のない `while` ステートメントを記述する場合に使用できます。

```
bool ProcessMessage() {...}
void ProcessMessages() {
    while (ProcessMessage())
        ;
}
```

また、空のステートメントを使って、次のようにブロックを閉じる "}" の直前でラベルを宣言できます。

```
void F() {
    ...
    if (done) goto exit;
    ...
    exit: ;
}
```

## 8.4 ラベル付きステートメント

*labeled-statement* を使うと、ステートメントの前にラベルを付けることができます。ラベル付きステートメントはブロック内で使用できますが、埋め込みステートメントとしては使用できません。

```
labeled-statement:  
    identifier : statement
```

ラベル付きステートメントでは、*identifier* によって指定された名前のラベルを宣言します。ラベルのスコープは、入れ子になったブロックを含め、そのラベルが宣言されたブロック全体です。2つの同じ名前のラベルが重複したスコープを持つと、コンパイルエラーになります。

ラベルは、そのラベルのスコープ内にある **goto** ステートメント (8.9.3 を参照) から参照できます。つまり、**goto** ステートメントはブロック内で、およびブロックの外に制御を移すことができますが、ブロック外からブロック内に制御を移動させることはできません。

ラベルにはそれぞれの宣言空間があり、それぞれの識別子で干渉することはありません。次の例を参照してください。

```
int F(int x) {  
    if (x >= 0) goto x;  
    x = -x;  
    x: return x;  
}
```

これは有効であり、名前 *x* はパラメーターおよびラベルの両方として使用されます。

ラベル付きステートメントの実行は、ラベルの次のステートメントの実行と正確に一致します。

通常の制御フローによって提供される到達可能性に加えて、到達可能な **goto** ステートメントによってラベルが参照される場合、ラベル付きステートメントは到達可能です。ただし、例外として、**finally** ブロックを含む **try** の中に **goto** ステートメントがあり、ラベル付きステートメントが **try** の外部にあり、**finally** ブロックの終了点に到達できない場合は、**goto** ステートメントからラベル付きステートメントには到達できません。

## 8.5 宣言ステートメント

*declaration-statement* は、ローカル変数またはローカル定数を宣言します。宣言ステートメントはブロック内で使用できますが、埋め込みステートメントとしては使用できません。

```
declaration-statement:  
    local-variable-declaration ;  
    local-constant-declaration ;
```

### 8.5.1 ローカル変数宣言

*local-variable-declaration* では、1つ以上のローカル変数を宣言します。

```
local-variable-declaration:  
    local-variable-type local-variable-declarators  
  
local-variable-type:  
    type  
    var  
  
local-variable-declarators:  
    local-variable-declarator  
    local-variable-declarators , local-variable-declarator
```

```

local-variable-declarator:
  identifier
  identifier = local-variable-initializer

local-variable-initializer:
  expression
  array-initializer

```

*local-variable-declaration* の *local-variable-type* では、宣言によって導入される変数の型を直接指定するか、または **var** 識別子を使用して、初期化子に基づいて型を推論する必要があることを示します。この型の後には、*local-variable-declarator* のリストが続き、各宣言子により新規の変数が導入されます。*local-variable-declarator* は、変数を指定する *identifier* で構成されます。オプションで "=" トークンと、変数の初期値を指定する *local-variable-initializer* を続けることもできます。

ローカル変数宣言のコンテキストでは、識別子 **var** はコンテキストキーワード(2.4.3を参照)と同じように機能します。*local-variable-type* が **var** として指定されている場合、スコープ内に **var** という名前の型がないと、宣言は "暗黙的に型指定されたローカル変数宣言" となり、その型は関連付けられた初期化子の式から推論されます。暗黙的に型指定されたローカル変数宣言には、次の制限があります。

- *local-variable-declaration* に複数の *local-variable-declarator* を含めることはできません。
- *local-variable-declarator* は *local-variable-initializer* を含んでいる必要があります。
- *local-variable-initializer* は *expression* である必要があります。
- *expression* 初期化子にはコンパイル時の型が必要です。
- 初期化子 *expression* では宣言されている変数そのものを参照することはできません。

暗黙的に型指定されたローカル変数の宣言が正しくない例を次に示します。

```

var x;          // Error, no initializer to infer type from
var y = {1, 2, 3}; // Error, array initializer not permitted
var z = null;   // Error, null does not have a type
var u = x => x + 1; // Error, anonymous functions do not have a type
var v = v++;    // Error, initializer cannot refer to variable itself

```

ローカル変数の値は、式の中で *simple-name*(7.6.2を参照)を使用して取得されます。また、ローカル変数の値は *assignment*(7.17を参照)を使用して変更されます。ローカル変数は、変数の値が取得されるすべての位置において、確実に代入(5.3を参照)されている必要があります。

*local-variable-declaration* で宣言されるローカル変数のスコープは、宣言が行われているブロックになります。ローカル変数の *local-variable-declarator* より前の記述位置でそのローカル変数を参照するとエラーになります。ローカル変数のスコープ内では、同名のローカル変数や定数を宣言するとコンパイルエラーになります。

複数の変数を宣言するローカル変数宣言は、同じ型の単一の変数を複数回宣言するのと同じです。さらに、ローカル変数宣言の変数初期化子は、宣言直後に挿入される代入ステートメントと正確に対応します。

次の例を参照してください。

```

void F() {
    int x = 1, y, z = x * 2;
}

```

これは次の式とまったく同じです。

```
void F() {
    int x; x = 1;
    int y;
    int z; z = x * 2;
}
```

暗黙的に型指定されたローカル変数宣言では、宣言されているローカル変数の型は変数を初期化するために使用された式の型と同じになります。次に例を示します。

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

暗黙的に型指定された上記のローカル変数宣言は、明示的に型指定された次の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>();
```

## 8.5.2 ローカル定数宣言

*local-constant-declaration* では、1つ以上のローカル定数を宣言します。

```
local-constant-declaration:
  const type constant-declarators

constant-declarators:
  constant-declarator
  constant-declarators , constant-declarator

constant-declarator:
  identifier = constant-expression
```

*local-constant-declaration* の *type* は、宣言で導入される定数の型を指定します。この型の後には、*constant-declarator* のリストが続き、各宣言子により新規の定数が導入されます。*constant-declarator* は、定数の名前を指定する *identifier* の後に、"=" トークンと、定数の値を指定する *constant-expression* (7.19 を参照) を続けることで構成されます。

ローカル定数宣言の *type* と *constant-expression* は、定数メンバーの宣言 (10.4 を参照) に適用されるのと同じ規則に従う必要があります。

ローカル定数の値は、式の中で *simple-name* (7.6.2 を参照) を使用して取得されます。

ローカル定数のスコープは、宣言が行われているブロックになります。*constant-declarator* より前の記述位置でローカル定数を参照するとエラーになります。ローカル定数のスコープ内では、同名のローカル変数や定数を宣言するとコンパイルエラーになります。

複数の定数を宣言するローカル定数宣言は、同じ型の单一の定数を複数回宣言するのと同じです。

## 8.6 式ステートメント

*expression-statement* では、指定された式を評価します。式で計算される値があると、値は破棄されます。

```

expression-statement:
  statement-expression ;

statement-expression:
  invocation-expression
  object-creation-expression
  assignment
  post-increment-expression
  post-decrement-expression
  pre-increment-expression
  pre-decrement-expression
  await-expression

```

ステートメントとして使用できない式もあります。特に、`x + y` および `x == 1` のように、値が破棄され、計算をほとんど行わない式は、ステートメントとして許可されません。

*expression-statement* の実行により、含まれる式が評価され、*expression-statement* の終了点に制御が移ります。*expression-statement* が到達可能な場合、*expression-statement* の終了点は到達可能です。

## 8.7 選択ステートメント

選択ステートメントは、式の値に基づいて、複数の実行可能なステートメントから、1つのステートメントを選択します。

```

selection-statement:
  if-statement
  switch-statement

```

### 8.7.1 if ステートメント

`if` ステートメントは、ブール型の式の値に基づいて、実行するステートメントを選択します。

```

if-statement:
  if ( boolean-expression ) embedded-statement
  if ( boolean-expression ) embedded-statement else embedded-statement

```

`else` の部分は、先行する `if` のうち、構文規則に基づいて正しく対応する最も近い `if` と対になります。次の `if` ステートメントがあるとします。

```
if (x) if (y) F(); else G();
```

上記のコードは、次のコードと同じです。

```
if (x) {
    if (y) {
        F();
    }
    else {
        G();
    }
}
```

`if` ステートメントは次のように実行されます。

- *boolean-expression* (7.20 を参照) が評価されます。
- ブール式の結果が `true` の場合、制御は最初の埋め込みステートメントに移ります。制御がそのステートメントの終了点に達すると、`if` ステートメントの終了点に制御が移ります。

- ブール式の結果が `false` で、`else` 部分が存在する場合、制御は 2 番目の埋め込みステートメントに移ります。制御がそのステートメントの終了点に達すると、`if` ステートメントの終了点に制御が移ります。
- ブール式の結果が `false` で、`else` 部分が存在しない場合、制御は `if` ステートメントの終了点に移ります。

`if` ステートメントが到達可能で、ブール式が定数値 `false` を持たない場合、`if` ステートメントの最初の埋め込みステートメントは到達可能です。

`if` ステートメントが到達可能で、ブール式が定数値 `true` をとらない場合は、`if` ステートメントの 2 番目の埋め込みステートメントは到達可能です。

埋め込みステートメントの少なくとも 1 つの終了点が到達可能な場合は、`if` ステートメントの終了点は到達可能です。また、`if` ステートメントが到達可能で、ブール式の値が定数値 `true` でない場合、`else` 部分を持たない `if` ステートメントの終了点は到達可能です。

### 8.7.2 switch ステートメント

`switch` ステートメントは、`switch` 式の値に対応するラベルのステートメントリストを選択し、実行します。

```
switch-statement:  
    switch ( expression ) switch-block  
  
switch-block:  
    { switch-sectionsopt }  
  
switch-sections:  
    switch-section  
    switch-sections switch-section  
  
switch-section:  
    switch-labels statement-list  
  
switch-labels:  
    switch-label  
    switch-labels switch-label  
  
switch-label:  
    case constant-expression :  
    default :
```

`switch-statement` は、キーワード `switch` の後にかっこで囲まれた式 (`switch` 式) と `switch-block` を続けたものから構成されます。`switch-block` は、中かっこで囲まれた、0 以上の `switch-section` で構成されます。それぞれの `switch-section` は、`switch-label` の後に `statement-list` (8.2.1 を参照) を続けたものが 1 つ以上集まって構成されます。

`switch` ステートメントの **管理型** は、`switch` 式によって確立されます。

- `switch` 式の型が `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`bool`、`char`、`string`、または `enum-type` の場合、またはそのいずれかの型に対応する null 許容型である場合、それが `switch` ステートメントの管理型になります。

- それ以外の場合は、switch 式の型から管理型 (`sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`string`、またはこれらのいずれかの型に対応する `null` 許容型) へのユーザー定義の暗黙的な変換 (6.4 を参照) が 1 つだけ存在する必要があります。
- それ以外の場合、そのような暗黙的な変換が存在しない、または暗黙的な変換が複数存在するときに、コンパイルエラーが発生します。

それぞれの `case` ラベルの定数式は、`switch` ステートメントの管理型に暗黙的に変換される値 (0 を参照) を表す必要があります。同じ `switch` ステートメント内の 2 つ以上の `case` ラベルが同じ定数値を指定すると、コンパイルエラーが発生します。

`switch` ステートメントに設定できる `default` ラベルは 1 つだけです。

`switch` ステートメントは次のように実行されます。

- `switch` 式は、評価されて管理型に変換されます。
- 同じ `switch` ステートメント内の `case` ラベルに指定された定数の 1 つが、`switch` 式の値に等しい場合は、その一致する `case` ラベルに続くステートメントリストに制御が移ります。
- 同じ `switch` ステートメント内の `case` ラベルに指定された定数のいずれも `switch` 式の値に等しくなく、`default` ラベルが存在する場合は、その `default` ラベルに続くステートメントリストに制御が移ります。
- 同じ `switch` ステートメント内の `case` ラベルに指定された定数のいずれも `switch` 式の値に等しくなく、`default` ラベルが存在しない場合は、`switch` ステートメントの終了点に制御が移ります。

`switch` セクションのステートメントリストの終了点が到達可能な場合は、コンパイル時にエラーが発生します。これは "フォールスルーの禁止" 規則と呼ばれます。次の例を参照してください。

```
switch (i) {
    case 0:
        CaseZero();
        break;
    case 1:
        CaseOne();
        break;
    default:
        CaseOthers();
        break;
}
```

これは、`switch` セクションに到達可能な終了点がないので有効です。C および C++ とは異なり、`switch` セクションの実行は、次の `switch` セクションに "フォールスルー" できません。例を次に示します。

```
switch (i) {
    case 0:
        CaseZero();
    case 1:
        CaseZeroOrOne();
    default:
        CaseAny();
}
```

このコードはコンパイルエラーになります。switch セクションを実行した後に、別の switch セクションを実行する場合は、次のように明示的な `goto case` ステートメントまたは `goto default` ステートメントを使用する必要があります。

```
switch (i) {  
    case 0:  
        CaseZero();  
        goto case 1;  
    case 1:  
        CaseZeroOrOne();  
        goto default;  
    default:  
        CaseAny();  
        break;  
}
```

*switch-section* では複数のラベルを使用できます。次の例を参照してください。

```
switch (i) {  
    case 0:  
        CaseZero();  
        break;  
    case 1:  
        CaseOne();  
        break;  
    case 2:  
    default:  
        CaseTwo();  
        break;  
}
```

この例は有効です。この例は、"フォールスルーの禁止" 規則には違反していません。ラベル `case 2:` およびラベル `default:` は同じ *switch-section* に含まれるからです。

`break` ステートメントが誤って省略された場合も、"フォールスルーの禁止" 規則により、C および C++ で発生しやすい種類のバグを回避できます。また、この規則により、`switch` ステートメントの switch セクションは、ステートメントの動作に影響を与えることなく、自由に再配置できます。たとえば、上記の `switch` ステートメントの各セクションを前後逆に配置しても、ステートメントの動作に影響を与えません。

```
switch (i) {  
    default:  
        CaseAny();  
        break;  
    case 1:  
        CaseZeroOrOne();  
        goto default;  
    case 0:  
        CaseZero();  
        goto case 1;  
}
```

switch セクションのステートメントリストは、通常 `break`、`goto case`、または `goto default` の各ステートメントで終わりますが、ステートメントリストの終了点を到達不可能として表す構成も使用できます。たとえば、ブール式 `true` によって制御される `while` ステートメントは、終了点には到達しません。同様に、`throw` ステートメントまたは `return` ステートメントは、制御が常に別の場所に移るので、終了点には到達しません。したがって、次の例は有効です。

```

switch (i) {
    case 0:
        while (true) F();
    case 1:
        throw new ArgumentException();
    case 2:
        return;
}

```

`switch` ステートメントの管理型を `string` 型にすることもできます。次に例を示します。

```

void DoCommand(string command) {
    switch (command.ToLower()) {
        case "run":
            DoRun();
            break;
        case "save":
            DoSave();
            break;
        case "quit":
            DoQuit();
            break;
        default:
            InvalidCommand(command);
            break;
    }
}

```

`switch` ステートメントは、文字列等値演算子 (7.10.7 を参照) と同様に大文字と小文字を区別し、`switch` 式の文字列が `case` ラベル定数と完全に一致する場合にのみ、その `switch` セクションを実行します。

`switch` ステートメントの管理型が `string` の場合、値 `null` を `case` ラベル定数として使用できます。

8.5*switch-block* の `statement-list` には、宣言ステートメント ([を参照](#)) を含めることができます。`switch` ブロック内で宣言されたローカル変数または定数のスコープは、その `switch` ブロックです。

`switch` ブロック内では、式のコンテキストで使用される名前の意味は、常に同じである必要があります (7.6.2.1 を参照)。

`switch` セクションのステートメントリストは、`switch` ステートメントが到達可能であり、次のいずれか 1 つが当てはまれば到達可能です。

- `switch` 式が非定数値である場合
- `switch` 式が、`switch` セクションの `case` ラベルに一致する定数値である場合
- `switch` 式が、任意の `case` ラベルに一致しない定数値であり、`switch` セクションに `default` ラベルが含まれる場合
- `switch` セクションの `switch` ラベルが、到達可能な `goto case` ステートメントや `goto default` ステートメントによって参照される場合

`switch` ステートメントの終了点は、次のいずれか 1 つでも当てはまれば到達可能です。

- `switch` ステートメントに、`switch` ステートメントを終了する到達可能な `break` ステートメントが含まれている場合
- `switch` ステートメントが到達可能であり、`switch` 式が非定数値であり、`default` ラベルが存在しない場合

- `switch` ステートメントが到達可能であり、`switch` 式がどの `case` ラベルにも一致しない定数値であり、`default` ラベルが存在しない場合

## 8.8 繰り返しステートメント

繰り返しステートメントは、埋め込みステートメントを反復実行します。

*iteration-statement:*  
  *while-statement*  
  *do-statement*  
  *for-statement*  
  *foreach-statement*

### 8.8.1 while ステートメント

`while` ステートメントは、条件に応じて埋め込みステートメントを 0 回以上実行します。

*while-statement:*  
  *while* ( *boolean-expression* ) *embedded-statement*

`while` ステートメントは次のように実行されます。

- *boolean-expression* (7.20 を参照) が評価されます。
- ブール式の結果が `true` の場合、制御は埋め込みステートメントに移ります。制御が埋め込みステートメントの終了点に達すると(通常、`continue` ステートメントの実行から)、`while` ステートメントの先頭に制御が移ります。
- ブール式の結果が `false` の場合、制御は `while` ステートメントの終了点に移ります。

`while` ステートメントの埋め込みステートメント内で `break` ステートメント (8.9.1 を参照) を使用すると、`while` ステートメントの終了点に制御を移すことができます(つまり、埋め込みステートメントの反復処理が終了します)。また、`continue` ステートメント (8.9.2 を参照) を使用すると、埋め込みステートメントの終了点に制御を移すことができます(つまり、`while` ステートメントの次の反復処理が実行されます)。

`while` ステートメントが到達可能で、ブール式が定数値 `false` を持たない場合、`while` ステートメントの埋め込みステートメントは到達可能です。

`while` ステートメントの終了点は、次のいずれか 1 つでも当てはまれば到達可能です。

- `while` ステートメントに、`while` ステートメントを終了する到達可能な `break` ステートメントが含まれている場合
- `while` ステートメントが到達可能で、ブール式の値が定数値 `true` でない場合

### 8.8.2 do ステートメント

`do` ステートメントは、条件に応じて埋め込みステートメントを 1 回以上実行します。

*do-statement:*  
  *do* *embedded-statement* *while* ( *boolean-expression* ) ;

`do` ステートメントは次のように実行されます。

- 埋め込みのステートメントに制御が移ります。

- 制御が埋め込みステートメントの終了点に達すると (`continue` ステートメントの実行によって到達する場合もあります)、*boolean-expression* (7.20 を参照) が評価されます。ブール式の結果が `true` の場合、制御は `do` ステートメントの先頭に移ります。それ以外の場合、制御は `do` ステートメントの終了点に移ります。

`do` ステートメントの埋め込みステートメント内で `break` ステートメント (8.9.1 を参照) を使用すると、`do` ステートメントの終了点に制御を移すことができます(つまり、埋め込みステートメントの反復処理が終了します)。また、`continue` ステートメント (8.9.2 を参照) を使用すると、埋め込みステートメントの終了点に制御を移すことができます。

`do` ステートメントが到達可能な場合、`do` ステートメントの埋め込みステートメントは到達可能です。

`do` ステートメントの終了点は、次のいずれか 1 つでも当てはまれば到達可能です。

- `do` ステートメントに、`do` ステートメントを終了する到達可能な `break` ステートメントが含まれている場合
- 埋め込みステートメントの終了点が到達可能で、ブール式の値が定数値 `true` でない場合

### 8.8.3 for ステートメント

`for` ステートメントは、一連の初期化式を評価し、条件が `true` の間は埋め込みステートメントを繰り返し実行して、一連の反復式を評価します。

```

for-statement:
  for ( for-initializeropt ; for-conditionopt ; for-iteratoropt ) embedded-statement

for-initializer:
  local-variable-declaration
  statement-expression-list

for-condition:
  boolean-expression

for-iterator:
  statement-expression-list

statement-expression-list:
  statement-expression
  statement-expression-list , statement-expression

```

`for-initializer` が存在する場合、これは *local-variable-declaration* (8.5.1 を参照) か、またはコンマ区切りの *statement-expression* (8.6 を参照) のリストになります。`for-initializer` によって宣言されたローカル変数のスコープは、その変数の *local-variable-declarator* から埋め込みステートメントの終わりまでです。このスコープには、`for-condition` および `for-iterator` が含まれます。

`for-condition` が存在する場合、これは *boolean-expression* (7.20 を参照) である必要があります。

`for-iterator` が存在する場合、これはコンマ区切りの *statement-expression* (8.6 を参照) のリストになります。

`for` ステートメントは次のように実行されます。

- `for-initializer` が存在する場合、変数初期化子またはステートメント式は、記述された順序で実行されます。このステップは一度だけ実行されます。
- `for-condition` が存在する場合は評価されます。

- *for-condition* が存在しない場合、またはその評価が `true` の場合、制御は埋め込みステートメントに移ります。制御が埋め込みステートメントの終了点に到達すると(通常は `continue` ステートメントの実行から)、*for-iterator* 式がある場合は順に評価され、上のステップの *for-condition* の評価から次の反復が実行されます。
- *for-condition* が存在し、その評価が `false` の場合、制御は `for` ステートメントの終了点に移ります。

`for` ステートメントの埋め込みステートメント内で `break` ステートメント(8.9.1を参照)を使用すると、`for` ステートメントの終了点に制御を移すことができます(つまり、埋め込みステートメントの反復処理が終了します)。また、`continue` ステートメント(8.9.2を参照)を使用すると、埋め込みステートメントの終了点に制御を移すことができます(つまり、*for-iterator* が実行され、`for` ステートメントの次の反復処理が *for-condition* から実行されます)。

`for` ステートメントの埋め込みステートメントは、次のいずれか 1 つでも当てはまれば到達可能です。

- `for` ステートメントが到達可能で、*for-condition* が存在しない場合
- `for` ステートメントが到達可能で、*for-condition* が存在し、定数値 `false` を取らない場合

`for` ステートメントの終了点は、次のいずれか 1 つでも当てはまれば到達可能です。

- `for` ステートメントに、`for` ステートメントを終了する到達可能な `break` ステートメントが含まれている場合
- `for` ステートメントが到達可能で、*for-condition* が存在し、定数値 `true` を取らない場合

#### 8.8.4 foreach ステートメント

`foreach` ステートメントはコレクションの要素を列举し、コレクションの要素ごとに埋め込みステートメントを実行します。

*foreach-statement:*  
    `foreach ( local-variable-type identifier in expression ) embedded-statement`

`foreach` ステートメントの *type* および *identifier* で、ステートメントの "反復変数" を宣言します。*local-variable-type* として `var` 識別子が指定されている場合、スコープ内に `var` という名前の型がないと、反復変数は "暗黙的に型指定された反復変数" となり、その型は次に示すように `foreach` ステートメントの要素型となります。反復変数は、読み取り専用のローカル変数に対応し、スコープはその埋め込みステートメント全体です。`foreach` ステートメントの実行中は、反復変数は、現在実行中の反復のコレクション要素を表します。埋め込みステートメントが、代入または `++` 演算子と `--` 演算子で反復変数を変更しようとしたり、反復変数を `ref` パラメーターまたは `out` パラメーターとして渡そうとしたりすると、コンパイルエラーが発生します。

以下では、説明を簡単にするために、`IEnumerable`、`IEnumerator`、`IEnumerable<T>`、および `IEnumerator<T>` は、`System.Collections` および `System.Collections.Generic` 名前空間のそれぞれに対応する型を参照します。

`foreach` ステートメントのコンパイル時の処理では、最初に式の "コレクション型"、"列挙子型"、および "要素型" が決定されます。この決定は次のように処理されます。

- *expression* の *X* の型が配列型の場合、*X* から `IEnumerable` インターフェイスへの暗黙的な参照変換があります(`System.Array` がこのインターフェイスを実装するため)。コレクション型は

`IEnumerable` インターフェイス、**列挙子型** は `IEnumerator` インターフェイス、**要素型** は配列型 `X` の要素型になります。

- `expression` の型 `X` が `dynamic` である場合は、`expression` から `IEnumerable` インターフェイス (6.1.8 を参照) への暗黙的な変換があります。コレクション型は `IEnumerable` インターフェイス、**列挙子型** は `IEnumerator` インターフェイスになります。**"要素型"** は、*local-variable-type* として `var` 識別子が指定されている場合は `dynamic`、それ以外の場合は `object` になります。
- それ以外の場合は、型 `X` に適切な `GetEnumerator` メソッドがあるかどうかを判定します。
  - 型引数のない `GetEnumerator` 識別子を対象として、型 `X` でメンバー検索を実行します。メンバー検索で一致が見つからなかった場合、あいまいさが生じた場合、またはメソッドグループではない一致が生じた場合は、後で説明する方法に従って列挙可能インターフェイスを確認します。メンバー検索でメソッドグループ以外の結果が生じた場合や一致が見つからなかった場合は、警告を出すことを推奨します。
  - 得られたメソッド グループと空の引数リストを使用して、オーバーロード解決を実行します。オーバーロード解決で、該当するメソッドが見つからない、あいまいさが生じる、または 1 つの最善のメソッドが見つかったものの、そのメソッドが静的であるかパブリックでなかった場合は、後で説明する方法に従って列挙可能インターフェイスを確認します。オーバーロード解決で、あいまいでないパブリックなインスタンス メソッド以外のものが生成されるか、または適用可能なメソッドが生成されない場合は、警告を出すことを推奨します。
  - `GetEnumerator` メソッドの戻り値の型 `E` が、クラス、構造体、またはインターフェイス型でない場合、エラーが発生し、以降の手順は行われません。
  - 型引数のない `Current` 識別子を対象として、`E` でメンバー検索が実行されます。メンバー検索で一致が見つからない、エラーが発生する、または結果が読み取りを許可されているパブリック インスタンス プロパティ以外である場合、エラーが発生し、以降の手順は行われません。
  - 型引数のない `MoveNext` 識別子を対象として、`E` でメンバー検索が実行されます。メンバー検索で一致が見つからない、エラーが発生する、または結果がメソッド グループ以外である場合、エラーが発生し、以降の手順は行われません。
  - メソッド グループと空の引数リストを使用して、オーバーロード解決が実行されます。オーバーロード解決で適切なメソッドが見つからない、あいまいさが生じる、または 1 つの最善のメソッドが見つかったものの、そのメソッドが静的であるかパブリックでなかった場合、あるいは戻り値の型が `bool` でない場合は、エラーが発生し、以降の手順は行われません。
  - **"コレクション型"** は `X`、**"列挙子型"** は `E`、**"要素型"** は `current` プロパティの型になります。
- それ以外の場合は、列挙可能なインターフェイスを確認します。
  - `X` から `IEnumerable<Ti>` への暗黙の変換がある、すべての型 `Ti` の中に、一意の型 `T` があり、`T` が `dynamic` ではなく、その他のすべての `Ti` に対して、`IEnumerable<T>` から `IEnumerable<Ti>` への暗黙の変換がある場合、**"コレクション型"** がインターフェイス `IEnumerable<T>` となり、**"列挙子型"** はインターフェイス `IEnumerator<T>` となり、**"要素型"** は `T` となります。
  - それ以外の場合、そのような型 `T` が複数存在すると、エラーが発生し、以降の手順は行われません。

- それ以外の場合、`x` から `System.Collections.IEnumerable` インターフェイスへの暗黙的な変換がある場合、コレクション型はこのインターフェイス、列挙子型はインターフェイス `System.Collections.IEnumerator`、要素型は `object` になります。
- それ以外の場合にはエラーが発生し、以降の手順は行われません。

上記の手順が成功すると、コレクション型 `C`、列挙子型 `E`、要素型 `T` が明確に生成されます。foreach ステートメントは次の形式になります。

```
foreach (V v in x) embedded-statement
```

これは次のように展開されます。

```
{
    E e = ((C)(x)).GetEnumerator();
    try {
        while (e.MoveNext()) {
            V v = (V)(T)e.Current;
            embedded-statement
        }
    }
    finally {
        ... // Dispose e
    }
}
```

変数 `e` は、式 `x`、埋め込みステートメント、またはプログラムのその他のソース コードからは、参照もアクセスできません。変数 `v` は、埋め込みステートメントでは読み取り専用です。`T`(要素型) から `V`(foreach ステートメントの *local-variable-type*) への明示的な変換(6.2 を参照)がない場合は、エラーが発生し、以降の手順は行われません。`x` が `null` 値を持つ場合、実行時に `System.NullReferenceException` がスローされます。

実装では、動作に上記の展開との整合性がある限り、パフォーマンス上の理由などから、異なる方法で foreach ステートメントを実装してもかまいません。

`while` ループ内への `v` の配置は、`embedded-statement` 内の匿名関数でどのようにキャプチャされるかに大きく影響します。

次に例を示します。

```
int[] values = { 7, 9, 13 };
Action f = null;
foreach (var value in values)
{
    if (f == null) f = () => Console.WriteLine("First value: " + value);
}
f();
```

`v` が `while` ループの外側で宣言された場合、すべてのイテレーションで共有され、`for` ループの後の値が最終値 `13` になります。これが `f` の呼び出しによって出力される値です。代わりに、各イテレーションには独自の変数 `v` があるため、最初のイテレーションで `f` によってキャプチャされた `v` は引き続き値 `7` を保持し、この値が出力されます(メモ: 旧バージョンの C# では `while` ループの外側で `v` を宣言しました)。

`finally` ブロックの本体は、次の手順に従って構築されます。

- `E` から `System.IDisposable` インターフェイスへの暗黙的な変換がある場合、次の処理が実行されます。

- `E` が `null` 非許容型である場合、`finally` 句は次の意味と同じになるように拡張されます。

```
finally {
    ((System.IDisposable)e).Dispose();
}
```

- それ以外の場合、`finally` 句は次の意味と同じになるように展開されます。

```
finally {
    if (e != null) ((System.IDisposable)e).Dispose();
}
```

ただし、`E` が値型または値型にインスタンス化される型パラメーターである場合は、`e` を `System.IDisposable` にキャストしてもボックス化は発生しません。

- それ以外の場合、`E` がシール型であれば、`finally` 句は空のブロックに展開されます。

```
finally {
}
```

- それ以外の場合、`finally` 句は次のように展開されます。

```
finally {
    System.IDisposable d = e as System.IDisposable;
    if (d != null) d.Dispose();
}
```

ローカル変数 `d` は、ユーザー コードから参照もアクセスもできません。特に、`finally` ブロックを スコープに含んでいる他の変数と衝突することはありません。

`foreach` が配列の要素を走査する順序は、次のように定義されます。1 次元配列の場合、要素はインデックス 0 から始まってインデックス `Length - 1` で終わるインデックスの昇順に走査されます。多次元配列の場合、要素は、最初に右端の次元のインデックスが増加し、次にその左側の次元のインデックスが増加し、さらにその左側の次元のインデックスが増加する、という順序で走査されます。

2 次元配列のそれぞれの値を要素順にプリント アウトするコード例を次に示します。

```
using System;
class Test
{
    static void Main() {
        double[,] values = {
            {1.2, 2.3, 3.4, 4.5},
            {5.6, 6.7, 7.8, 8.9}
        };
        foreach (double elementValue in values)
            Console.Write("{0} ", elementValue);
        Console.WriteLine();
    }
}
```

生成される出力は次のとおりです。

```
1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9
```

次に例を示します。

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

`n` の型は、`numbers` の要素型である `int` と推論されます。

## 8.9 ジャンプステートメント

ジャンプステートメントは、無条件に制御を移します。

```
jump-statement:
    break-statement
    continue-statement
    goto-statement
    return-statement
    throw-statement
```

ジャンプステートメントが制御を移す先の場所は、ジャンプステートメントの "ターゲット" と呼ばれます。

ジャンプステートメントがブロック内で発生し、ジャンプステートメントのターゲットがそのブロック外にある場合、ジャンプステートメントはブロックを終了すると言います。ジャンプステートメントは、ブロック外に制御を飛び出させることはできますが、ブロック内に制御を飛び込ませることはできません。

ジャンプステートメントの実行は、間に `try` ステートメントがあると複雑になります。ジャンプステートメントの間に `try` ステートメントがない場合は、無条件に制御をジャンプステートメントからターゲットに移します。間に `try` ステートメントがある場合、実行はこれよりも複雑です。ジャンプステートメントが `finally` ブロックを持つ1つ以上の `try` ブロックを終了する場合、制御は最初に最も内側の `try` ステートメントの `finally` ブロックに移ります。制御が `finally` ブロックの終了点に達すると、その外側の `try` ステートメントの `finally` ブロックに制御が移されます。間にあるすべての `try` ステートメントの `finally` ブロックが実行されるまで、このプロセスが繰り返されます。

次に例を示します。

```
using System;
class Test
{
    static void Main() {
        while (true) {
            try {
                try {
                    Console.WriteLine("Before break");
                    break;
                }
                finally {
                    Console.WriteLine("Innermost finally block");
                }
            }
            finally {
                Console.WriteLine("Outermost finally block");
            }
        }
        Console.WriteLine("After break");
    }
}
```

制御がジャンプステートメントのターゲットに移る前に、2つの `try` ステートメントに関連付けられた `finally` ブロックが実行されます。

生成される出力は次のとおりです。

```
Before break
Innermost finally block
Outermost finally block
After break
```

### 8.9.1 break ステートメント

**break** ステートメントは、すぐ外側にある **switch**、**while**、**do**、**for**、**foreach** の各ステートメントを終了します。

```
break-statement:
    break ;
```

**break** ステートメントのターゲットは、すぐ外側にある **switch**、**while**、**do**、**for**、または **foreach** の各ステートメントの終了点です。**break** ステートメントの外側に、**switch**、**while**、**do**、**for**、または **foreach** の各ステートメントがない場合は、コンパイル時のエラーが発生します。

複数の **switch**、**while**、**do**、**for**、または **foreach** の各ステートメントが互いに入れ子になっている場合、**break** ステートメントは最も内側のステートメントにだけ適用されます。複数の入れ子レベルを越えて制御を移すには、**goto** ステートメント (8.9.3 を参照) を使用する必要があります。

**break** ステートメントは **finally** ブロック (8.10 を参照) を終了できません。**break** ステートメントが **finally** ブロック内で実行される場合、**break** ステートメントのターゲットは同じ **finally** ブロック内に存在する必要があります。それ以外の場合は、コンパイル時のエラーが発生します。

**break** ステートメントは次のように実行されます。

- **break** ステートメントが **finally** ブロックを持つ1つ以上の **try** ブロックを終了する場合、制御は最初に最も内側の **try** ステートメントの **finally** ブロックに移ります。制御が **finally** ブロックの終了点に達すると、その外側の **try** ステートメントの **finally** ブロックに制御が移されます。間にあるすべての **try** ステートメントの **finally** ブロックが実行されるまで、このプロセスが繰り返されます。
- **break** ステートメントのターゲットに制御が移ります。

**break** ステートメントは無条件に制御を他の場所に移すため、**break** ステートメントの終了点は常に到達不可能です。

### 8.9.2 continue ステートメント

**continue** ステートメントは、すぐ外側にある **while**、**do**、**for**、**foreach** の各ステートメントを新たに繰り返します。

```
continue-statement:
    continue ;
```

**continue** ステートメントのターゲットは、すぐ外側にある **while**、**do**、**for**、または **foreach** の各ステートメントの埋め込みステートメントの終了点です。**continue** ステートメントの外側に、**while**、**do**、**for**、または **foreach** の各ステートメントがない場合は、コンパイル時のエラーが発生します。

複数の **while**、**do**、**for**、または **foreach** の各ステートメントが互いに入れ子になっている場合、**continue** ステートメントは最も内側のステートメントにだけ適用されます。複数の入れ子レベルを越えて制御を移すには、**goto** ステートメント (8.9.3 を参照) を使用する必要があります。

`continue` ステートメントは `finally` ブロック (8.10 を参照) を終了できません。`continue` ステートメントが `finally` ブロック内で実行される場合、`continue` ステートメントのターゲットは同じ `finally` ブロック内に存在する必要があります。それ以外の場合は、コンパイル時のエラーが発生します。

`continue` ステートメントは次のように実行されます。

- `continue` ステートメントが `finally` ブロックを持つ 1 つ以上の `try` ブロックを終了する場合、制御は最初に最も内側の `try` ステートメントの `finally` ブロックに移ります。制御が `finally` ブロックの終了点に達すると、その外側の `try` ステートメントの `finally` ブロックに制御が移されます。間にあるすべての `try` ステートメントの `finally` ブロックが実行されるまで、このプロセスが繰り返されます。

- `continue` ステートメントのターゲットに制御が移ります。

`continue` ステートメントは無条件に制御を他の場所に移すため、`continue` ステートメントの終了点は常に到達不可能です。

### 8.9.3 goto ステートメント

`goto` ステートメントは、ラベルでマークされたステートメントに制御を移します。

```
goto-statement:
    goto identifier ;
    goto case constant-expression ;
    goto default ;
```

`goto identifier` ステートメントのターゲットは、指定のラベルが付いたステートメントです。指定された名前のラベルが現在の関数メンバー内に存在しない場合、または `goto` ステートメントがラベルのスコープ内に存在しない場合は、コンパイルエラーが発生します。この規則により、`goto` ステートメントを使って、入れ子になったスコープの "外側" には制御を移すことができますが、入れ子になったスコープの "内側" には移すことができません。次に例を示します。

```
using System;
class Test
{
    static void Main(string[] args) {
        string[,] table = {
            {"Red", "Blue", "Green"},
            {"Monday", "Wednesday", "Friday"}
        };
        foreach (string str in args) {
            int row, colm;
            for (row = 0; row <= 1; ++row)
                for (colm = 0; colm <= 2; ++colm)
                    if (str == table[row, colm])
                        goto done;
            Console.WriteLine("{0} not found", str);
            continue;
        done:
            Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);
        }
    }
}
```

`goto` ステートメントを使って、入れ子になったスコープの外側に制御を移します。

**goto case** ステートメントのターゲットは、指定の定数値を持つ **case** ラベルを含む、すぐ外側の **switch** ステートメント (8.7.2 を参照) のステートメントリストです。**goto case** ステートメントの外側に **switch** ステートメントがない場合、*constant-expression* がすぐ外側の **switch** ステートメントの管理型に暗黙的に変換可能 (0 を参照) でない場合、またはすぐ外側の **switch** ステートメントに、指定された定数値の **case** ラベルが含まれない場合は、コンパイル エラーが発生します。

**goto default** ステートメントのターゲットは、**default** ラベルを含む、すぐ外側の **switch** ステートメント (8.7.2 を参照) のステートメントリストです。**goto default** ステートメントの外側に **switch** ステートメントがない場合、またはすぐ外側の **switch** ステートメントに **default** ラベルが含まれない場合は、コンパイル時のエラーが発生します。

**goto** ステートメントは **finally** ブロック (8.10 を参照) を終了できません。**goto** ステートメントが **finally** ブロック内で実行される場合、**goto** ステートメントのターゲットは同じ **finally** ブロック内に存在する必要があります。それ以外の場合は、コンパイル時のエラーが発生します。

**goto** ステートメントは次のように実行されます。

- **goto** ステートメントが **finally** ブロックを持つ 1 つ以上の **try** ブロックを終了する場合、制御は最初に最も内側の **try** ステートメントの **finally** ブロックに移ります。制御が **finally** ブロックの終了点に達すると、その外側の **try** ステートメントの **finally** ブロックに制御が移されます。間にあるすべての **try** ステートメントの **finally** ブロックが実行されるまで、このプロセスが繰り返されます。
- **goto** ステートメントのターゲットに制御が移ります。

**goto** ステートメントは無条件に制御を他の場所に移すため、**goto** ステートメントの終了点は常に到達不可能です。

#### 8.9.4 return ステートメント

**return** ステートメントは、**return** ステートメントが存在する関数の現在の呼び出し元に制御を返します。

```
return-statement:
    return expressionopt ;
```

式を指定しない **return** ステートメントを使用できるのは、値を計算しない関数メンバー内、つまり、結果の型 (10.6.10 を参照) が **void** のメソッド、プロパティまたはインデクサーの **set** アクセサー、イベントの **add** アクセサーおよび **remove** アクセサー、インスタンス コンストラクター、静的コンストラクター、またはデストラクターだけです。

式を指定した **return** ステートメントを使用できるのは、値を計算する関数メンバー内、つまり **void** 以外の結果の型を持つメソッド、プロパティまたはインデクサーの **get** アクセサー、またはユーザー定義の演算子だけです。式の型から、包含する関数メンバーの戻り値の型への暗黙的な変換 (0 を参照) が存在する必要があります。

**return** ステートメントは、匿名関数式の本体でも使用でき (7.15 を参照)、それらの関数に対してどの変換が存在するかの判断に関与します。

**return** ステートメントを **finally** ブロック (8.10 を参照) で実行すると、コンパイル エラーが発生します。

**return** ステートメントは次のように実行されます。

- `return` ステートメントに式を指定する場合は、式が評価されます。結果の値は、暗黙的な変換によって、包含する関数メンバーの戻り値の型に変換されます。変換結果が、関数によって生成される結果値になります。
- `return` ステートメントの外側に `finally` ブロックを持つ 1 つ以上の `try` ブロックまたは `catch` ブロックがある場合、制御は最初に最も内側の `try` ステートメントの `finally` ブロックに移ります。制御が `finally` ブロックの終了点に達すると、その外側の `try` ステートメントの `finally` ブロックに制御が移されます。外側のすべての `try` ステートメントの `finally` ブロックが実行されるまで、このプロセスが繰り返されます。
- 包含する関数が非同期関数でない場合、制御は包含する関数の呼び出し元に結果値(存在する場合)と共に返されます。
- 包含する関数が非同期関数である場合、制御は現在の呼び出し元に返され、結果値がある場合は、戻りタスクに記録されます(エラー! 参照元が見つかりません。を参照)。

`return` ステートメントは無条件に制御を他の場所に移すため、`return` ステートメントの終了点は常に到達不可能です。

### 8.9.5 throw ステートメント

`throw` ステートメントは例外をスローします。

*throw-statement:*  
    *throw* *expression<sub>opt</sub>* ;

式を持つ `throw` ステートメントは、式を評価することで生成された値をスローします。この式は、`System.Exception` のクラス型の値を示す必要があります。これは `System.Exception` から派生するクラス型か、その実質的な基底クラスが `System.Exception` から派生する型パラメーターのクラス型です。式の評価により `null` が生成される場合は、代わりに `System.NullReferenceException` がスローされます。

式を持たない `throw` ステートメントを使用できるのは、`catch` ブロック内だけです。この場合、ステートメントは、`catch` ブロックによって現在処理されている例外を再スローします。

`throw` ステートメントは無条件に制御を他の場所に移すため、`throw` ステートメントの終了点は常に到達不可能です。

例外 `E` がスローされると、その例外を処理できる `try` ステートメントの最初の `catch` 句に制御が移ります。例外をスローするポイントから適切な例外ハンドラーに制御を移すポイントまでに発生するプロセスは、例外の伝達と呼ばれます。例外の反映は、例外に一致する `catch` 句が見つかるまで次のステップを繰り返し評価する処理で構成されます。この説明で使用するスロー ポイントとは、初期状態で例外がスローされる場所です。

- 現在の関数メンバーで、スロー ポイントの外側にある各 `try` ステートメントが実行されます。最も内側の `try` ステートメントから最も外側の `try` ステートメントまでのステートメント `s` ごとに、次のステップが評価されます。
  - `s` の `try` ブロックの内側にスロー ポイントがある場合、および `s` に 1 つ以上の `catch` 句がある場合、`catch` 句は出現する順に実行されて適切な例外ハンドラーを見つけます。`E` のランタイム型が `T` から派生するように、例外の種類 `T`(または実行時に例外の種類 `T` を示す型パラメーター)を指定している最初の `catch` 句が、一致と見なされます。汎用的な `catch` 句(8.10

を参照)は、あらゆる例外の種類に一致すると見なされます。一致する **catch** 句が見つかると、その **catch** 句のブロックに制御を移すことで例外の反映は完了します。

- それ以外の場合、**s** の **try** ブロックまたは **catch** ブロックの内側にスロー ポイントがあり、かつ **s** に **finally** ブロックがあると、制御は **finally** ブロックに移ります。**finally** ブロックが別の例外をスローする場合、現在の例外の処理は終了します。それ以外の場合は、制御が **finally** ブロックの終了点に到達すると、現在の例外の処理は継続されます。
- 例外ハンドラーが現在の関数呼び出し内に見つからなかった場合、関数呼び出しは終了し、以下のいずれかが発生します。
  - 現在の関数が非同期ではない場合は、関数メンバーを呼び出したステートメントに対応するスロー ポイントを持つ関数の呼び出し元について、上のステップを繰り返します。
  - 現在の関数が非同期でタスクを返す場合、戻りタスクに例外が記録され、違反状態または取り消し済み状態になります(エラー! 参照元が見つかりません。 を参照)。
  - 現在の関数が非同期で **void** を返す場合、現在のスレッドの同期コンテキストが通知を受け取ります(エラー! 参照元が見つかりません。 を参照)。
- 例外処理により現在のスレッドのすべての関数メンバー呼び出しが終了する、つまりスレッドに例外ハンドラーがなくなると、スレッド自体が終了します。このような終了の影響は、実装で定義されます。

## 8.10 try ステートメント

**try** ステートメントは、ブロックの実行中に発生する例外をキャッチするしくみを提供します。さらに、**try** ステートメントを使用して、制御が **try** ステートメントを離れるときに常に実行されるコードのブロックを指定することもできます。

```

try-statement:
  try block catch-clauses
  try block finally-clause
  try block catch-clauses finally-clause

catch-clauses:
  specific-catch-clauses general-catch-clauseopt
  specific-catch-clausesopt general-catch-clause

specific-catch-clauses:
  specific-catch-clause
  specific-catch-clauses specific-catch-clause

specific-catch-clause:
  catch ( type identifieropt ) block

general-catch-clause:
  catch block

finally-clause:
  finally block

```

**try** ステートメントには、次の 3 種類の形式があります。

- **try** ブロックの後に 1 つ以上の **catch** ブロックを続ける形式
- **try** ブロックの後に **finally** ブロックを続ける形式

- **try** ブロックの後に 1 つ以上の **catch** ブロック、その後に **finally** ブロックを続ける形式

**catch** 句で型を指定した場合、その型は **System.Exception** である必要があります。これは **System.Exception** から派生した型か、実質的な基底クラスが **System.Exception** から派生している型パラメーターによる型です。

**catch** 句が *class-type* と *identifier* の両方を指定する場合は、指定された名前と型の "例外変数" が宣言されます。例外変数はローカル変数に対応し、そのスコープは **catch** ブロック全体です。**catch** ブロックの実行中は、例外変数は現在処理中の例外を表します。確実な代入のチェックのために、例外変数は全体のスコープに確実に代入されたと見なされます。

**catch** 句に例外変数名が含まれない場合は、**catch** ブロック内の例外オブジェクトにはアクセスできません。

例外の型や例外変数名を指定しない **catch** 句は、汎用的な **catch** 句と呼ばれます。**try** ステートメントに設定できる汎用的な **catch** 句は 1 つだけで、存在する場合はそれが最後の **catch** 句である必要があります。

プログラミング言語によっては、**System.Exception** の派生オブジェクトとして表現できない例外をサポートする場合があります。このような例外は、C# コードでは生成されません。汎用的な **catch** 句を使用すると、このような例外をキャッチできます。このように、汎用的な **catch** 句は、他の言語の例外もキャッチできるという点で、**System.Exception** 型を指定する **catch** 句とは意味が異なります。

例外ハンドラーを探すために、**catch** 句は文法順に検査されます。**catch** 句の指定する型が、同じ **try** ステートメントのそれ以前の **catch** 句で指定されたのと同じ型か、その型から派生している場合は、コンパイルエラーが発生します。この制限がない場合は、到達不可能な **catch** 句を記述できることになります。

**catch** ブロック内では、式を持たない **throw** ステートメント (8.9.5 を参照) を使用することで、その **catch** ブロックでキャッチされた例外を再スローできます。例外変数に代入しても、再スローされる例外は変更されません。

次に例を示します。

```
using System;
class Test
{
    static void F() {
        try {
            G();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in F: " + e.Message);
            e = new Exception("F");
            throw; // re-throw
        }
    }
    static void G() {
        throw new Exception("G");
    }
}
```

```

static void Main() {
    try {
        F();
    }
    catch (Exception e) {
        Console.WriteLine("Exception in Main: " + e.Message);
    }
}

```

メソッド `F` は例外をキャッチし、診断情報をコンソールに出力します。次に、例外変数を変更し、例外を再スローします。再スローされる例外が元の例外なので、出力は次のようにになります。

```

Exception in F: G
Exception in Main: G

```

最初の `catch` ブロックが、現在の例外を再スローする代わりに `e` をスローしていた場合、出力は次のようにになります。

```

Exception in F: G
Exception in Main: F

```

`break`、`continue`、`goto` の各ステートメントで `finally` ブロックの外側に制御を移すと、コンパイル時のエラーが発生します。`break`、`continue`、または `goto` の各ステートメントが `finally` ブロック内で実行される場合、ステートメントのターゲットは同じ `finally` ブロック内に存在する必要があります。それ以外の場合は、コンパイル時のエラーが発生します。

`return` ステートメントを `finally` ブロックで実行すると、コンパイルエラーが発生します。

`try` ステートメントは次のように実行されます。

- `try` ブロックに制御が移ります。
- `try` ブロックの終了点に制御が到達すると、次の処理が行われます。
  - `try` ステートメントに `finally` ブロックがある場合は、`finally` ブロックが実行されます。
  - 制御は `try` ステートメントの終了点に移ります。
- `try` ブロックの実行中に例外が `try` ステートメントに反映されると、次の処理が行われます。
  - `catch` 句がある場合は出現順に検査され、適切な例外ハンドラーを見つけます。例外の種類または例外の種類の基本型を指定する最初の `catch` 句が、一致と見なされます。汎用的な `catch` 句は、あらゆる例外の種類に一致すると見なされます。一致する `catch` 句が見つかると、次の処理が行われます。
    - 一致する `catch` 句が例外変数を宣言する場合、例外オブジェクトは例外変数に代入されます。
    - `catch` ブロックに制御が移ります。
  - `catch` ブロックの終了点に制御が到達すると、次の処理が行われます。
    - `try` ステートメントに `finally` ブロックがある場合は、`finally` ブロックが実行されます。
    - 制御は `try` ステートメントの終了点に移ります。
- `catch` ブロックの実行中に例外が `try` ステートメントに反映されると、次の処理が行われます。

- `try` ステートメントに `finally` ブロックがある場合は、`finally` ブロックが実行されます。
- 次の外側の `try` ステートメントに例外が伝達されます。
- `try` ステートメントに `catch` 句がない場合、または `catch` 句が例外に一致しない場合は、次の処理が行われます。
  - `try` ステートメントに `finally` ブロックがある場合は、`finally` ブロックが実行されます。
  - 次の外側の `try` ステートメントに例外が伝達されます。

制御が `try` ステートメントを離れるときには、`finally` ブロックのステートメントが必ず実行されます。これは、制御の移動が通常の実行の結果である場合にも、`break`、`continue`、`goto`、`return` の各ステートメントの実行の結果である場合にも、または `try` ステートメントからの例外の伝達の結果である場合にも当てはまります。

`finally` ブロックの実行中に例外がスローされ、同じ `finally` ブロック内でキャッチされない場合、例外は次の外側の `try` ステートメントに反映されます。別の例外が伝達中であった場合、その例外は失われます。例外の伝達のプロセスの詳細については、`throw` ステートメントの説明 (8.9.5) を参照してください。

`try` ステートメントが到達可能な場合、`try` ステートメントの `try` ブロックは到達可能です。

`try` ステートメントが到達可能な場合、`try` ステートメントの `catch` ブロックは到達可能です。

`try` ステートメントが到達可能な場合、`try` ステートメントの `finally` ブロックは到達可能です。

次の両方が当てはまる場合、`try` ステートメントの終了点は到達可能です。

- `try` ブロックの終了点が到達可能であるか、または `catch` ブロックの少なくとも 1 つの終了点が到達可能である場合。
- `finally` ブロックが存在している場合は、`finally` ブロックの終了点が到達可能な場合。

## 8.11 checked ステートメントおよび unchecked ステートメント

`checked` ステートメントと `unchecked` ステートメントは、整数型の算術演算および変換に対する "オーバーフロー チェック コンテキスト" を制御するために使用します。

```
checked-statement:  
  checked block  
  
unchecked-statement:  
  unchecked block
```

`checked` ステートメントを使うと、`block` 内のすべての式が `checked` コンテキスト内で評価されます。`unchecked` ステートメントを使うと、`block` 内のすべての式が `unchecked` コンテキスト内で評価されます。

`checked` ステートメントおよび `unchecked` ステートメントは、式ではなくブロックが処理される点を除いて、`checked` 演算子および `unchecked` 演算子 (7.6.12 を参照) とまったく同じです。

## 8.12 lock ステートメント

`lock` ステートメントは、指定のオブジェクトに対する相互排他ロックを取得し、ステートメントを実行してから、ロックを解放します。

*lock-statement:*  
`lock ( expression ) embedded-statement`

`lock` ステートメントの式は、*reference-type* とわかっている型の値を示す必要があります。暗黙のボックス化変換 (6.1.7 を参照) は、`lock` ステートメントの式に対しては実行されません。このため、式に *value-type* の値を指定するとコンパイル エラーになります。

`lock` ステートメントは次の形式になります。

`lock (x) ...`

ここで、`x` は *reference-type* の式を示します。このステートメントは、以下とまったく同じです。

```
bool __lockwasTaken = false;
try {
    System.Threading.Monitor.Enter(x, ref __lockwasTaken);
    ...
}
finally {
    if (__lockwasTaken) System.Threading.Monitor.Exit(x);
}
```

ただし、`x` が評価されるのは 1 回だけです。

相互排他ロックが保持されている間、同じ実行スレッドで実行中のコードは、ロックを取得および解放できます。ただし、他のスレッドで実行中のコードは、ロックが解放されない限りロックを取得できません。

静的データへのアクセスを同期するために `System.Type` オブジェクトをロックすることは推奨しません。同じ型で他のコードがロックされていると、デッドロックが発生します。プライベートな静的オブジェクトをロックすることにより静的データへのアクセスを同期する方がより適切です。次に例を示します。

```
class Cache
{
    private static readonly object synchronizationObject = new object();
    public static void Add(object x) {
        lock (Cache.synchronizationObject) {
            ...
        }
    }
    public static void Remove(object x) {
        lock (Cache.synchronizationObject) {
            ...
        }
    }
}
```

## 8.13 using ステートメント

`using` ステートメントは、1 つ以上のリソースを取得し、ステートメントを実行してから、リソースを破棄します。

*using-statement:*  
`using ( resource-acquisition ) embedded-statement`

```
resource-acquisition:
  local-variable-declaration
  expression
```

"リソース" は、`System.IDisposable` を実装するクラスまたは構造体です。`System.IDisposable` には、`Dispose` という、パラメーターを取らないメソッドが 1 つ含まれます。リソースを使用しているコードは、`Dispose` を呼び出して、リソースが不要になったことを示すことができます。`Dispose` が呼び出されない場合は、ガベージコレクションの結果として最終的には自動的に破棄されます。

*resource-acquisition* の形式が *local-variable-declaration* である場合、*local-variable-declaration* の型は `dynamic` か、または暗黙的に `System.IDisposable` に変換できる型である必要があります。

*resource-acquisition* の形式が *expression* である場合、この式は `System.IDisposable` に暗黙的に変換できる必要があります。

*resource-acquisition* で宣言されたローカル変数は読み取り専用であり、初期化子を含む必要があります。埋め込みステートメントが、代入または `++` 演算子と `--` 演算子でローカル変数を変更しようとしたり、ローカル変数のアドレスを取得しようとしたり、ローカル変数を `ref` パラメーターまたは `out` パラメーターとして渡そうとしたりすると、コンパイルエラーが発生します。

`using` ステートメントは、取得、使用、および破棄という 3 つの部分に変換されます。リソースの使用は暗黙的に、`finally` 句を含む `try` ステートメントの内側になります。この `finally` 句はリソースを破棄します。`null` リソースが取得されると、`Dispose`への呼び出しは行われず、例外もスローされません。`dynamic` 型のリソースは、使用の前と破棄の前に正しく変換が行われるようにするため、取得中に暗黙的な動的変換 (6.1.8 を参照) によって動的に `IDisposable` に変換されます。

`using` ステートメントは次の形式になります。

```
using (ResourceType resource = expression) statement
```

これは、次のいずれかに当てはまります。`ResourceType` が `null` 非許容の値型のときは、次のように展開されます。

```
{
  ResourceType resource = expression;
  try {
    statement;
  }
  finally {
    ((IDisposable)resource).Dispose();
  }
}
```

それ以外の場合、`ResourceType` が `null` 許容の値型または `dynamic` 以外の参照型のときは、次のように展開されます。

```
{
  ResourceType resource = expression;
  try {
    statement;
  }
  finally {
    IDisposable d = (IDisposable)resource;
    if (resource != null) d.Dispose();
  }
}
```

それ以外の場合、`ResourceType` が `dynamic` のときは、次のように展開されます。

```
{
    ResourceType resource = expression;
    IDisposable d = resource;
    try {
        statement;
    }
    finally {
        if (d != null) d.Dispose();
    }
}
```

どの展開においても、`resource` 変数は埋め込みステートメント内で読み取り専用であり、`d` 変数は埋め込みステートメントからアクセスすることも参照することもできません。

実装では、動作に上記の展開との整合性がある限り、パフォーマンス上の理由などから、異なる方法で `using` ステートメントを実装してもかまいません。

`using` ステートメントは次の形式になります。

```
using (expression) statement
```

これにも同様に、3つの展開があります。この場合、`ResourceType` は、`expression` がある場合は、暗黙的にそのコンパイル時の型です。それ以外の場合、インターフェイス `IDisposable` が `ResourceType` として使用されます。`resource` 変数は埋め込みステートメントからはアクセスも参照もできません。

`resource-acquisition` が *local-variable-declaration* の形式である場合は、指定された型の複数のリソースを取得できます。`using` ステートメントは次の形式になります。

```
using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement
```

このステートメントは、次に示す入れ子になった一連の `using` ステートメントとまったく同じです。

```
using (ResourceType r1 = e1)
    using (ResourceType r2 = e2)
        ...
            using (ResourceType rN = eN)
                statement
```

次のコード例では、`log.txt` というファイルを作成し、2行のテキストをファイルに書き込みます。次に、読み取り用に同じファイルを開き、ファイルに含まれているテキスト行をコンソールにコピーします。

```
using System;
using System.IO;
class Test
{
    static void Main() {
        using (TextWriter w = File.CreateText("log.txt")) {
            w.WriteLine("This is line one");
            w.WriteLine("This is line two");
        }

        using (TextReader r = File.OpenText("log.txt")) {
            string s;
            while ((s = r.ReadLine()) != null) {
                Console.WriteLine(s);
            }
        }
    }
}
```

`TextWriter` クラスと `TextReader` クラスは `IDisposable` インターフェイスを実装するため、このコード例で `using` ステートメントを使用すると、書き込み操作または読み取り操作の後に、使用したファイルを正しく閉じることができます。

## 8.14 yield ステートメント

`yield` ステートメントは、反復子ブロック (8.2 を参照) の中で、列挙子オブジェクト (10.14.4 を参照) または列挙可能なオブジェクト (10.14.5 を参照) に値を生成したり、反復処理の終了を示したりするために使用します。

*yield-statement:*

```
yield return expression ;
yield break ;
```

`yield` は予約語ではなく、`return` キーワードまたは `break` キーワードの直前で使用する場合にのみ特別な意味を持ちます。その他のコンテキストでは、`yield` は識別子として使用できます。

`yield` ステートメントを使用できる場所については、以下で説明するように、いくつかの制限があります。

- `yield` ステートメント (いずれかの形式) を *method-body*、*operator-body*、または *accessor-body* の外側で実行すると、コンパイルエラーになります。
- `yield` ステートメント (いずれかの形式) を匿名関数の内側で実行すると、コンパイルエラーになります。
- `yield` ステートメント (すべての形式) を `try` ステートメントの `finally` 句内で実行すると、コンパイル時のエラーが発生します。
- `yield return` ステートメントを `catch` 句を含む `try` ステートメント内で実行すると、コンパイル時のエラーが発生します。

次の例は、`yield` ステートメントが有効な場合と無効な場合を示します。

```
delegate IEnumerable<int> D();
IEnumerator<int> GetEnumerator() {
    try {
        yield return 1;      // Ok
        yield break;        // Ok
    }
    finally {
        yield return 2;      // Error, yield in finally
        yield break;        // Error, yield in finally
    }
    try {
        yield return 3;      // Error, yield return in try...catch
        yield break;        // Ok
    }
    catch {
        yield return 4;      // Error, yield return in try...catch
        yield break;        // Ok
    }
    D d = delegate {
        yield return 5;      // Error, yield in an anonymous function
    };
}
```

```
int MyMethod() {
    yield return 1;           // Error, wrong return type for an iterator block
}
```

`yield return` ステートメントの式の型から反復子の `yield` 型 (10.14.3 を参照) への暗黙の型変換 (6.1 を参照) が存在する必要があります。

`yield return` ステートメントは次のように実行されます。

- ステートメントに指定されている式を評価し、暗黙的に `yield` 型に変換し、列挙子オブジェクトの `Current` プロパティに代入します。
- 反復子ブロックの実行を中断します。`yield return` ステートメントが 1 つ以上の `try` ブロック内にある場合、この時点では関連付けられた `finally` ブロックは実行されません。
- 列挙子オブジェクトの `MoveNext` メソッドが呼び出し元に `true` を返して、列挙子オブジェクトが正常に次の項目に進んだことを示します。

列挙子オブジェクトの `MoveNext` メソッドの次の呼び出しで、前回中断された位置から反復子ブロックの実行が再開されます。

`yield break` ステートメントは次のように実行されます。

- `yield break` ステートメントの外側に `finally` ブロックを持つ 1 つ以上の `try` ブロックがある場合、制御は最初に最も内側の `try` ステートメントの `finally` ブロックに移ります。制御が `finally` ブロックの終了点に達すると、その外側の `try` ステートメントの `finally` ブロックに制御が移されます。外側のすべての `try` ステートメントの `finally` ブロックが実行されるまで、このプロセスが繰り返されます。
- 制御が反復子ブロックの呼び出し元に戻されます。呼び出し元は、列挙子オブジェクトの `MoveNext` メソッドまたは `Dispose` メソッドです。

`yield break` ステートメントは無条件に制御を他の場所に移すため、`yield break` ステートメントの終了点は常に到達不可能です。

## 9. 名前空間

C# プログラムは、名前空間を使って整理されます。名前空間は、プログラムの "内部" 構成システムとして、および他のプログラムに公開するプログラム要素を表す手段、つまり "外部" 構成システムとして使用されます。

名前空間を簡単に使用するために、`using` ディレクティブ (9.4 を参照) が提供されています。

### 9.1 コンパイル単位

*compilation-unit* は、ソース ファイルの構造全体を定義します。1 つのコンパイル単位は、順に、0 以上の *using-directive*、0 以上の *global-attribute*、0 以上の *namespace-member-declaration* で構成されます。

```
compilation-unit:  
    extern-alias-directivesopt using-directivesopt global-attributesopt  
    namespace-member-declarationsopt
```

C# プログラムは、1 つ以上のコンパイル単位で構成され、各コンパイル単位は、別個のソース ファイルに含まれています。C# プログラムがコンパイルされる場合は、コンパイル単位のすべてが同時に処理されます。したがって、コンパイル単位は相互に依存できます。循環状の依存も可能です。

コンパイル単位の *using-directive* は、そのコンパイル単位の *global-attribute* および *namespace-member-declaration* に影響を与えますが、他のコンパイル単位への影響はありません。

コンパイル単位の *global-attributes* (17 を参照) を使用して、ターゲットのアセンブリおよびモジュールの属性を指定できます。アセンブリとモジュールは、型の物理的なコンテナーとして機能します。アセンブリは、複数の独立したモジュールで構成されることもあります。

プログラムの各コンパイル単位の *namespace-member-declaration* は、グローバル名前空間と呼ばれる单一の宣言空間にメンバーを追加します。次に例を示します。

```
ファイル A.cs  
class A {}  
  
ファイル B.cs  
class B {}
```

2 つのコンパイル単位が单一のグローバル名前空間に追加されます。この場合、2 つのクラスを A と B という完全修飾名で宣言しています。2 つのコンパイル単位が同じ宣言空間に追加されるため、それぞれに同じ名前のメンバー宣言が含まれる場合にはエラーになります。

### 9.2 名前空間の宣言

*namespace-declaration* は、キーワード `namespace`、名前空間の名前と本体、および省略可能なセミコロンで構成されます。

```
namespace-declaration:  
    namespace qualified-identifier namespace-body ;opt
```

```

qualified-identifier:
  identifier
  qualified-identifier . identifier

namespace-body:
  { extern-alias-directivesopt using-directivesopt namespace-member-declarationsopt }

```

*namespace-declaration* は、*compilation-unit* の最上位の宣言として、または別の *namespace-declaration* 内のメンバー宣言として行われます。*namespace-declaration* が *compilation-unit* の最上位の宣言として行われる場合、その名前空間はグローバル名前空間のメンバーになります。*namespace-declaration* が別の *namespace-declaration* 内で行われる場合、内側の名前空間は、外側の名前空間のメンバーになります。いずれの場合でも、名前空間の名前は、記述されている名前空間内で一意である必要があります。

名前空間は暗黙的に **public** であり、名前空間の宣言にはアクセス修飾子を含めることはできません。

*namespace-body* 内では、オプションの *using-directive* で他の名前空間および型の名前をインポートします。これにより、修飾名を使わなくても直接参照できます。オプションの *namespace-member-declaration* は、名前空間の宣言空間にメンバーを追加します。すべての *using-directive* はメンバー宣言の前に配置する必要があります。

*namespace-declaration* の *qualified-identifier* は、単一の識別子か、または ".." トークンで区切られた複数の識別子になります。後者の形式を使うと、複数の名前空間宣言を文法的に入れ子にしなくとも、入れ子になった名前空間をプログラムで定義できます。次に例を示します。

```

namespace N1.N2
{
    class A {}
    class B {}
}

```

これは、次と同じ意味になります。

```

namespace N1
{
    namespace N2
    {
        class A {}
        class B {}
    }
}

```

名前空間はオープンエンドであり、同じ完全限定名を持つ 2 つの名前空間宣言は同じ宣言空間に追加されます(3.3 を参照)。次に例を示します。

```

namespace N1.N2
{
    class A {}
}

namespace N1.N2
{
    class B {}
}

```

上の例では、2 つの名前空間宣言は同じ宣言空間に含まれ、完全修飾名が **N1.N2.A** と **N1.N2.B** である 2 つのクラスが宣言されています。2 つの宣言が同じ宣言空間に追加されるため、それぞれに同じ名前のメンバー宣言が含まれる場合はエラーになります。

## 9.3 extern エイリアス

*extern-alias-directive*により、名前空間のエイリアスとして機能する識別子が導入されます。エイリアスの対象の名前空間の指定はプログラムのソースコード外にあり、エイリアスの対象の名前空間の入れ子になった名前空間にも適用されます。

```
extern-alias-directives:
    extern-alias-directive
    extern-alias-directives extern-alias-directive

extern-alias-directive:
    extern alias identifier ;
```

*extern-alias-directive* のスコープは、すぐ外側のコンパイル単位または名前空間本体の *using-directives*、*global-attributes*、および *namespace-member-declarations* までです。

*extern-alias-directive* を含むコンパイル単位または名前空間本体内では、*extern-alias-directive* で導入された識別子を使って、エイリアスを使用した名前空間を参照できます。*identifier* として **global** という語を指定するとコンパイルエラーになります。

*extern-alias-directive* は、特定のコンパイル単位または名前空間本体内でエイリアスを使用できるようになりますが、基になる宣言空間に新規メンバーを追加するわけではありません。つまり、*extern-alias-directive* は波及性がなく、*extern-alias-directive* が記述されているコンパイル単位または名前空間の本体の中だけで有効です。

次のプログラムは、2つの **extern** エイリアス **X** と **Y** を宣言および使用します。それぞれの **extern** エイリアスは、固有の名前空間階層のルートを示します。

```
extern alias X;
extern alias Y;

class Test
{
    X::N.A a;
    X::N.B b1;
    Y::N.B b2;
    Y::N.C c;
}
```

このプログラムは **extern** エイリアス **X** および **Y** の存在を宣言していますが、エイリアスの実際の定義はプログラムの外にあります。これで、同じ名前の **N.B** クラスを **X.N.B** および **Y.N.B** として、または名前空間エイリアス修飾子を使用して **X::N.B** および **Y::N.B** として参照できます。外部定義のない **extern** エイリアスを定義すると、エラーが発生します。

## 9.4 Using ディレクティブ

**using** ディレクティブを使うと、他の名前空間に定義された名前空間と型を簡単に使うことができます。**using** ディレクティブは、*namespace-or-type-name* (3.8 を参照) および *simple-name* (7.6.2 を参照) の名前解決プロセスに影響します。ただし、宣言とは異なり、**using** ディレクティブでは、コンパイル単位の基になる宣言空間またはディレクティブが使用される名前空間に新しいメンバーが追加されることはありません。

```
using-directives:
    using-directive
    using-directives using-directive
```

```
using-directive:
  using-alias-directive
  using-namespace-directive
```

*using-alias-directive* (9.4.1 を参照) は、名前空間または型のエイリアスを導入します。

*using-namespace-directive* (9.4.2 を参照) は、名前空間の型メンバーをインポートします。

*using-directive* のスコープは、記述されているコンパイル単位または名前空間本体の *namespace-member-declarations* まで拡張されます。*using-directive* のスコープは、同等の *using-directive* を含みません。そのため、同等の *using-directive* は相互に影響せず、それらのディレクティブを記述する順序は重要ではありません。

#### 9.4.1 Using alias ディレクティブ

*using-alias-directive* は、すぐ外側のコンパイル単位または名前空間本体内で、名前空間または型のエイリアスとして機能する識別子を導入します。

```
using-alias-directive:
  using identifier = namespace-or-type-name ;
```

*using-alias-directive* を含むコンパイル単位または名前空間本体内のメンバー宣言では、*using-alias-directive* で導入された識別子を使って、特定の名前空間または型を参照できます。次に例を示します。

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using A = N1.N2.A;
    class B: A {}
}
```

この例の N3 名前空間のメンバー宣言内では、A が N1.N2.A のエイリアスです。したがって、N3.B クラスは N1.N2.A クラスから派生しています。N1.N2 のエイリアス R を作成し、R.A を参照することによっても、同じ効果が得られます。

```
namespace N3
{
    using R = N1.N2;
    class B: R.A {}
}
```

*using-alias-directive* の *identifier* は、*using-alias-directive* を直接含むコンパイル単位または名前空間の宣言空間内で、一意である必要があります。次に例を示します。

```
namespace N3
{
    class A {}
}

namespace N3
{
    using A = N1.N2.A;      // Error, A already exists
```

上の N3 には既に A というメンバーが含まれているため、その識別子を *using-alias-directive* で使用するとコンパイルエラーが発生します。同様に、同じコンパイル単位または名前空間本体で、2つ以上の *using-alias-directive* がエイリアスを同名で宣言するとコンパイルエラーが発生します。

*using-alias-directive* は、特定のコンパイル単位または名前空間本体内でエイリアスを使用できるようになりますが、基になる宣言空間に新規メンバーを追加するわけではありません。つまり、*using-alias-directive* は波及性がなく、*using-alias-directive* が記述されているコンパイル単位または名前空間の本体の中だけで有効です。次に例を示します。

```
namespace N3
{
    using R = N1.N2;
}

namespace N3
{
    class B: R.A {}           // Error, R unknown
}
```

R を導入する *using-alias-directive* のスコープは、それを含む名前空間の本体内のメンバー宣言までしか拡張されないため、2番目の名前空間宣言では R は認識されません。ただし、コンパイル単位に *using-alias-directive* を配置すると、エイリアスは両方の名前空間宣言内で利用できるようになります。

```
using R = N1.N2;

namespace N3
{
    class B: R.A {}
}

namespace N3
{
    class C: R.A {}
}
```

*using-alias-directive* で導入された名前は、通常のメンバーと同様に、入れ子になったスコープ内の同様の名前のメンバーによって隠れられます。次に例を示します。

```
using R = N1.N2;

namespace N3
{
    class R {}
    class B: R.A {}        // Error, R has no member A
}
```

B の宣言での R.A への参照はコンパイルエラーになります。R が参照するのは N3.R であり、N1.N2 ではないからです。

*using-alias-directive* の記述順序に意味はありません。また、*using-alias-directive* によって参照される *namespace-or-type-name* の解決は、*using-alias-directive* 自体にも、すぐ外側のコンパイル単位または名前空間本体に含まれる他の *using-directive* にも影響されません。つまり、また、*using-alias-directive* の *namespace-or-type-name* は、すぐ外側のコンパイル単位または名前空間本体に *using-directive* がない場合と同様に解決されます。ただし、*using-alias-directive* は、すぐ外側のコンパイル単位または名前空間本体に含まれる *extern-alias-directives* によって影響される可能性があります。次に例を示します。

```
namespace N1.N2 {}

namespace N3
{
    extern alias E;
```

```

using R1 = E.N;          // OK
using R2 = N1;           // OK
using R3 = N1.N2;         // OK
using R4 = R2.N2;         // Error, R2 unknown
}

```

最後の *using-alias-directive* は、最初の *using-alias-directive* の影響を受けないので、コンパイルエラーになります。最初の *using-alias-directive* は、*extern* エイリアス *E* のスコープが *using-alias-directive* を含むため、エラーになりません。

*using-alias-directive* は、その外側の名前空間や、その名前空間の入れ子になった名前空間や型など、任意の名前空間または型のエイリアスを作成できます。

エイリアスを使って名前空間や型にアクセスするのは、宣言名を使って名前空間や型にアクセスするのと、まったく同じ結果になります。例を次に示します。

```

namespace N1.N2
{
    class A {}
}

namespace N3
{
    using R1 = N1;
    using R2 = N1.N2;

    class B
    {
        N1.N2.A a;           // refers to N1.N2.A
        R1.N2.A b;           // refers to N1.N2.A
        R2.A c;              // refers to N1.N2.A
    }
}

```

*N1.N2.A*、*R1.N2.A*、および *R2.A* の名前は同じで、いずれも *N1.N2.A* という完全修飾名のクラスを参照しています。

*using* エイリアスではクローズ構築型の名前を付けることができますが、型引数を指定せずに非バインドジェネリック型宣言の名前を付けることはできません。次に例を示します。

```

namespace N1
{
    class A<T>
    {
        class B {}
    }
}

namespace N2
{
    using W = N1.A;           // Error, cannot name unbound generic type
    using X = N1.A.B;         // Error, cannot name unbound generic type
    using Y = N1.A<int>;     // Ok, can name closed constructed type
    using Z<T> = N1.A<T>;   // Error, using alias cannot have type parameters
}

```

#### 9.4.2 Using namespace ディレクティブ

*using-namespace-directive* は、名前空間に含まれる型をすぐ外側のコンパイル単位または名前空間の本体にインポートします。これにより、各型の識別子を修飾なしで使用できます。

```
using-namespace-directive:
  using namespace-name ;
```

*using-namespace-directive* を含むコンパイル単位または名前空間本体のメンバー宣言内では、指定の名前空間に含まれる型を直接参照できます。次に例を示します。

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using N1.N2;
    class B: A {}
}
```

この場合、N3 名前空間のメンバー宣言内では N1.N2 の型メンバーを直接利用でき、クラス N3.B はクラス N1.N2.A から派生します。

*using-namespace-directive* は、指定の名前空間に含まれる型をインポートしますが、入れ子になった名前空間はインポートされません。次に例を示します。

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using N1;
    class B: N2.A {}      // Error, N2 unknown
}
```

*using-namespace-directive* は、N1 に含まれる型をインポートしますが、N1 内で入れ子になった名前空間はインポートされません。したがって、N2 という名前のメンバーがスコープ内に存在しないため、B の宣言内での N2.A への参照はコンパイルエラーになります。

*using-alias-directive* とは異なり、*using-namespace-directive* は、外側のコンパイル単位または名前空間本体の中で識別子が定義されている型をインポートします。実際、*using-namespace-directive* によってインポートされた名前は、外側のコンパイル単位または名前空間本体内の同様の名前のメンバーによって隠ぺいされます。次に例を示します。

```
namespace N1.N2
{
    class A {}
    class B {}
}

namespace N3
{
    using N1.N2;
    class A {}
}
```

この場合、N3 名前空間のメンバー宣言内で、A は N1.N2.A ではなく N3.A を参照します。

同じコンパイル単位または名前空間本体内の *using-namespace-directive* によってインポートされた複数の名前空間に、同名の型が含まれる場合、その名前に対する参照はあいまいになると見なされます。次に例を示します。

```
namespace N1
{
    class A {}
}

namespace N2
{
    class A {}
}

namespace N3
{
    using N1;
    using N2;
    class B: A {}           // Error, A is ambiguous
}
```

N1 と N2 の両方にメンバー A が含まれますが、N3 は両方をインポートするため、N3 内の A を参照するとコンパイルエラーになります。この状況では、A への参照の修飾を使用して、または特定の A を選択する *using-alias-directive* を導入することで、矛盾を解決できます。次に例を示します。

```
namespace N3
{
    using N1;
    using N2;
    using A = N1.A;
    class B: A {}           // A means N1.A
}
```

*using-alias-directive* と同様に、*using-namespace-directive* は、コンパイル単位または名前空間の基になる宣言空間に新規メンバーを追加せず、*using-namespace-directive* が記述されているコンパイル単位または名前空間本体の中だけで有効です。

*using-namespace-directive* によって参照される *namespace-name* は、*using-alias-directive* によって参照される *namespace-or-type-name* と同じ方法で解決されます。したがって、同じコンパイル単位または名前空間本体に含まれる *using-namespace-directive* は、相互に影響せず、任意の順序で記述できます。

## 9.5 名前空間のメンバー

*namespace-member-declaration* は、*namespace-declaration* (9.2 を参照) か *type-declaration* (9.6 を参照) のいずれかです。

```
namespace-member-declarations:
    namespace-member-declaration
    namespace-member-declarations  namespace-member-declaration

namespace-member-declaration:
    namespace-declaration
    type-declaration
```

コンパイル単位または名前空間本体は、*namespace-member-declaration* を含むことができ、その名前空間メンバーの宣言は、*namespace-member-declaration* が記述されているコンパイル単位または名前空間本体の基になる宣言空間に、新規メンバーを追加します。

## 9.6 型の宣言

*type-declaration* は、*class-declaration* (10.1 を参照)、*struct-declaration* (11.1 を参照)、*interface-declaration* (13.1 を参照)、*enum-declaration* (14.1 を参照)、*delegate-declaration* (15.1 を参照) のいずれかです。

```
type-declaration:
    class-declaration
    struct-declaration
    interface-declaration
    enum-declaration
    delegate-declaration
```

*type-declaration* は、コンパイル単位の最上位の宣言として、または名前空間、クラス、構造体内のメンバー宣言として行うことができます。

型 *T* の型宣言がコンパイル単位の最上位の宣言として行われる場合、新しく宣言される型の完全修飾名は単に *T* となります。型 *T* の型宣言が名前空間、クラス、または構造体内で行われる場合、新しく宣言される型の完全修飾名は *N.T* となります。ここで、*N* は型宣言が記述される名前空間、クラス、または構造体の完全修飾名です。

クラスや構造体内で宣言された型は、入れ子になった型 (10.3.8 を参照) と呼ばれます。

型宣言に使用できるアクセス修飾子と既定のアクセスは、次のように、宣言が行われるコンテキストに依存します (3.5.1 を参照)。

- コンパイル単位または名前空間で宣言された型は、**public** アクセスまたは **internal** アクセスを行うことができます。既定は **internal** アクセスです。
- クラス内で宣言された型は、**public**、**protected** **internal**、**protected**、**internal**、**private** の各アクセスを行うことができます。既定は **private** アクセスです。
- 構造体内で宣言された型は、**public**、**internal**、**private** の各アクセスを行うことができます。既定は **private** アクセスです。

## 9.7 名前空間エイリアス修飾子

**名前空間エイリアス修飾子** :: を使用すると、新しい型やメンバーが導入されても名前検索が影響を受けないことが保証されます。名前空間エイリアス修飾子は、常に、左側の識別子と右側の識別子という 2 つの識別子の間で使用します。通常の . 修飾子とは異なり、:: 修飾子の左側の識別子は、単に **extern** エイリアスまたは **using** エイリアスとしてのみ検索されます。

*qualified-alias-member* は次のように定義されます。

```
qualified-alias-member:
    identifier :: identifier type-argument-listopt
```

*qualified-alias-member* は、*namespace-or-type-name* (3.8 を参照) として、または *member-access* (7.6.4 を参照) の左オペランドとして使用できます。

*qualified-alias-member* には次の 2 つの形式があります。

- N::I<A<sub>1</sub>, ..., A<sub>K</sub>>*。ここで、*N* および *I* は識別子を示し、<*A<sub>1</sub>*, ..., *A<sub>K</sub>*> は型引数リストです (*K* は常に少なくとも 1)。
- N::I,*。ここで、*N* および *I* は識別子を示します。(この場合、*K* はゼロと見なされます)。

この形式を使用して、*qualified-alias-member* の意味は次のように決定されます。

- *N* が識別子 `global` の場合は、次のように、*I* に対してグローバルな名前空間を検索します。
  - グローバル名前空間が *I* という名前空間を含み、*K* がゼロの場合、*qualified-alias-member* はその名前空間を参照します。
  - それ以外の場合で、グローバル名前空間が *I* という非ジェネリック型を含み、*K* がゼロの場合、*qualified-alias-member* はその型を参照します。
  - それ以外の場合で、グローバル名前空間が *I* という型を含み、*I* が *K* 個の型パラメーターを持つ場合、*qualified-alias-member* は指定された型引数で構築されたその型を参照します。
  - 以上のいずれにも該当しない場合、*qualified-alias-member* は未定義で、コンパイルエラーになります。
- それ以外の場合は、*qualified-alias-member* (存在する場合) のすぐ外側の名前空間宣言 (9.2 を参照) から開始して *qualified-alias-member* を含むコンパイル単位まで、順番に外側の各名前空間宣言 (存在する場合) に進みながらエンティティが見つかるまで次の手順が評価されます。
  - 名前空間宣言またはコンパイル単位に、*N* を型に関連付ける *using-alias-directive* が含まれる場合、*qualified-alias-member* は未定義でコンパイルエラーになります。
  - それ以外の場合で、名前空間宣言またはコンパイル単位に *N* を名前空間に関連付ける *extern-alias-directive* または *using-alias-directive* が含まれる場合は、次のように処理されます。
    - *N* に関連付けられた名前空間が *I* という名前の名前空間を含み、*K* がゼロの場合、*qualified-alias-member* はその名前空間を参照します。
    - それ以外の場合で、*N* に関連付けられた名前空間が *I* という非ジェネリック型を含み、*K* がゼロの場合、*qualified-alias-member* はその型を参照します。
    - それ以外の場合で、*N* に関連付けられた名前空間が *I* という型を含み、*I* が *K* 個の型パラメーターを持つ場合、*qualified-alias-member* は指定された型引数で構築されたその型を参照します。
    - 以上のいずれにも該当しない場合、*qualified-alias-member* は未定義で、コンパイルエラーになります。
- 以上のいずれにも該当しない場合、*qualified-alias-member* は未定義で、コンパイルエラーになります。

型を参照するエイリアスと一緒に名前空間エイリアス修飾子を使用すると、コンパイルエラーになります。また、識別子 *N* が `global` の場合は、`global` を型または名前空間に関連付ける *using* エイリアスがあったとしても、グローバル名前空間で検索が実行されます。

### 9.7.1 エイリアスの一意性

各コンパイル単位および名前空間の本体には、*extern* エイリアスおよび *using* エイリアスに対する個別の宣言空間があります。したがって、*extern* エイリアスまたは *using* エイリアスの名前は、すぐ外側のコンパイル単位または名前空間の本体に宣言されている *extern* エイリアスおよび *using* エイリアスの集合内では一意である必要がありますが、エイリアスが常に `::` 修飾子付きで使用される場合、そのエイリアスは型または名前空間と同じ名前を持つことができます。

次に例を示します。

```
namespace N
{
    public class A {}
    public class B {}
}

namespace N
{
    using A = System.IO;
    class X
    {
        A.Stream s1;           // Error, A is ambiguous
        A::Stream s2;           // Ok
    }
}
```

ここでは、クラス A と using エイリアス A の両方がスコープにあるため、2 番目の名前空間の A という名前は 2 つの意味を持ちます。このため、修飾名 A.Stream での A の使用はあいまいであります。コンパイルエラーが発生します。ただし、A を修飾子 :: と共に使用すると、A は名前空間エイリアスとしてのみ検索されるため、エラーになりません。



# 10. クラス

クラスとは、データ メンバー (定数およびフィールド)、関数メンバー (メソッド、プロパティ、イベント、インデクサー、演算子、インスタンス コンストラクター、デストラクター、および静的コンストラクター)、および入れ子になった型を格納するデータ構造です。クラス型は、継承をサポートします。継承とは、派生クラスが基底クラスを拡張および特化するための機構です。

## 10.1 クラス宣言

*class-declaration* は、新しいクラスを宣言する *type-declaration* (9.6 を参照) です。

```
class-declaration:  
    attributesopt class-modifiersopt partialopt class identifier type-parameter-listopt  
        class-baseopt type-parameter-constraints-clausesopt class-body ;opt
```

*class-declaration* は省略可能な属性のセット (17 を参照)、省略可能な *class-modifiers* のセット (10.1.1 を参照)、省略可能な *partial* 修飾子、キーワード *class* と、クラスを指定する識別子、省略可能な *type-parameter-list* (10.1.3 を参照)、省略可能な *class-base* 指定 (10.1.4 を参照)、省略可能な *type-parameter-constraints-clauses* のセット (10.1.5 を参照)、*class-body* (10.1.6 を参照)、および省略可能なセミコロンから構成されます。

クラス宣言で *type-parameter-constraints-clauses* を指定する場合は、*type-parameter-list* も指定する必要があります。

*type-parameter-list* を指定するクラス宣言は "ジェネリック クラス宣言" です。また、ジェネリック クラス宣言またはジェネリック 構造体宣言の入れ子になったクラスは、クラスそれ自体がジェネリック クラス宣言です。これは、構築された型を作成するには格納する型の型パラメーターを指定する必要があるためです。

### 10.1.1 クラス修飾子

*class-declaration* には、オプションとして一連のクラス修飾子を含めることができます。

```
class-modifiers:  
    class-modifier  
    class-modifiers class-modifier  
  
class-modifier:  
    new  
    public  
    protected  
    internal  
    private  
    abstract  
    sealed  
    static
```

1 つのクラス宣言内で同じ修飾子を複数回使用すると、コンパイル エラーになります。

入れ子になったクラスでは、`new`修飾子を使用できます。この修飾子は、クラスが継承されたメンバーと同じ名前で隠ぺいするように指定します。詳細については、10.3.4を参照してください。入れ子になっていないクラス宣言で`new`修飾子を使用すると、コンパイル時のエラーになります。

`public`、`protected`、`internal`、および`private`の各修飾子は、クラスのアクセシビリティを制御します。クラスの宣言が行われるコンテキストによっては、これらの修飾子の一部は使用できません(3.5.1を参照)。

`abstract`、`sealed`、`static`の各修飾子について、次のセクションで説明します。

#### 10.1.1.1 抽象クラス

`abstract`修飾子は、クラスが完全ではなく、基底クラスとしてのみ使用するように想定されていることを示します。抽象クラスと非抽象クラスの違いは次のとおりです。

- 抽象クラスを直接インスタンス化することはできません。抽象クラスに`new`演算子を使用すると、コンパイルエラーになります。コンパイル時の型が抽象である変数と値を作成することはできますが、そのような変数と値は必然的に`null`になるか、または抽象型から派生した非抽象型のインスタンスへの参照を持つことになります。
- 抽象クラスは、抽象クラスを格納できますが必須ではありません。
- 抽象クラスはシールできません。

非抽象クラスが抽象クラスから派生される場合、その非抽象クラスは、継承された抽象メンバーの実際の実装をすべて含んでいる必要があります。このような実装は、継承された抽象メンバーをオーバーライドして行います。次に例を示します。

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

abstract class B: A
{
    public void G() {}
}

class C: B
{
    public override void F() {
        // actual implementation of F
    }
}
```

抽象クラス`A`は、抽象メソッド`F`を導入します。クラス`B`は追加のメソッド`G`を導入しますが、`F`を実装しないので、`B`も抽象として宣言する必要があります。`C`クラスは`F`をオーバーライドし、実際の実装を提供します。`C`には抽象メンバーがないので、`C`は非抽象にできますが、必須ではありません。

#### 10.1.1.2 シールクラス

`sealed`修飾子は、クラスが派生されないようにするために使用します。シールクラスが別のクラスの基底クラスとして指定されると、コンパイルエラーになります。

シールクラスを抽象クラスにすることもできません。

**sealed** 修飾子は、主に意図しない派生を防ぐために使用されますが、実行時の特定の最適化も有効になります。特に、シールクラスは派生クラスを持たないので、シールクラスのインスタンスに対する仮想関数メンバーの呼び出しを非仮想呼び出しに変換できます。

### 10.1.1.3 静的クラス

**static** 修飾子は、宣言されるクラスを "静的クラス" としてマークするために使用します。静的クラスはインスタンス化できず、型として使用することもできません。静的クラスには、静的メンバーのみを含めることができます。静的クラスのみが、拡張メソッドの宣言を含むことができます(10.6.9を参照)。

静的クラスの宣言には、次の制限事項があります。

- 静的クラスには、**sealed** 修飾子も **abstract** 修飾子も指定できません。しかし、静的クラスはインスタンス化することも派生元として使用することもできないため、シールクラスかつ抽象クラスであるかのように動作します。
- 静的クラスに *class-base* 指定(10.1.4を参照)を含めることはできません。基底クラスや実装されるインターフェイスのリストを明示的に指定することはできません。静的クラスは、暗黙的に **object** 型から継承します。
- 静的クラスは、静的メンバー(10.3.7を参照)のみを含むことができます。定数および入れ子になった型は、静的メンバーに分類されます。
- 静的クラスは、アクセシビリティが **protected** または **protected internal** として宣言されるメンバーを持つことはできません。

以上の制限に違反すると、コンパイルエラーになります。

静的クラスは、インスタンス コンストラクターを持ちません。静的クラスにインスタンス コンストラクターを宣言することはできません。また、静的クラスには既定のインスタンス コンストラクター(10.11.4を参照)はありません。

静的クラスのメンバーは、自動的に静的にはなりません。定数および入れ子になった型以外は、メンバー宣言に明示的に **static** 修飾子を含む必要があります。クラスが静的外部クラス内に入れ子になっている場合、入れ子の内部のクラスは、明示的に **static** 修飾子を含まない限り、静的クラスではありません。

#### 10.1.1.3.1 静的クラス型の参照

*namespace-or-type-name*(3.8を参照)は、次の場合に静的クラスを参照できます。

- *namespace-or-type-name* が形式 **T.I** の *namespace-or-type-name* 内の **T** である場合
- または、*namespace-or-type-name* が形式 **typeof(T)** の *typeof-expression*(7.5.11を参照)内の **T** である場合

*primary-expression*(7.5を参照)は、次の場合に静的クラスを参照できます。

- *primary-expression* は、形式 **E.I** の *member-access*(7.5.4を参照)の **E** です。

他の場合は、静的クラスを参照するとコンパイルエラーになります。たとえば、静的クラスを基底クラス、メンバーの構成要素型(10.3.8を参照)、ジェネリック型引数、または型パラメーターの制約に使用するとエラーになります。同様に、配列型、ポインター型、**new** 式、キャスト式、**is** 式、**as** 式、**sizeof** 式、または既定値の式で静的クラスを使用することはできません。

### 10.1.2 Partial 修飾子

`partial` 修飾子は、この *class-declaration* が部分型宣言であることを示します。外側にある名前空間または型宣言にある同じ名前の複数の部分型宣言は、10.2 で説明されている規則に従って、合わせて 1 つの型宣言が形成されます。

プログラム テキストの個別のセグメントにクラス宣言を分散すると、これらのセグメントが異なるコンテキストに生成されたり、管理される場合に有益になります。たとえば、クラス宣言の一部を機械的に生成し、残りの部分を手動で作成できます。2 つをテキスト分離することで、他方の更新と衝突する更新が行われるのを防止できます。

### 10.1.3 型パラメーター

型パラメーターは単なる識別子であり、構築された型を作成するために指定する型引数のプレースホルダーを表します。型パラメーターは、後で定義される型の形式的なプレースホルダーです。これに対して、型引数 (4.4.1 を参照) は実際の型であり、構築された型を作成するときの型パラメーターとして使用されます。

```

type-parameter-list:
    < type-parameters >

type-parameters:
    attributesopt type-parameter
    type-parameters , attributesopt type-parameter

type-parameter:
    identifier

```

クラス宣言の各型パラメーターは、そのクラスの宣言空間 (3.3 を参照) 内に名前を定義します。したがって、そのクラスに宣言されている他の型パラメーターまたはメンバーと同じ名前を付けることはできません。型パラメーターに、型自体と同じ名前を付けることはできません。

### 10.1.4 Class base 指定

クラスの宣言には、*class-base* 指定を含めることができます。この指定では、クラスの直接基底クラス、およびクラスによって直接実装されるインターフェイス (13 を参照) を定義します。

```

class-base:
    : class-type
    : interface-type-list
    : class-type , interface-type-list

interface-type-list:
    interface-type
    interface-type-list , interface-type

```

クラス宣言に指定する基底クラスとして、構築されたクラス型 (4.4 を参照) を使用できます。基底クラスはそれだけでは型パラメーターにはなりませんが、スコープ内で型パラメーターと一緒に使用できます。

```
class Extend<V>: V {}           // Error, type parameter used as base class
```

#### 10.1.4.1 基底クラス

*class-type* が *class-base* に含まれている場合、その *class-type* は、宣言されるクラスの直接基底クラスを指定します。クラス宣言に *class-base* がない場合、または *class-base* にインターフェイス型だけが

指定されている場合は、直接基底クラスが `object` であると仮定されます。クラスは直接基底クラスからメンバーを継承します。詳細については、10.3.3 を参照してください。

次に例を示します。

```
class A {}
class B: A {}
```

クラス `A` は `B` の直接基底クラスと呼ばれ、`B` は `A` からの派生と呼ばれます。`A` は直接基底クラスを明示的に指定していないため、その直接基底クラスは暗黙的に `object` になります。

構築されたクラス型では、ジェネリック クラス宣言に基底クラスが指定されている場合、基底クラス宣言の各 `type-parameter` を構築された型の対応する `type-argument` で置き換えることで、構築された型の基底クラスが取得されます。次のジェネリック クラス宣言があるとします。

```
class B<U,V> {...}
class G<T>: B<string,T[]> {...}
```

構築された型 `G<int>` の基底クラスは `B<string,int[]>` になります。

クラスの型の直接基底クラスは、少なくとも、クラスの型自体と同程度にアクセス可能である必要があります (3.5.2 を参照)。たとえば、`public` クラスが `private` クラスや `internal` クラスから派生するとコンパイルエラーになります。

クラス型の直接基底クラスは、`System.Array`、`System.Delegate`、`System.MulticastDelegate`、`System.Enum`、または `System.ValueType` 以外の型にする必要があります。さらに、ジェネリック クラス宣言では、直接基底クラスまたは間接基底クラスとして `System.Attribute` を使用することはできません。

クラス `B` の直接基底クラス指定 `A` の意味が判断されるとき、直接基底クラス `B` は一時的に `object` として仮定されます。これによって、基底クラスの指定の意味が再帰的にそれ自身に依存することがないことが直感的にわかるようになります。次に例を示します。

```
class A<T> {
    public class B{}
}
class C : A<C.B> {}
```

基底クラスの指定 `A<C.B>` では直接基底クラス `C` が `object` であると見なされるため、これはエラーになります。そのため、(3.8 の規則により) `C` がメンバー `B` を含むとは見なされません。

クラス型の基底クラスは、直接基底クラスとその基底クラスです。つまり、基底クラスの集合は、直接基底クラスに関する推移的閉包です。上の例を参照すると、`B` の基底クラスは `A` と `object` です。次に例を示します。

```
class A {...}
class B<T>: A {...}
class C<T>: B<IComparable<T>> {...}
class D<T>: C<T[]> {...}
```

`D<int>` の基底クラスは、`C<int[]>`、`B<IComparable<int[]>>`、`A`、および `object` です。

`object` クラスを除くと、どのクラス型も直接基底クラスを 1 つだけ持ちます。`object` クラスは直接基底クラスを持たず、他のすべてのクラスの最終的な基底クラスとなります。

クラス **B** がクラス **A** から派生する場合、**A** が **B** に依存するとコンパイルエラーになります。クラスは直接基底クラス(存在する場合)および、入れ子のすぐ外側にあるクラス(存在する場合)に "**直接依存**" します。この定義により、クラスが依存するクラスの完全な集合は、"**直接依存**" 関係の再帰的および推移的な閉包となります。

次の例を参照してください。

```
class A: A {}
```

これは、クラスがそれ自身に依存しているため、エラーとなります。同様に、次の例があります。

```
class A: B {}
class B: C {}
class C: A {}
```

これは、クラスが循環的に依存しているのでエラーになります。次に例を示します。

```
class A: B.C {}
class B: A
{
    public class C {}
}
```

これは、**A** が **B.C**(直接基底クラス)に依存し、**B.C** は **B**(すぐ外側のクラス)に依存し、**B** は **A** に依存して最終的に循環依存になるため、コンパイルエラーが発生します。

クラスは、その内側で入れ子になったクラスに依存しません。次に例を示します。

```
class A
{
    class B: A {}
}
```

**A** は直接基底クラスであると同時にすぐ外側のクラスであるため、**B** は **A** に依存しますが、**B** は **A** の基底クラスでも外側のクラスでもないため、**A** は **B** に依存しません。したがって、この例は有効です。

**sealed** クラスからはクラスを派生できません。次に例を示します。

```
sealed class A {}
class B: A {}           // Error, cannot derive from a sealed class
```

クラス **B** は、**sealed** クラス **A** からの派生が原因でエラーになります。

#### 10.1.4.2 インターフェイスの実装

*class-base* 指定には、インターフェイス型のリストが含まれることがあります。この場合、クラスは指定のインターフェイス型を直接実装していると言います。インターフェイスの実装の詳細については、13.4 を参照してください。

#### 10.1.5 型パラメーターの制約

ジェネリック型およびメソッドの宣言では、*type-parameter-constraints-clause* を含めることによって、オプションで型パラメーターの制約を指定できます。

```
type-parameter-constraints-clauses:
    type-parameter-constraints-clause
    type-parameter-constraints-clauses  type-parameter-constraints-clause
```

```

type-parameter-constraints-clause:
  where type-parameter : type-parameter-constraints

type-parameter-constraints:
  primary-constraint
  secondary-constraints
  constructor-constraint
  primary-constraint , secondary-constraints
  primary-constraint , constructor-constraint
  secondary-constraints , constructor-constraint
  primary-constraint , secondary-constraints , constructor-constraint

primary-constraint:
  class-type
  class
  struct

secondary-constraints:
  interface-type
  type-parameter
  secondary-constraints , interface-type
  secondary-constraints , type-parameter

constructor-constraint:
  new ( )

```

各 *type-parameter-constraints-clause* は、トークン **where**、型パラメーターの名前、コロン、および型パラメーターの制約リストから構成されます。各型パラメーターの **where** 句は 1 つだけで、**where** 句は任意の順序で並べることができます。プロパティアクセサーの **get** トークンおよび **set** トークンと同様に、**where** トークンはキーワードではありません。

**where** 句に指定する制約のリストには、1 つの主制約、1 つ以上の 2 次制約 **\b**、およびコンストラクター制約 **\b \b** である **new()** をこの順序で含めることができます。

主制約には、クラス型、"参照型制約" **class**、または "値型の制約" **struct** を指定できます。2 次制約は、*type-parameter* または *interface-type* です。

参照型制約の指定により、型パラメーターに使用する型引数は参照型である必要があります。すべてのクラス型、インターフェイス型、デリゲート型、配列型、および(下記の定義に従って)参照型とわかっている型パラメーターは、この制約を満たします。

値型の制約は、型パラメーターに使用する型引数が **null** 非許容の値型である必要があると指定しています。すべての **null** 非許容の構造体型、列挙型、および値型の制約を持つ型パラメーターは、この制約を満たします。**null** 許容型 (4.1.10 を参照) は値型に分類されていますが、値型の制約を満たしません。値型の制約を持つ型パラメーターは、同時に *constructor-constraint* を持つことはできません。

ポインター型は型引数として使用できません。また、参照型制約や値型の制約を満たすとは見なされません。

制約がクラス型、インターフェイス型、または型パラメーターの場合、その型は型パラメーターに使用されるすべての型引数がサポートする必要がある最低限の "基本型" を表します。構築された型またはジェネリック メソッドが使用されている場合は、型引数が型パラメーターの制約に従っているかどうかがコンパイル時に必ずチェックされます。指定された型引数は、4.4.4 で説明されている条件を満たしている必要があります。

*class-type* の制約は、次の規則を満たす必要があります。

- 型はクラス型であること。
- 型が `sealed` でないこと。
- 型が `System.Array`、`System.Delegate`、`System.Enum`、または `System.ValueType` でないこと。
- 型が `object` でないこと。すべての型は `object` から派生するため、このような制約は許可されても何の効果もありません。
- 特定の型パラメーターに対するクラス型の制約は 1 つまでであること。

*interface-type* 制約として指定された型は、次の規則を満たす必要があります。

- 型はインターフェイス型であること。
- 1 つの型は、1 つの `where` 句の中で複数回指定されないこと。

どちらの場合も、制約は、関連付けられた型の型パラメーターまたはメソッド宣言を構築された型の一部として使用でき、宣言された型を使用できます。

型パラメーター制約として指定するクラス型またはインターフェイス型は、少なくとも、ジェネリック型または宣言されたメソッドと同程度にアクセス可能 (3.5.4 を参照) である必要があります。

*type-parameter* 制約として指定された型は、次の規則を満たす必要があります。

- 型は型パラメーターであること。
- 1 つの型は、1 つの `where` 句の中で複数回指定されないこと。

また、型パラメーターの依存関係が循環していないこと。依存とは次のように定義される推移的関係です。

- 型パラメーター `T` が型パラメーター `S` の制約として使用される場合、`S` は `T` に "依存" します。
- 型パラメーター `S` が型パラメーター `T` に依存し、`T` が型パラメーター `U` に依存する場合、`S` は `U` に依存します。

この関係がある状況で、型パラメーターが (直接または間接に) それ自体に依存すると、コンパイルエラーになります。

依存する型パラメーターの間では、制約は一貫している必要があります。型パラメーター `S` が型パラメーター `T` に依存する場合は、次の条件を満たす必要があります。

- `T` に値型の制約がないこと。値型の制約がある場合、`T` は実質的にシールされて `S` と `T` が同じ型になり、2 つの型パラメーターの必要性が失われます。
- `S` に値型の制約がある場合は、`T` に *class-type* 制約がないこと。
- `S` に *class-type* 制約 `A` があり、`T` に *class-type* 制約 `B` がある場合は、`A` から `B` への恒等変換や暗黙の参照変換、または `B` から `A` への暗黙の参照変換があること。
- `S` が型パラメーター `U` にも依存し、`U` に *class-type* 制約 `A` があり、`T` に *class-type* 制約 `B` がある場合、`A` から `B` への恒等変換または暗黙の参照変換、または `B` から `A` への暗黙の参照変換があること。

`S` が値型の制約を持ち、`T` が参照型制約を持つことは許可されます。この場合、実質的に、`T` は型 `System.Object`、`System.ValueType`、`System.Enum`、および任意のインターフェイス型に限定されます。

型パラメーターの `where` 句にコンストラクター制約 (`new()` という形式) が含まれる場合は、`new` 演算子を使用して型のインスタンスを作成できます (7.6.10.1 を参照)。コンストラクター制約を持つ型パラメーターに対して使用する型引数は、パラメーターのないパブリック コンストラクター (このコンストラクターはすべての値型に対して暗黙的に存在します) を持つか、値型の制約またはコンストラクター制約を持つ型パラメーターである必要があります (詳細については 10.1.5 を参照)。

次に、制約の例を示します。

```
interface IPrintable
{
    void Print();
}

interface IComparable<T>
{
    int CompareTo(T value);
}

interface IKeyProvider<T>
{
    T GetKey();
}

class Printer<T> where T: IPrintable {...}
class SortedList<T> where T: IComparable<T> {...}
class Dictionary<K,V>
{
    where K: IComparable<K>
    where V: IPrintable, IKeyProvider<K>, new()
}
```

次の例は、型パラメーターの依存関係に循環が発生するため、エラーになります。

```
class Circular<S,T>
{
    where S: T
    where T: S           // Error, circularity in dependency graph
}
```

次に、その他の無効になる状況を示します。

```
class Sealed<S,T>
{
    where S: T
    where T: struct      // Error, T is sealed
}

class A {...}
class B {...}
```

```

class Incompat<S,T>
    where S: A, T
    where T: B          // Error, incompatible class-type constraints
{
    ...
}

class StructWithClass<S,T,U>
    where S: struct, T
    where T: U
    where U: A          // Error, A incompatible with struct
{
    ...
}

```

型 *C* の動的消去は、次のように構築された型 *C<sub>o</sub>* になります。

- *C* が入れ子になった型 *Outer.Inner* の場合、*C<sub>o</sub>* は入れ子になった型 *Outer<sub>o</sub>.Inner<sub>o</sub>* になります。
- *C* が構築された型 *G<A<sup>1</sup>, ..., A<sup>n</sup>>* (ここで、*A<sup>1</sup>*、...、*A<sup>n</sup>* は型引数) の場合、*C<sub>o</sub>* は構築された型 *G<A<sup>1</sup><sub>o</sub>, ..., A<sup>n</sup><sub>o</sub>>* になります。
- *C* が配列型 *E[]* の場合、*C<sub>o</sub>* は配列型 *E<sub>o</sub>[]* になります。
- *C* がポインター型 *E\** の場合、*C<sub>o</sub>* はポインター型 *E<sub>o</sub>\** になります。
- *C* が *dynamic* の場合、*C<sub>o</sub>* は *object* になります。
- それ以外の場合、*C<sub>o</sub>* は *C* になります。

型パラメーター *T* の "実質的な基底クラス" は次のように定義されます。

*R* を型のセットとします。*R* は次のようになります。

- 型パラメーターである *T* のそれぞれの制約に対し、*R* は実質的な基底クラスを格納します。
- 構造型である *T* のそれぞれの制約に対し、*R* は *System.ValueType* を格納します。
- 列挙型である *T* のそれぞれの制約に対し、*R* は *System.Enum* を格納します。
- デリゲート型である *T* のそれぞれの制約に対し、*R* は動的消去を格納します。
- 配列型である *T* のそれぞれの制約に対し、*R* は *System.Array* を格納します。
- クラス型である *T* のそれぞれの制約に対し、*R* は動的消去を格納します。

Then

- *T* に値型の制約がある場合、実質的な基底クラスは *System.ValueType* になります。
- *R* が空の場合、実質的な基底クラスは *object* になります。
- それ以外の場合、*T* の実質的な基底クラスは、セット *R* の最も内側の型になります (6.4.3 を参照)。セットに内側の型がない場合、*T* の実質的な基底クラスは *object* になります。最も内側の型が存在することは、一貫性規則によって保証されます。

型パラメーターが、基本メソッドから制約が継承されるメソッド型パラメーターの場合、実質的な基底クラスは型の置き換え後に計算されます。

これらの規則により、実質的な基底クラスは常に *class-type* であることが保証されます。

型パラメーター *T* の "有効なインターフェイス セット" は、次のように定義されます。

- $T$  に *secondary-constraints* がない場合、有効なインターフェイス セットは空になります。
- $T$  に *interface-type* 制約があつて *type-parameter* 制約がない場合、有効なインターフェイス セットは、その *interface-type* 制約の動的消去のセットになります。
- $T$  に *interface-type* 制約がなく *type-parameter* 制約がある場合、有効なインターフェイス セットは、その *type-parameter* 制約の有効なインターフェイス セットの和集合になります。
- $T$  に *interface-type* 制約と *type-parameter* 制約の両方がある場合、有効なインターフェイス セットは、その *interface-type* 制約の動的消去のセットと *type-parameter* 制約の有効なインターフェイス セットとの和集合になります。

型パラメーターが参照型制約を持つ場合、または実質的な基底クラスが `object` でも `System.ValueType` でもない場合、その型パラメーターは "参照型であることがわかっている" と見なされます。

制約付きの型パラメーター型の値を使用して、制約によって暗黙的に設定されたインスタンス メンバーにアクセスできます。次に例を示します。

```
interface IPrintable
{
    void Print();
}

class Printer<T> where T: IPrintable
{
    void Printone(T x) {
        x.Print();
    }
}
```

$T$  は常に `IPrintable` を実装するように制約されているため、`IPrintable` のメソッドを  $x$  上で直接呼び出すことができます。

### 10.1.6 クラス本体

クラスの *class-body* は、そのクラスのメンバーを定義します。

```
class-body:
{ class-member-declarationsopt }
```

### 10.2 部分型

型宣言は、複数の "部分型宣言" に分けることができます。型宣言はこのセクションの規則に従って各部から構築され、プログラムの残りのコンパイルと実行時の処理では 1 つの宣言として扱われます。

*class-declaration*、*struct-declaration*、または *interface-declaration* は、`partial` 修飾子を含む場合、部分型の宣言を示します。`partial` はキーワードではなく、型宣言で `class`、`struct`、または `interface` キーワードの直前に使用されるか、メソッド宣言で `void` 型の前に使用される場合にのみ修飾子として機能します。その他のコンテキストでは、通常の識別子として使用できます。

部分型宣言の各部分は、`partial` 修飾子を含む必要があります。これは同じ名前で、他の部分と同じ名前空間または型宣言で宣言されている必要があります。`partial` 修飾子は、型宣言の他の部分が他の場所に存在する可能性があることを示しますが、必ず存在することを示すわけではありません。型の宣言が 1 つの場合でも `partial` 修飾子を使用できます。

コンパイル時に各部分を 1 つの型宣言にマージできるように、部分型のすべての部分は一緒にコンパイルする必要があります。部分型によって、既にコンパイルされている型を拡張することはできません。

**partial** 修飾子を使用すると、入れ子になった型を複数の部分に分けて宣言できます。一般的に、入れ子の外側の型も **partial** を使用して宣言し、入れ子の内側の型の各部分は、外側の型の個別の部分として宣言します。

**partial** 修飾子は、デリゲート型または列挙型の宣言では使用できません。

### 10.2.1 属性

部分型の属性は、各部分の属性を未指定の順序で結合して決定されます。複数の部分に属性が配置されている場合は、型に属性を複数回指定した場合と同じです。次の例には、2 つの部分があります。

```
[Attr1, Attr2("hello")]
partial class A {}

[Attr3, Attr2("goodbye")]
partial class A {}
```

これは、次の宣言と同じです。

```
[Attr1, Attr2("hello"), Attr3, Attr2("goodbye")]
class A {}
```

型パラメーターの属性も、同じように結合されます。

### 10.2.2 修飾子

部分型の宣言がアクセシビリティの指定 (**public**、**protected**、**internal**、および **private** 修飾子) を含む場合は、アクセシビリティの指定を含む他のすべての部分と一致する必要があります。部分型にアクセシビリティの指定を含む部分がない場合は、適切な既定のアクセシビリティが型に対して指定されます (3.5.1 を参照)。

入れ子になった型の部分宣言の少なくとも 1 つが **new** 修飾子を含む場合は、入れ子になった型が継承メンバーを隠ぺいした場合の警告は報告されません (3.7.1.2 を参照)。

クラスの部分宣言の少なくとも 1 つが **abstract** 修飾子を含む場合、クラスは抽象と見なされます (10.1.1.1 を参照)。それ以外の場合、クラスは非抽象と見なされます。

クラスの部分宣言の少なくとも 1 つが **sealed** 修飾子を含む場合、クラスはシールクラスと見なされます (10.1.1.2 を参照)。それ以外の場合、クラスはシールされていないクラスと見なされます。

クラスは、抽象とシールの両方になることはできません。

**unsafe** 修飾子が部分型の宣言に使用されている場合は、その特定の部分のみが **unsafe** コンテキストと見なされます (18.1 を参照)。

### 10.2.3 型パラメーターと制約

ジェネリック型が複数の部分に分かれて宣言されている場合は、各部分に型パラメーターが記述されている必要があります。各部分には、同数の型パラメーターが存在し、それぞれの型パラメーターは正しい順序で同じ名前を持つ必要があります。

部分ジェネリック型の宣言が制約 (**where** 句) を含む場合、その制約は、制約を含む他のすべての部分と一致する必要があります。具体的には、制約を含む各部分は同じ型パラメーターの集合に対して制約を持つ必要があり、各型パラメーターの主制約、2 次制約、およびコンストラクター制約の集合が

同じである必要があります。制約の 2 つの集合は、同じメンバーを含む場合、等価です。部分ジェネリック型に、型パラメーターの制約を指定する部分がない場合、その型パラメーターは制約されていないと見なされます。

次の例を参照してください。

```
partial class Dictionary<K,V>
    where K: IComparable<K>
    where V: IKeyProvider<K>, IPersistable
{
    ...
}

partial class Dictionary<K,V>
    where V: IPersistable, IKeyProvider<K>
    where K: IComparable<K>
{
    ...
}

partial class Dictionary<K,V>
{
    ...
}
```

この例では、制約(最初の 2 つ)を含む部分は、型パラメーターの同じ集合に対して実質的に同じ主制約、2 次制約、およびコンストラクター制約の集合を指定しているため、正しいコードです。

#### 10.2.4 基底クラス

部分クラスの宣言が基底クラスの指定を含む場合、その宣言は基底クラスの指定を含む他の宣言と一致する必要があります。部分クラスに基底クラスの指定を含む部分がない場合、基底クラスは `System.Object` になります(10.1.4.1 を参照)。

#### 10.2.5 基本インターフェイス

複数の部分に分かれて宣言された型の基本インターフェイスの集合は、各部分で指定された基本インターフェイスの和集合になります。1 つの基本インターフェイスは、各部分では 1 回だけ指定できますが、同じ基本インターフェイスを複数の部分で指定することはできます。1 つの基本インターフェイスのメンバーの実装は、1 つに限定されます。

次に例を示します。

```
partial class C: IA, IB {...}
partial class C: IC {...}
partial class C: IA, IB {...}
```

ここでは、クラス `C` の基本インターフェイスのセットは `IA`、`IB`、および `IC` です。

一般的に、各部分はその部分に宣言されたインターフェイスの実装を提供しますが、これは必須ではありません。別の部分に宣言されているインターフェイスの実装を提供する場合もあります。

```
partial class X
{
    int IComparable.CompareTo(object o) {...}
}
partial class X: IComparable
{
    ...
}
```

## 10.2.6 メンバー

部分メソッド(10.2.7を参照)を除いて、複数の部分に分かれて宣言された型のメンバーの集合は、単純に各部分に宣言されたメンバーの和集合です。型宣言のすべての部分の本体は同じ宣言空間(3.3を参照)を共有し、各メンバーのスコープ(3.7を参照)はすべての部分の本体まで及びます。メンバーのアクセシビリティドメインは、常にその外側の型のすべての部分を含み、ある部分で宣言された **private** メンバーに別の部分から自由にアクセスできます。型の複数の部分で同じメンバーを宣言すると、そのメンバーが **partial** 修飾子を持つ型の場合を除き、コンパイルエラーになります。

```
partial class A
{
    int x;                      // Error, cannot declare x more than once
    partial class Inner          // Ok, Inner is a partial type
    {
        int y;
    }
}

partial class A
{
    int x;                      // Error, cannot declare x more than once
    partial class Inner          // Ok, Inner is a partial type
    {
        int z;
    }
}
```

型内のメンバーの順序が C# コードで意味を持つことはほとんどありませんが、他の言語や環境とのインターフェイスでは意味を持つ場合があります。このような場合、複数の部分に宣言された型内でのメンバーの順序は、未定義です。

## 10.2.7 部分メソッド

部分メソッドは、型宣言のある部分で定義し、別の部分で実装できます。実装は省略可能です。どの部分にも部分メソッドを実装しない場合、部分メソッドの宣言とそのすべての呼び出しは、各部分の組み合わせから作成される型宣言から削除されます。

部分メソッドはアクセス修飾子を定義できませんが、暗黙的に **private** となります。その戻り値の型は **void** とする必要があり、そのパラメーターに **out** 修飾子を含めることはできません。識別子 **partial** は、**void** 型の直前に使用される場合にのみメソッドの宣言で特殊なキーワードとして認識されます。それ以外の場合は、通常の識別子として使用できます。部分メソッドは、インターフェイスメソッドを明示的に実装できません。

部分メソッドの宣言には 2 種類があります。メソッド宣言の本体がセミコロンの場合、その宣言は "部分メソッド定義宣言" と呼ばれます。本体が *block* で指定されている場合、その宣言は "部分メソッド実装宣言" と呼ばれます。型宣言の各部分を通して、特定シグネチャの部分メソッド定義宣言 1 つと、特定シグネチャの部分メソッド実装宣言 1 つのみを含めることができます。部分メソッド実装宣言が指定されている場合、対応する部分メソッド定義宣言が存在し、これらの宣言が以下に指定されたように一致する必要があります。

- 宣言が同じ修飾子(同じ順序でなくても可)、メソッド名、型パラメーターの数、およびパラメーターの数を持つ必要があります。
- 宣言内で対応するパラメーターが同じ修飾子(同じ順序でなくても可)および同じ型(型パラメータ名の相違を除く)を持つ必要があります。

- 宣言内で対応する型パラメーターが同じ制約(型パラメーター名の相違を除く)を持つ必要があります。

部分メソッド実装宣言は、対応する部分メソッド定義宣言と同じ部分に使用できます。

オーバーロードの解決には、部分メソッド定義宣言のみが関係します。したがって、実装宣言が定義されているかどうかに応じて、呼び出し式が部分メソッドの呼び出しに解決される場合があります。部分メソッドでは常に `void` が返されるため、このような呼び出し式は必ず式ステートメントになります。さらに、部分メソッドは暗黙的に `private` であるため、このようなステートメントは、部分メソッドが宣言された型宣言のいずれかの部分に存在します。

部分型宣言の各部に特定部分メソッドに対する実装宣言が含まれない場合、それを呼び出す式ステートメントは、組み合わされた型宣言から単に削除されます。したがって、構成するすべての式をはじめとする呼び出し式は、実行時に何の効果もありません。部分メソッド自身も削除され、組み合わされた型宣言のメンバーになりません。

特定部分メソッドに対する実装宣言が存在する場合、部分メソッドの呼び出しが維持されます。部分メソッドにより、次の点を除いて、部分メソッド実装宣言と同様のメソッド宣言が行われます。

- `partial` 修飾子が含まれません。
- 得られるメソッド宣言の属性は、部分メソッド定義宣言と部分メソッド実装宣言の属性を不特定の順序で組み合わせたものです。重複は削除されません。
- 得られるメソッド宣言のパラメーターの属性は、部分メソッド定義宣言と部分メソッド実装宣言の該当パラメーターの属性を不特定の順序で組み合わせたものです。重複は削除されません。

部分メソッド `M` に対して実装宣言がなく、定義宣言のみが指定されている場合、次の制限事項があります。

- メソッドのデリゲート(7.6.10.5を参照)を作成するとコンパイルエラーになります。
- 式のツリー型(6.5.2を参照)に変換された匿名関数の内側で `M` を参照するとコンパイルエラーになります。
- `M` の呼び出しの一部として発生する式は、確実な代入の状態(5.3を参照)に影響を与えません。これはコンパイルエラーにつながる可能性があります。
- `M` をアプリケーションのエントリ ポイント(3.1を参照)に使用できません。

部分メソッドは、型宣言のある部分で別の部分の動作(ツールによって生成されたものなど)をカスタマイズする場合に役立ちます。次の部分クラスの宣言があるとします。

```
partial class Customer
{
    string name;
    public string Name {
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }
    }
}
```

```

        partial void OnNameChanging(string newName);
        partial void OnNameChanged();
    }

```

このクラスがその他の部分なしでコンパイルされる場合、部分メソッド定義宣言とその呼び出しが削除され、これにより組み合わせて得られるクラス宣言は次と同じになります。

```

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

```

ここで、部分メソッドの実装宣言となる別の部分が指定されている場合を想定してみます。

```

partial class Customer
{
    partial void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }
    partial void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}

```

この場合、組み合わせて得られるクラス宣言は次と同じになります。

```

class Customer
{
    string name;
    public string Name {
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }
    }
    void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }
    void OnNamechanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}

```

## 10.2.8 名前のバインディング

拡張できる型の各部分は同じ名前空間に宣言する必要がありますが、一般的に、部分は異なる名前空間宣言内に記述されます。したがって、部分ごとに別の `using` ディレクティブ (9.4 を参照) が存在す

る場合があります。ある部分の簡易名 (7.5.2 を参照) を解釈するときは、その部分の外側の名前空間宣言にある `using` ディレクティブのみが考慮されます。これにより、次のように、同じ識別子が部分によって別の意味を持つことがあります。

```
namespace N
{
    using List = System.Collections.ArrayList;
    partial class A
    {
        List x;           // x has type System.Collections.ArrayList
    }
}
namespace N
{
    using List = widgets.LinkedList;
    partial class A
    {
        List y;           // y has type widgets.LinkedList
    }
}
```

### 10.3 クラスのメンバー

クラスのメンバーは、*class-member-declaration* で導入されたメンバーと、直接基底クラスから継承したメンバーで構成されます。

```
class-member-declarations:
    class-member-declaration
    class-member-declarations  class-member-declaration

class-member-declaration:
    constant-declaration
    field-declaration
    method-declaration
    property-declaration
    event-declaration
    indexer-declaration
    operator-declaration
    constructor-declaration
    destructor-declaration
    static-constructor-declaration
    type-declaration
```

クラス型のメンバーは次のカテゴリに分けられます。

- 定数。 クラスに関連付けられた定数値を表します (10.4 を参照)。
- フィールド。 クラスの変数です (10.5 を参照)。
- メソッド。 クラスによって実行できる演算とアクションを実装します (10.6 を参照)。
- プロパティ。 自身の読み取りや書き込みに関連付けられた名前付きの特性およびアクションを定義します (10.7 を参照)。
- イベント。 クラスによって生成できる通知を定義します (10.8 を参照)。

- インデクサー。クラスのインスタンスに、配列と構文的に同じ方法でインデックスを付けることができます (10.9 を参照)。
- 演算子。クラスのインスタンスに適用できる式演算子を定義します (10.10 を参照)。
- インスタンス コンストラクター。クラスのインスタンスを初期化するのに必要なアクションを実装します (10.11 を参照)。
- デストラクター。クラスのインスタンスが破棄される前に実行するアクションを実装します (10.13 を参照)。
- 静的コンストラクター。クラスを初期化するのに必要なアクションを実装します (10.12 を参照)。
- 型。クラスに対してローカルな型を表します (10.3.8 を参照)。

実行可能なコードを含むことのできるメンバーは、クラス型の *function members* と総称されます。クラス型の関数メンバーは、そのクラス型のメソッド、プロパティ、イベント、インデクサー、演算子、インスタンス コンストラクター、デストラクター、静的コンストラクターです。

*class-declaration* によって新しい宣言空間が作成され (3.3 を参照)、*class-declaration* のすぐ内側にある *class-member-declarations* により、この宣言空間に新しいメンバーが導入されます。*class-member-declaration* には次の規則が適用されます。

- インスタンス コンストラクター、デストラクター、および静的コンストラクターは、すぐ外側のクラスと同じ名前である必要があります。その他すべてのメンバーは、すぐ外側のクラスの名前とは別の名前である必要があります。
- 定数、フィールド、プロパティ、イベント、型の各名前は、同じクラス内で宣言されている他のすべてのメンバーと異なる名前である必要があります。
- メソッドの名前は、同じクラスで宣言されている他のすべての非メソッドと異なる名前である必要があります。さらに、メソッドのシグネチャ (3.6 を参照) は、同じクラスで宣言されている他のすべてのメソッドのシグネチャと異なっている必要があります。同じクラスで宣言される 2 つのメソッドに、`ref` と `out` の違いしかないシグネチャを指定することはできません。
- インスタンス コンストラクターのシグネチャは、同じクラスで宣言されている他のすべてのインスタンス コンストラクターのシグネチャと異なっている必要があります。同じクラスで宣言される 2 つのコンストラクターに、`ref` と `out` の違いしかないシグネチャを指定することはできません。
- インデクサーのシグネチャは、同じクラスで宣言されている他のすべてのインデクサーのシグネチャと異なっている必要があります。
- 演算子のシグネチャは、同じクラスで宣言されている他のすべての演算子と異なっている必要があります。

クラス型の継承されたメンバー (10.3.3 を参照) は、クラスの宣言空間には含まれていません。したがって、派生クラスでは、継承メンバーと名前またはシグネチャが同じメンバーを宣言できます。ただし実際は、このメンバーによって継承メンバーが隠ぺいされます。

### 10.3.1 インスタンス型

各クラス宣言には、"インスタンス型" というバインドされた型 (4.4.3 を参照) が関連付けられています。ジェネリック クラス宣言では、インスタンス型は、型宣言から構築された型 (4.4 を参照) を作成

することによって形成され、指定された型引数がそれぞれ対応する型パラメーターになります。インスタンス型は型パラメーターを使用するため、型パラメーターがスコープ内にある場合、つまりクラス宣言内にある場合にのみ使用できます。インスタンス型は、クラス宣言内に記述されたコードでは `this` の型です。非ジェネリッククラスの場合、インスタンス型は単に宣言されたクラスです。次の例は、いくつかのクラス宣言とそのインスタンス型を示します。

```
class A<T>                                // instance type: A<T>
{
    class B {}                            // instance type: A<T>.B
    class C<U> {}                      // instance type: A<T>.C<U>
}
class D {}                                // instance type: D
```

### 10.3.2 構築された型のメンバー

構築された型の非継承メンバーは、メンバー宣言の各 *type-parameter* を構築された型の対応する *type-argument* で置き換えることで取得されます。この置き換えは、型宣言のセマンティックの意味に基づいて行われ、単なるテキスト代入ではありません。

たとえば、次のようなジェネリッククラス宣言について考えます。

```
class Gen<T,U>
{
    public T[,] a;
    public void G(int i, T t, Gen<U,T> gt) {...}
    public U Prop { get {...} set {...} }
    public int H(double d) {...}
}
```

構築された型 `Gen<int[], IComparable<string>>` には、次のメンバーがあります。

```
public int[][] a;
public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}
public IComparable<string> Prop { get {...} set {...} }
public int H(double d) {...}
```

ジェネリッククラス宣言 `Gen` のメンバー `a` の型は、"T の 2 次元配列" です。したがって、上記の構築された型のメンバー `a` の型は、"int の 1 次元配列の 2 次元配列"、つまり `int[][]` になります。

インスタンス関数メンバー内では、`this` の型は、それを含む宣言のインスタンス型(10.3.1を参照)です。

ジェネリッククラスのすべてのメンバーは、外側のクラスの型パラメーターを直接、または構築された型の一部として使用できます。実行時に特定のクローズ構築型(4.4.2を参照)を使用すると、使用された型パラメーターは、構築された型に指定した実際の型引数に置き換えられます。次に例を示します。

```
class C<V>
{
    public V f1;
    public C<V> f2 = null;
```

```

    public C(v x) {
        this.f1 = x;
        this.f2 = this;
    }
}

class Application
{
    static void Main() {
        C<int> x1 = new C<int>(1);
        Console.WriteLine(x1.f1);      // Prints 1
        C<double> x2 = new C<double>(3.1415);
        Console.WriteLine(x2.f1);      // Prints 3.1415
    }
}

```

### 10.3.3 繙承

クラスは、直接基底クラス型のメンバーを “**継承**” します。継承とは、基底クラスのインスタンス コンストラクター、デストラクター、および静的コンストラクターを除いて、クラスが暗黙的に直接基底クラス型のすべてのメンバーを格納することを意味します。継承の重要な点は次のとおりです。

- 継承は推移的です。C が B から派生し、B が A から派生する場合、C は B で宣言されたメンバーと A で宣言されたメンバーを継承します。
- 派生クラスは、直接基底クラスを “**拡張**” します。派生クラスは、継承するメンバーに新しいメンバーを追加できますが、継承したメンバーの定義は削除できません。
- インスタンス コンストラクター、デストラクター、および静的コンストラクターは継承されませんが、その他すべてのメンバーは、宣言されたアクセシビリティにかかわらず継承されます(3.5 を参照)。ただし、宣言されたアクセシビリティによっては、継承されたメンバーは派生クラスではアクセスできないことがあります。
- 派生クラスは、新しいメンバーと同じ名前やシグネチャで宣言することで、継承したメンバーを “**隠ぺい**”(3.7.1.2 を参照) できます。ただし、継承したメンバーが隠ぺいされても、そのメンバーは削除されません。派生クラスを通じてそのメンバーに直接アクセスできなくなるだけです。
- クラスのインスタンスには、クラスに宣言されたすべてのインスタンス フィールドと基底クラスが含まれます。また、派生クラス型から基底クラス型への暗黙的な変換(6.1.6 を参照)が存在します。したがって、派生クラスのインスタンスへの参照は、基底クラスのインスタンスへの参照として処理できます。
- クラスは、仮想メソッド、プロパティ、およびインデクサーを宣言できます。派生クラスはこれらの関数のメンバーの実装をオーバーライドできます。これにより、クラスはポリモーフィックな動作ができます。この動作の中で、関数のメンバー呼び出しによって実行されるアクションは、その関数のメンバーが呼び出されるインスタンスの実行時の型によって異なります。

構築クラス型の継承メンバーは、直接基底クラス型(10.1.4.1 を参照)のメンバーです。これは、*base-class-specification* に使用される対応する各型パラメーターを構築された型の型引数で置き換えることで得られます。これらのメンバーは、メンバー宣言の各 *type-parameter* を *base-class-specification* の対応する *type-argument* で置き換えることで変換されます。

```

class B<U>
{
    public U F(long index) {...}
}

```

```
class D<T>: B<T[]>
{
    public T G(string s) {...}
```

上記の例では、構築された型 `D<int>` に、型パラメーター `T` を型引数 `int` で置き換えて取得した非継承メンバー `public int G(string s)` があります。また `D<int>` には、クラス宣言 `B` から継承したメンバーもあります。この継承メンバーは、最初に基底クラス指定 `B<T[]>` の `T` を `int` で置き換えることで、`D<int>` の基底クラス型 `B<int[]>` を決定することにより決定されます。次に、`B` に対する型引数として `int[]` を `public UF(long index)` の `U` に置き換えて、`public int[] F(long index)` 継承メンバーを生成します。

#### 10.3.4 new 修飾子

*class-member-declaration* では、継承したメンバーと同じ名前またはシグネチャでメンバーを宣言できます。この場合、派生クラスのメンバーが基底クラスのメンバーを "隠ぺいする" と言います。継承したメンバーを隠ぺいしてもエラーとは見なされませんが、コンパイラは警告を出します。警告を出さないようにするには、派生クラスのメンバーの宣言に `new` 修飾子を含めます。これで、派生メンバーが基本メンバーを隠ぺいすることを明示できます。これは 3.7.1.2 で詳細に説明しています。

継承メンバーを隠ぺいしない宣言に `new` 修飾子が含まれる場合は、そのことを表す警告が出されます。`new` 修飾子を削除すると、この警告は出されません。

#### 10.3.5 アクセス修飾子

*class-member-declaration* では、`public`、`protected internal`、`protected`、`internal`、または `private` の 5 種類の宣言されたアクセシビリティ (3.5.1 を参照) のうちいずれか 1 つを設定できます。`protected internal` の組み合わせを除いて、複数のアクセス修飾子を指定するとコンパイルエラーが発生します。*class-member-declaration* にアクセス修飾子が含まれない場合は、`private` が使用されます。

#### 10.3.6 構成要素型

メンバーの宣言で使用される型は、そのメンバーの構成要素型と呼ばれます。構成要素型には、定数、フィールド、プロパティ、イベント、インデクサーの型、メソッドや演算子の戻り値の型、メソッド、インデクサー、演算子、インスタンス コンストラクターのパラメーターの型などがあります。メンバーの構成要素型は、少なくともメンバー自体と同程度にアクセスできる必要があります (3.5.4 を参照)。

#### 10.3.7 静的メンバーとインスタンス メンバー

クラスのメンバーは、"静的メンバー" か "インスタンス メンバー" のどちらかになります。一般的に、静的メンバーがクラス型に属し、インスタンス メンバーがオブジェクト (クラス型のインスタンス) に属すると考えると便利です。

フィールド、メソッド、プロパティ、イベント、演算子、またはコンストラクターの宣言に `static` 修飾子が含まれる場合は、静的メンバーが宣言されます。さらに、定数宣言や型宣言では、暗黙に静的メンバーが宣言されます。静的メンバーには次の特性があります。

- 静的メンバー `M` が `E.M` 形式の *member-access* (7.6.4 を参照) で参照される場合、`E` は `M` を含む型を表す必要があります。`E` がインスタンスを表すとコンパイルエラーになります。

- 静的フィールドは、特定のクローズ クラス型のすべてのインスタンスにより共有される 1 つの格納場所を正確に表します。特定のクローズ クラス型のインスタンスがいくつ作成されても、静的フィールドのコピーは 1 つだけです。
- 静的関数のメンバー(メソッド、プロパティ、イベント、演算子、またはコンストラクター)は、特定のインスタンスには作用しないので、静的関数のメンバーで `this` を参照するとコンパイルエラーになります。

フィールド、メソッド、プロパティ、イベント、インデクサー、コンストラクター、またはデストラクターの宣言に `static` 修飾子が含まれない場合は、インスタンス メンバーが宣言されます。インスタンス メンバーは非静的メンバーと表現されることもあります。インスタンス メンバーには次の特性があります。

- インスタンス メンバー `M` が `E.M` 形式の *member-access* (7.6.4 を参照) で参照される場合、`E` は `M` を含む型のインスタンスを表す必要があります。`E` が型を表すとバインディング エラーになります。
- クラスのどのインスタンスにも、そのクラスのすべてのインスタンス フィールドのセットが格納されます。
- インスタンス関数のメンバー(メソッド、プロパティ、インデクサー、インスタンス コンストラクター、またはデストラクター)は、クラスの指定されたインスタンスに作用し、このインスタンスは `this` (7.6.7 を参照) としてアクセスできます。

静的メンバーとインスタンス メンバーへのアクセス規則を次のコード例で示します。

```
class Test
{
    int x;
    static int y;

    void F() {
        x = 1;           // Ok, same as this.x = 1
        y = 1;           // Ok, same as Test.y = 1
    }

    static void G() {
        x = 1;           // Error, cannot access this.x
        y = 1;           // Ok, same as Test.y = 1
    }

    static void Main() {
        Test t = new Test();
        t.x = 1;          // Ok
        t.y = 1;          // Error, cannot access static member through instance
        Test.x = 1;        // Error, cannot access instance member through type
        Test.y = 1;        // Ok
    }
}
```

`F` メソッドは、インスタンス関数のメンバー内では *simple-name* (7.6.2 を参照) を使ってインスタンス メンバーと静的メンバーの両方にアクセスできることを示しています。`G` メソッドは、静的関数のメンバー内では *simple-name* を使ってインスタンス メンバーにアクセスするとコンパイルエラーになることを示しています。`Main` メソッドは、*member-access* (7.6.4 を参照) 内ではインスタンス メンバーにはインスタンス、静的メンバーには型を使ってアクセスする必要があることを示しています。

### 10.3.8 入れ子になった型

クラス宣言や構造体宣言内で宣言された型は、"入れ子になった型"と呼ばれます。コンパイル単位や名前空間内で宣言された型は、"入れ子になっていない型"と呼ばれます。

次に例を示します。

```
using System;
class A
{
    class B
    {
        static void F()
        {
            Console.WriteLine("A.B.F");
        }
    }
}
```

クラス **B** はクラス **A** 内で宣言されているので、入れ子になった型です。クラス **A** は、コンパイル単位内で宣言されているので入れ子になっていない型です。

#### 10.3.8.1 完全修飾名

入れ子になった型の完全修飾名 (3.8.1 を参照) は **S.N** という形式です。ここで、**S** は型 **N** が宣言された型の完全修飾名です。

#### 10.3.8.2 宣言されたアクセシビリティ

入れ子になっていない型には、**public** または **internal** の宣言されたアクセシビリティを設定できます。既定では、**internal** の宣言されたアクセシビリティが設定されています。入れ子になった型のアクセシビリティの宣言は、これらの形式と追加の形式を使うことができます。どのような形式を使うかは、宣言を包含する型がクラスと構造体のどちらであるかによって決まります。

- クラス内で宣言された入れ子になった型は、5種類の宣言されたアクセシビリティの形式 (**public**、**protected internal**、**protected**、**internal**、または **private**) のいずれかを使用できます。他のクラス メンバーと同様に、宣言されたアクセシビリティの既定値は **private** です。
- 構造体内で宣言された入れ子になった型は、3種類の宣言されたアクセシビリティの形式 (**public**、**internal**、または **private**) のいずれかを使用できます。他の構造体メンバーと同様に、宣言されたアクセシビリティの既定値は **private** です。

次の例を参照してください。

```
public class List
{
    // Private data structure
    private class Node
    {
        public object Data;
        public Node Next;

        public Node(object data, Node next)
        {
            this.Data = data;
            this.Next = next;
        }
    }

    private Node first = null;
    private Node last = null;
}

// Public interface
```

```

    public void AddToFront(object o) {...}
    public void AddToBack(object o) {...}
    public object RemoveFromFront() {...}
    public object RemoveFromBack() {...}
    public int Count { get {...} }
}

```

この例では、入れ子になったプライベートクラス **Node** を宣言しています。

#### 10.3.8.3 隠ぺい

入れ子になった型は、基本メンバーを隠ぺい (3.7.1 を参照) できます。入れ子になった型の宣言には **new** 修飾子を使用できるので、隠ぺいを明示的に表現できます。次の例を参照してください。

```

using System;
class Base
{
    public static void M() {
        Console.WriteLine("Base.M");
    }
}
class Derived: Base
{
    new public class M
    {
        public static void F() {
            Console.WriteLine("Derived.M.F");
        }
    }
}
class Test
{
    static void Main() {
        Derived.M.F();
    }
}

```

これは、入れ子になったクラス **M** が、**Base** で定義されたメソッド **M** を隠ぺいすることを示します。

#### 10.3.8.4 this アクセス

入れ子になった型とそれを包含する型の間には、*this-access* (7.6.7 を参照) に関して特殊な関係はありません。特に、入れ子になった型の中で **this** を使って、包含する型のインスタンス メンバーを参照することはできません。入れ子になった型が、その型を包含する型のインスタンス メンバーにアクセスする必要がある場合は、入れ子になった型のコンストラクター引数として、包含する型のインスタンスの **this** を使用することでアクセスできます。次に例を示します。

```

using System;
class C
{
    int i = 123;
    public void F() {
        Nested n = new Nested(this);
        n.G();
    }
}

```

```

public class Nested
{
    C this_c;
    public Nested(C c) {
        this_c = c;
    }
    public void G() {
        Console.WriteLine(this_c.i);
    }
}
class Test
{
    static void Main() {
        C c = new C();
        c.F();
    }
}

```

このコード例は、その手法です。C のインスタンスが Nested のインスタンスを作成し、その後、C のインスタンス メンバーにアクセスできるように、それ自体の this を Nested のコンストラクターに渡します。

#### 10.3.8.5 包含する型の private メンバーおよび protected メンバーへのアクセス

入れ子になった型は、それを包含する型のメンバーのうち、宣言されたアクセシビリティが **private** および **protected** であるメンバーも含めて、包含する型にアクセスできるメンバーのすべてにアクセスできます。次の例を参照してください。

```

using System;
class C
{
    private static void F() {
        Console.WriteLine("C.F");
    }
    public class Nested
    {
        public static void G() {
            F();
        }
    }
}
class Test
{
    static void Main() {
        C.Nested.G();
    }
}

```

これは、入れ子になったクラス Nested を含む C クラスを示しています。Nested の中で、メソッド G は C で定義された静的メソッド F を呼び出し、F はアクセシビリティ private が宣言されています。

また、入れ子になった型は、それを包含する型の基本型で定義された protected メンバーにもアクセスできます。次に例を示します。

```
using System;
```

```

class Base
{
    protected void F() {
        Console.WriteLine("Base.F");
    }
}
class Derived: Base
{
    public class Nested
    {
        public void G() {
            Derived d = new Derived();
            d.F();           // ok
        }
    }
}
class Test
{
    static void Main() {
        Derived.Nested n = new Derived.Nested();
        n.G();
    }
}

```

入れ子になったクラス `Derived.Nested` は、`Derived` のインスタンスから呼び出すことで、`Derived` の基底クラス `Base` で定義された `protected` メソッド `F` にアクセスします。

#### 10.3.8.6 ジェネリック クラスの入れ子になった型

ジェネリック クラス宣言には、入れ子になった型の宣言を含むことができます。外側のクラスの型パラメーターを入れ子になった型の中で使用できます。入れ子になった型の宣言には、入れ子になった型に対してのみ適用される追加の型パラメーターを含むことができます。

ジェネリック クラス宣言内に含まれるすべての型宣言は、暗黙的にジェネリック型宣言になります。ジェネリック型の内側に入れ子になった型への参照を記述するときは、外側の構築された型を、その型引数も含めて指定する必要があります。しかし、外側のクラスの中からは、入れ子になった型を修飾なしで使用でき、入れ子になった型を構築するときは、外側のクラスのインスタンス型を暗黙的に使用できます。次の例は、`Inner` から作成された構築型を参照する 3 つの適切な方法を示します。最初の 2 つの方法は等価です。

```

class Outer<T>
{
    class Inner<U>
    {
        public static void F(T t, U u) {...}

        static void F(T t) {
            Outer<T>.Inner<string>.F(t, "abc");      // These two statements have
            Inner<string>.F(t, "abc");                // the same effect
            Outer<int>.Inner<string>.F(3, "abc");   // This type is different
            Outer.Inner<string>.F(t, "abc");         // Error, Outer needs type arg
        }
}

```

これは適切なプログラミングスタイルではありませんが、次のように、入れ子になった型の型パラメーターは、外側の型で宣言されたメンバーまたは型パラメーターを隠蔽できます。

```

class Outer<T>
{
    class Inner<T>      // valid, hides Outer's T
    {
        public T t;      // Refers to Inner's T
    }
}

```

### 10.3.9 予約済みのメンバー名

基になる C# 実行時実装を簡単にするために、プロパティ、イベント、またはインデクサーのソースメンバー宣言ごとに、実装はメンバー宣言、メンバー名、およびメンバーの型の種類に基づいて、2つのメソッドシグネチャを予約する必要があります。プログラムで、これらの予約されたシグネチャの 1 つに一致するシグネチャを持つメンバーを宣言すると、基になる実行時実装が予約されたシグネチャを使用しない場合でもコンパイルエラーになります。

予約された名前は、それによって宣言が導入されることはないとため、メンバー検索には関与しません。ただし、宣言に関連する予約されたメソッドシグネチャは、継承 (10.3.3 を参照) に関与し、`new` 修飾子 (10.3.4 を参照) で非表示にできます。

これらの名前の予約には、次の 3 つの目的があります。

- 基になる実装機能が、C# 言語機能への `get` アクセスや `set` アクセスに対して通常の識別子をメソッド名として使用できるようにする。
- 他の言語が、C# 言語機能への `get` アクセスや `set` アクセスに対して通常の識別子をメソッド名として相互運用できるようにする。
- すべての C# 実装で、予約メンバー名の特性について一貫性を保つことで、ある準拠コンパイラによって承認されたソースが別のコンパイラによって承認されるようにする。

デストラクター (10.13 を参照) の宣言によっても、シグネチャが予約されます (10.3.9.4 を参照)。

#### 10.3.9.1 プロパティ用に予約済みのメンバー名

型 `T` のプロパティ `P` (10.7 を参照) については、次のシグネチャが予約されています。

```

T get_P();
void set_P(T value);

```

プロパティが読み取り専用または書き込み専用であっても、両方のシグネチャが予約されます。

次に例を示します。

```

using System;
class A
{
    public int P {
        get { return 123; }
    }
}
class B: A
{
    new public int get_P() {
        return 456;
    }
    new public void set_P(int value) {
    }
}

```

```

class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        Console.WriteLine(a.P);
        Console.WriteLine(b.P);
        Console.WriteLine(b.get_P());
    }
}

```

クラス A は読み取り専用のプロパティ P を定義しているので、`get_P` メソッドと `set_P` メソッドのシグネチャが予約されます。クラス B はクラス A から派生して、予約された両方のシグネチャを非表示にします。この例では、次のように出力されます。

```

123
123
456

```

### 10.3.9.2 イベント用に予約済みのメンバー名

デリゲート型 T のイベント E (10.8 を参照) については、次のシグネチャが予約されています。

```

void add_E(T handler);
void remove_E(T handler);

```

### 10.3.9.3 インデクサー用に予約済みのメンバー名

パラメーターリスト L を持つ型 T のインデクサー (10.9 を参照) については、次のシグネチャが予約されています。

```

T get_Item(L);
void set_Item(L, T value);

```

インデクサーが読み取り専用または書き込み専用であっても、両方のシグネチャが予約されます。

さらに、メンバー名 Item が予約されます。

### 10.3.9.4 デストラクター用に予約済みのメンバー名

デストラクター (10.13 を参照) を格納するクラスについては、次のシグネチャが予約されています。

```

void Finalize();

```

## 10.4 定数

"定数" は、コンパイル時に算出できる定数値を表すクラス メンバーです。constant-declaration によって、指定された型の 1 つ以上の定数が導入されます。

*constant-declaration:*  
*attributes<sub>opt</sub>* *constant-modifiers<sub>opt</sub>* **const** *type* *constant-declarators* ;

*constant-modifiers:*  
*constant-modifier*  
*constant-modifiers* *constant-modifier*

*constant-modifier:*  
**new**  
**public**  
**protected**  
**internal**  
**private**

```

constant-declarators:
  constant-declarator
  constant-declarators , constant-declarator

constant-declarator:
  identifier = constant-expression

```

*constant-declaration* には、*attributes* (17 を参照) の集合、**new** 修飾子 (10.3.4 を参照)、および 4 つのアクセス修飾子の有効な組み合わせ (10.3.5 を参照) を指定できます。属性および修飾子は、*constant-declaration* で宣言されるすべてのメンバーに適用されます。定数が静的メンバーと見なされる場合でも、*constant-declaration* に **static** 修飾子は必要ありません。また、使用することもできません。1 つの定数宣言内で同じ修飾子を複数回使用すると、エラーになります。

*constant-declaration* の *type* は、宣言で導入されるメンバーの型を指定します。この型の後には、*constant-declarator* のリストが続き、各宣言子により新しいメンバーが導入されます。*constant-declarator* は、メンバーを指定する *identifier* で構成されます。"=" トークンと、メンバーの値を指定する *constant-expression* (7.19 を参照) をその後に続けます。

定数の宣言で指定する型は、**sbyte**、**byte**、**short**、**ushort**、**int**、**uint**、**long**、**ulong**、**char**、**float**、**double**、**decimal**、**bool**、**string**、*enum-type*、または *reference-type* である必要があります。各 *constant-expression* は、結果となる値がターゲットの型か、暗黙の型変換 (0 を参照) によってターゲットの型に変換できる型である必要があります。

定数の *type* は、少なくとも定数自体と同程度にアクセス可能である必要があります (3.5.4 を参照)。

定数の値は、*simple-name* (7.6.2 を参照) または *member-access* (7.6.4 を参照) を使って式に取得されます。

定数は、*constant-expression* に含めることができます。このため、*constant-expression* を必要とする構成要素に定数を使用できます。定数式を必要とする構成要素の例には、**case** ラベル、**goto case** ステートメント、**enum** メンバー宣言、属性、その他の定数宣言などがあります。

7.19 で説明されているように、*constant-expression* は、コンパイル時に完全に評価できる式です。**string** 以外の *reference-type* で非 **null** 値を作成する唯一の方法は **new** 演算子を適用することであり、**new** 演算子は *constant-expression* で許可されていないので、**string** 以外の *reference-types* の定数が取り得る値は **null** です。

定数値のシンボル名が必要でありながら、値の型を定数宣言で使用できない場合、またはコンパイル時に *constant-expression* によって値を計算できない場合は、**readonly** フィールド (10.5.2 を参照) を代用できます。

複数の定数を宣言する定数宣言は、同じ属性、修飾子、および型の单一の定数を複数回宣言するのと同じです。次に例を示します。

```

class A
{
    public const double X = 1.0, Y = 2.0, Z = 3.0;
}

```

上記のコードは、次のコードと同じです。

```
class A
{
    public const double X = 1.0;
    public const double Y = 2.0;
    public const double Z = 3.0;
}
```

定数は、依存関係が循環しない限り、同じプログラム内の他の定数に依存できます。コンパイラは定数宣言を適切な順序に自動的に並べ替えて評価します。次に例を示します。

```
class A
{
    public const int X = B.Z + 1;
    public const int Y = 10;
}
class B
{
    public const int Z = A.Y + 1;
}
```

コンパイラはまず `A.Y` を評価し、次に `B.Z` を評価します。最後に `A.X` を評価して、値 10、11、および 12 を生成します。定数宣言は、他のプログラムの定数に依存できますが、依存関係は一方向に限られます。上の例では、`A` と `B` が別個のプログラムで宣言された場合は、`A.X` が `B.Z` に依存することはできても、`B.Z` は同時には `A.Y` に依存できません。

## 10.5 フィールド

"フィールド" は、オブジェクトまたはクラスに関連付けられた変数を表すメンバーです。*field-declaration* は、指定された型の 1 つ以上のフィールドを導入します。

```
field-declaration:
  attributesopt field-modifiersopt type variable-declarators ;

field-modifiers:
  field-modifier
  field-modifiers field-modifier

field-modifier:
  new
  public
  protected
  internal
  private
  static
  readonly
  volatile

variable-declarators:
  variable-declarator
  variable-declarators , variable-declarator

variable-declarator:
  identifier
  identifier = variable-initializer

variable-initializer:
  expression
  array-initializer
```

*field-declaration* には、属性(17 を参照)のセット、`new` 修飾子(10.3.4 を参照)、4つのアクセス修飾子の有効な組み合わせ(10.3.5 を参照)、および `static` 修飾子(10.5.1 を参照)を指定できます。さらに、*field-declaration* には `readonly` 修飾子(10.5.2 を参照)または `volatile` 修飾子(10.5.3 を参照)を含めることができますが、両方を指定することはできません。属性および修飾子は、*field-declaration* で宣言されるすべてのメンバーに適用されます。1つのフィールド宣言内で同じ修飾子を複数回使用すると、エラーになります。

*field-declaration* の *type* は、宣言で導入されるメンバーの型を指定します。この型の後には、*variable-declarator* のリストが続き、各宣言子により新しいメンバーが導入されます。*variable-declarator* は、メンバーを指定する *identifier* で構成されます。オプションで "=" トークンと、そのメンバーの初期値を指定する *variable-initializer*(10.5.5 を参照)を続けることもできます。

フィールドの *type* は、少なくともフィールド自体と同程度にアクセス可能である必要があります(3.5.4 を参照)。

フィールドの値は、式で *simple-name*(7.6.2 を参照)または *member-access*(7.6.4 を参照)を使用することで取得されます。非 `readonly` フィールドの値は、*assignment*(7.17 を参照)を使って変更されます。読み取り専用でないフィールドの値は、後置インクリメント演算子と後置デクリメント演算子(7.6.9 を参照)、および前置インクリメント演算子と前置デクリメント演算子(7.7.5 を参照)を使って取得および変更できます。

複数のフィールドを宣言するフィールド宣言は、同じ属性、修飾子、および型の单一のフィールドを複数回宣言するのと同じです。次に例を示します。

```
class A
{
    public static int x = 1, y, z = 100;
```

上記のコードは、次のコードと同じです。

```
class A
{
    public static int x = 1;
    public static int y;
    public static int z = 100;
```

### 10.5.1 静的フィールドとインスタンス フィールド

フィールド宣言に `static` 修飾子が含まれる場合、宣言によって導入されるフィールドは "静的フィールド" です。`static` 修飾子がない場合、宣言によって導入されるフィールドは "インスタンスフィールド" です。静的フィールドとインスタンスフィールドは、C# でサポートされる数種類の変数(5 を参照)の中の 2 つであり、それぞれ "静的変数" および "インスタンス変数" と表現されることがあります。

静的フィールドは特定のインスタンスの一部ではなく、クローズ型(4.4.2 を参照)のすべてのインスタンス間で共有されます。クローズクラス型のインスタンスがいくつ作成される場合でも、関連付けられたアプリケーション ドメインに対する静的フィールドのコピーは 1 つだけです。

次に例を示します。

```
class C<V>
{
    static int count = 0;
```

```

public C() {
    count++;
}
public static int Count {
    get { return count; }
}
}

class Application
{
    static void Main() {
        C<int> x1 = new C<int>();
        Console.WriteLine(C<int>.Count);      // Prints 1
        C<double> x2 = new C<double>();
        Console.WriteLine(C<int>.Count);      // Prints 1
        C<int> x3 = new C<int>();
        Console.WriteLine(C<int>.Count);      // Prints 2
    }
}

```

インスタンス フィールドはインスタンスに属します。具体的には、クラスのどのインスタンスにも、そのクラスのすべてのインスタンス フィールドのセットが格納されます。

フィールドが **E.M** 形式の *member-access* (7.6.4 を参照) で参照される場合、M が静的フィールドであれば E は M を含む型を表す必要があり、M がインスタンス フィールドであれば、E は M を含む型のインスタンスを表す必要があります。

静的メンバーとインスタンス メンバーの違いについては、10.3.7 を参照してください。

### 10.5.2 Readonly フィールド

*field-declaration* に **readonly** 修飾子が含まれる場合、宣言によって導入されるフィールドは "**readonly** フィールド" です。**readonly** フィールドへの直接代入は、その宣言の一部としてだけ、または同じクラスのインスタンス コンストラクターか静的コンストラクターの中でだけ行うことができます。**readonly** フィールドは、これらのコンテキストで何度も代入できます。特に、**readonly** フィールドへの直接代入を行うことができるのは、次のコンテキストだけです。

- フィールドを導入する *variable-declarator* 内。宣言に *variable-initializer* を含めます。
- インスタンス フィールドの場合は、フィールド宣言を含んでいるクラスのインスタンス コンストラクター内。静的フィールドの場合は、フィールド宣言を含んでいるクラスの静的コンストラクター内。また、これらのコンテキスト内でだけ、**readonly** フィールドを **out** パラメーターまたは **ref** パラメーターとして渡すことができます。

他のコンテキストで **readonly** フィールドに代入しようとしたり、**readonly** フィールドを **out** パラメーターや **ref** パラメーターとして渡そうとしたりすると、コンパイル エラーになります。

#### 10.5.2.1 定数への静的 readonly フィールドの使用

定数値のシンボル名が必要で、その値の型を **const** 宣言で使用できない場合、またはその値をコンパイル時に計算できない場合は、**static readonly** フィールドが役に立ちます。次に例を示します。

```

public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);

    private byte red, green, blue;

    public Color(byte r, byte g, byte b) {
        red = r;
        green = g;
        blue = b;
    }
}

```

`Black`、`white`、`Red`、`Green`、および`Blue`の各メンバーを `const` メンバーとして宣言することはできません。これらの値はコンパイル時に計算できないためです。ただし、代わりに `static readonly` として宣言すると、ほぼ同じ効果が得られます。

### 10.5.2.2 定数と静的 readonly フィールドのバージョン管理

定数と `readonly` フィールドでは、バイナリ バージョン管理の意味が異なります。式が定数を参照する場合、定数の値はコンパイル時に取得されますが、式が `readonly` フィールドを参照する場合、フィールドの値は実行時まで取得されません。2つの別個のプログラムで構成されるアプリケーションを考えてみます。

```

using System;
namespace Program1
{
    public class Utils
    {
        public static readonly int X = 1;
    }
}

namespace Program2
{
    class Test
    {
        static void Main() {
            Console.WriteLine(Program1.Utils.X);
        }
    }
}

```

`Program1` 名前空間と `Program2` 名前空間は、それぞれ別個にコンパイルされる2つのプログラムを表します。`Program1.Utils.X` は静的 `readonly` フィールドとして宣言されているため、`Console.WriteLine` ステートメントで出力される値は、コンパイル時にはわかりませんが、実行時に取得されます。したがって、`X` の値を変更して `Program1` を再コンパイルした場合、`Program2` を再コンパイルしなくとも、`Console.WriteLine` ステートメントは新しい値を出力します。ただし、`X` が定数の場合、`X` の値は `Program2` をコンパイルしたときに取得されています。この場合、`Program1` で行った変更は、`Program2` を再コンパイルするまで反映されません。

### 10.5.3 Volatile フィールド

*field-declaration* に `volatile` 修飾子が含まれる場合、宣言によって導入されるフィールドは "*volatile フィールド*" です。

非 volatile フィールドの場合は、命令を並べ替える最適化手法を使うと、*lock-statement* (8.12 を参照) によって提供されるような同期化を行わずにフィールドにアクセスするマルチスレッドプログラムで、予期しない結果が生じることがあります。これらの最適化を実行できるのは、コンパイラ、実行時システム、またはハードウェアです。volatile フィールドの場合、並べ替えによる最適化は次のように制限されます。

- volatile フィールドの読み取りは、"**volatile 読み取り**" と呼ばれます。volatile 読み取りは "取得形式" であり、命令シーケンスでその後に行われるどのメモリ参照よりも前に行われます。
- volatile フィールドへの書き込みは、"**volatile 書き込み**" と呼ばれます。volatile 書き込みは "解放形式" であり、命令シーケンスの書き込み命令の前に行われるどのメモリ参照よりも後に行われます。

これらの制限により、他のスレッドによる volatile 書き込みは、それらが実行された順にすべてのスレッドで行われます。準拠した実装では、すべての実行スレッドから参照できる volatile 書き込みの 1 つの完全な順序付けを用意する必要はありません。volatile フィールドの型は、次のいずれかの値である必要があります。

- *reference-type*
- byte、sbyte、short、ushort、int、uint、char、float、bool、System.IntPtr、または System.UIntPtr の各型。
- byte、sbyte、short、ushort、int、または uint の列挙基本型を持つ *enum-type*。

次の例を参照してください。

```
using System;
using System.Threading;

class Test
{
    public static int result;
    public static volatile bool finished;

    static void Thread2() {
        result = 143;
        finished = true;
    }

    static void Main() {
        finished = false;

        // Run Thread2() in a new thread
        new Thread(new ThreadStart(Thread2)).Start();

        // Wait for Thread2 to signal that it has a result by setting
        // finished to true.
        for (;;) {
            if (finished) {
                Console.WriteLine("result = {0}", result);
                return;
            }
        }
    }
}
```

この例では、次のように出力されます。

```
result = 143
```

この例では、`Main` メソッドが `Thread2` メソッドを実行する新しいスレッドを開始します。このメソッドは、値を `result` という非 `volatile` フィールドに格納し、次に `volatile` フィールド `finished` に `true` を格納します。メインスレッドは、フィールド `finished` が `true` に設定された時点で、フィールド `result` を読み込みます。`finished` は `volatile` として宣言されているため、メインスレッドはフィールド `result` から値 143 を読み込む必要があります。フィールド `finished` が `volatile` として宣言されていない場合は、`finished` への格納後に `result` への格納がメインスレッドから参照可能になります。その結果、メインスレッドがフィールド `result` から読み込む値は 0 になります。`finished` を `volatile` フィールドとして宣言すると、このような矛盾を回避できます。

#### 10.5.4 フィールドの初期化

静的フィールドかインスタンスフィールドかにかかわらず、フィールドの初期値は、そのフィールドの型の既定値(5.2を参照)です。この既定の初期化を行う前にフィールドの値を参照することはできません。このため、フィールドが"未初期化"状態になることはありません。次の例を参照してください。

```
using System;
class Test
{
    static bool b;
    int i;
    static void Main() {
        Test t = new Test();
        Console.WriteLine("b = {0}, i = {1}", b, t.i);
    }
}
```

この例では、次のように出力されます。

```
b = False, i = 0
```

`b` と `i` はどちらも自動的に既定値に初期化されるためです。

#### 10.5.5 変数初期化子

フィールド宣言には、*variable-initializer* が含まれる場合があります。静的フィールドの場合、変数初期化子は、クラスの初期化中に実行される代入ステートメントに対応します。インスタンスフィールドの場合、変数初期化子は、クラスのインスタンス作成時に実行される代入ステートメントに対応します。

次の例を参照してください。

```
using System;
class Test
{
    static double x = Math.Sqrt(2.0);
    int i = 100;
    string s = "Hello";
    static void Main() {
        Test a = new Test();
        Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
    }
}
```

この例では、次のように出力されます。

```
x = 1.4142135623731, i = 100, s = Hello
```

これは、`x`への代入が静的フィールド初期化子の実行時に行われ、`i`および`s`への代入が、インスタンス フィールド初期化子の実行時に行われるためです。

10.5.4 で説明している既定値の初期化は、変数初期化子のあるフィールドを含むすべてのフィールドで実行されます。したがって、クラスが初期化されると、そのクラスのすべての静的フィールドは最初に既定値に初期化され、その後、静的フィールド初期化子が記述順に実行されます。同様に、クラスのインスタンスが作成されると、そのインスタンスのすべてのインスタンス フィールドは最初に既定値に初期化され、その後、インスタンス フィールド初期化子が記述順に実行されます。

変数初期化子を持つ静的フィールドは、その既定値状態を参照できます。ただし、スタイルの問題として、これは望ましくありません。次の例を参照してください。

```
using System;
class Test
{
    static int a = b + 1;
    static int b = a + 1;
    static void Main()
    {
        Console.WriteLine("a = {0}, b = {1}", a, b);
    }
}
```

これで、この動作が示されます。`a` と `b` が循環的に定義されているにもかかわらず、プログラムは有効です。出力は次のようにになります。

```
a = 1, b = 2
```

初期化子が実行される前に、静的フィールドの `a` と `b` は `0` (`int` の既定値) に初期化されます。`a` の初期化子が実行されると、`b` の値は `0` になり、`a` は `1` に初期化されます。`b` の初期化子が実行されると、`a` の値は既に `1` なので、`b` は `2` に初期化されます。

### 10.5.5.1 静的フィールドの初期化

クラスの静的フィールドの変数初期化子は、クラス宣言に記述された順序どおりに実行される一連の代入に相当します。クラスに静的コンストラクター (10.12 を参照) が存在する場合は、その静的コンストラクターを実行する \r \h 直前に静的フィールド初期化子を実行します。それ以外の場合は、それぞれの実装に応じて、そのクラスの静的フィールドが初めて使用される前に静的フィールド初期化子を実行します。次の例を参照してください。

```
using System;
class Test
{
    static void Main()
    {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }
    public static int F(string s)
    {
        Console.WriteLine(s);
        return 1;
    }
}
class A
{
    public static int X = Test.F("Init A");
```

```
class B
{
    public static int Y = Test.F("Init B");
```

これは、次のように出力されます。

```
Init A
Init B
1 1
```

または、次のように出力されます。

```
Init B
Init A
1 1
```

これは、**X** の初期化子と **Y** の初期化子の実行順序が任意であるためです。初期化子が実行されるのは、これらのフィールドが参照される前に限定されます。これとは別の例を示します。

```
using System;
class Test
{
    static void Main() {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }
    public static int F(string s) {
        Console.WriteLine(s);
        return 1;
    }
}
class A
{
    static A() {}
    public static int X = Test.F("Init A");
}
class B
{
    static B() {}
    public static int Y = Test.F("Init B");
```

出力は次のようになります。

```
Init B
Init A
1 1
```

これは、静的コンストラクターを実行する場合の規則 (10.12 を参照) により、最初に **B** の静的コンストラクター (および **B** の静的フィールド初期化子) を実行してから、**A** の静的コンストラクターとフィールド初期化子を実行する必要があるためです。

### 10.5.2 インスタンス フィールドの初期化

クラスのインスタンス フィールド変数初期化子は、クラスのインスタンス コンストラクター (10.11.1 を参照) のいずれかへのエントリ時に、直ちに実行される一連の代入に対応します。変数初期化子は、クラス宣言に出てくる順番どおりに実行されます。クラスのインスタンス作成および初期化プロセスの詳細については、10.11 を参照してください。

インスタンス フィールドの変数初期化子は、作成中のインスタンスを参照できません。したがって、変数初期化子で `this` を参照するとコンパイル エラーになります。同様に、変数初期化子で `simple-name` を使用してインスタンス メンバーを参照するとコンパイル エラーになります。次に例を示します。

```
class A
{
    int x = 1;
    int y = x + 1;    // Error, reference to instance member of this
}
```

`y` の変数初期化子は作成中のインスタンスのメンバーを参照しているため、コンパイル エラーが発生します。

## 10.6 メソッド

"メソッド" は、オブジェクトまたはクラスによって実行される演算またはアクションを実装するメンバーです。メソッドは *method-declaration* を使用して宣言されます。

```
method-declaration:
    method-header method-body

method-header:
    attributesopt method-modifiersopt partialopt return-type member-name type-parameter-listopt
        ( formal-parameter-listopt ) type-parameter-constraints-clausesopt

method-modifiers:
    method-modifier
    method-modifiers method-modifier

method-modifier:
    new
    public
    protected
    internal
    private
    static
    virtual
    sealed
    override
    abstract
    extern
    async

return-type:
    type
    void

member-name:
    identifier
    interface-type . identifier

method-body:
    block
    ;
```

*method-declaration* には、属性(17 を参照)のセット、4つのアクセス修飾子の有効な組み合わせ(10.3.5 を参照)、**new** 修飾子(10.3.4 を参照)、**static** 修飾子(10.6.2 を参照)、**virtual** 修飾子(10.6.3 を参照)、**override** 修飾子(10.6.4 を参照)、**sealed** 修飾子(10.6.5 を参照)、**abstract** 修飾子(10.6.6 を参照)、および**extern** 修飾子(10.6.7 を参照)を指定できます。

次の条件がすべて満たされる場合、宣言に含まれる修飾子の組み合わせは有効であると見なされます。

- アクセス修飾子の有効な組み合わせ(10.3.5 を参照)が含まれる。
- 同じ修飾子が複数回含まれない。
- **static** 修飾子、**virtual** 修飾子、および**override** 修飾子のうち、最大でもいずれか1つしか含まれない。
- **new** 修飾子と**override** 修飾子のうち、最大でもどちらか1つしか含まれない。
- **abstract** 修飾子が含まれる場合は、**static**、**virtual**、**sealed**、または**extern** の各修飾子が含まれない。
- **private** 修飾子が含まれる場合は、**virtual**、**override**、または**abstract** の各修飾子が含まれない。
- **sealed** 修飾子が含まれる場合は、**override** 修飾子も含まれる。
- **partial** 修飾子が含まれる場合は、**new**、**public**、**protected**、**internal**、**private**、**virtual**、**sealed**、**override**、**abstract**、または**extern** の各修飾子が含まれない。

**async** 修飾子のあるメソッドは非同期関数で、エラー! 参照元が見つかりません。で説明する規則に従います。

メソッド宣言の *return-type* は、メソッドで計算して返される値の型を指定します。メソッドが値を返さない場合、*return-type* は **void** です。**partial** 修飾子が含まれる場合、戻り値の型が **void** である必要があります。

*member-name* はメソッドの名前を指定します。メソッドが明示的なインターフェイス メンバー実装(13.4.1 を参照)でない場合、*member-name* は単に *identifier* です。明示的なインターフェイス メンバー実装の場合、*member-name* は順に、*interface-type*、"."、および *identifier* で構成されます。

省略可能な *type-parameter-list* は、メソッド(10.1.3 を参照)の型パラメーターを指定します。*type-parameter-list* を指定する場合、メソッドは "ジェネリック メソッド" です。メソッドに **extern** 修飾子を含める場合、*type-parameter-list* を指定できません。

省略可能な *formal-parameter-list* は、メソッド(10.6.1 を参照)のパラメーターを指定します。

省略可能な *type-parameter-constraints-clauses* は、個々の型パラメーター(10.1.5 を参照)の制約を指定します。*type-parameter-list* も指定し、メソッドに **override** 修飾子がない場合にのみ指定できます。

*return-type* と、メソッドの *formal-parameter-list* で参照される各型は、少なくともメソッド自体と同程度にアクセス可能である必要があります(3.5.4 を参照)。

**abstract** メソッドおよび **extern** メソッドの場合、*method-body* はセミコロンのみで構成されます。**partial** メソッドの場合、*method-body* はセミコロンまたはブロックのどちらかで構成されます。その他すべてのメソッドの場合、*method-body* を構成するのは、メソッド呼び出し時に実行するステートメントを指定する *block* です。

*method-body* がセミコロンで構成される場合、宣言に **async** 修飾子を含めることはできません。

メソッドの名前、型パラメーターリスト、および仮パラメーターリストは、メソッドのシグネチャ(3.6 を参照)を定義します。特に、メソッドのシグネチャは、その名前、型パラメーターの数と番号、修飾子、および仮パラメーターの型で構成されます。このため、仮パラメーターの型で使用されるメソッドの型パラメーターは、名前で識別されるのではなく、メソッドの型引数リストで何番目にあるかにより識別されます。戻り値の型、型パラメーターまたは仮パラメーターの名前は、メソッドのシグネチャに含まれません。

メソッドの名前は、同じクラスで宣言されている他のすべての非メソッドと異なる名前である必要があります。さらに、メソッドのシグネチャは、同じクラスで宣言されている他のすべてのメソッドのシグネチャと異なっている必要があります。同じクラスで宣言される 2 つのメソッドに、**ref** と **out** の違いしかないシグネチャを指定することはできません。

メソッドの *type-parameter* は、*method-declaration* 全体にわたってスコープ内にあり、*return-type*、*method-body*、および *type-parameter-constraints-clauses* において、そのスコープ全体で使用する型の形成に使用できますが、*attributes* では使用できません。

すべての仮パラメーターと型パラメーターには異なる名前を付ける必要があります。

### 10.6.1 メソッドのパラメーター

メソッドのパラメーター(存在する場合)は、メソッドの *formal-parameter-list* によって宣言されます。

```

formal-parameter-list:
    fixed-parameters
    fixed-parameters , parameter-array
    parameter-array

fixed-parameters:
    fixed-parameter
    fixed-parameters , fixed-parameter

fixed-parameter:
    attributesopt parameter-modifieropt type identifier default-argumentopt

default-argument:
    = expression

parameter-modifier:
    ref
    out
    this

parameter-array:
    attributesopt params array-type identifier

```

仮パラメーター リストは、1 つ以上のコンマ区切りのパラメーターで構成され、その最後だけを *parameter-array* にすることができます。

*fixed-parameter* は、省略可能な属性のセット(17 を参照)、省略可能な **ref**、**out**、または **this** 修飾子、型、識別子、省略可能な *default-argument* から構成されています。それぞれの *fixed-parameter* は、指定された名前で指定された型のパラメーターを宣言します。**this** 修飾子は、メソッドを拡張メソッドとして指定し、静的メソッドの最初のパラメーターにのみ認められます。拡張メソッドについては、10.6.9 で詳細に説明します。

*default-argument*を持つ*fixed-parameter*は"省略可能なパラメーター"と呼ばれ、*default-argument*を持たない*fixed-parameter*は"必須パラメーター"と呼ばれます。必須パラメーターは、*formal-parameter-list*内で省略可能なパラメーターの後には指定できません。

*ref* パラメーターまたは*out* パラメーターには*default-argument*を指定できません。*default-argument*内の*expression*は、次のいずれかである必要があります。

- *constant-expression*
- *new s()* の形式の式(ただし、*s* は値型)
- *default(s)* の形式の式(ただし、*s* は値型)

*expression*は、恒等変換または*null* 許容変換によってパラメーターの型に暗黙的に変換できる必要があります。

部分メソッド実装宣言(10.2.7を参照)、インターフェイスメンバーの明示的実装(13.4.1を参照)、または単一パラメーターインデクサーの宣言(10.9を参照)に省略可能なパラメーターが出現する場合は、コンパイラが警告を生成します。これらのメンバーは、引数の省略を許容する形で呼び出すことはできません。

*parameter-array*は、省略可能な属性のセット(17を参照)、*params*修飾子、*array-type*、および識別子から構成されています。パラメーター配列は、指定された名前で指定された配列型の単一パラメーターを宣言します。パラメーター配列の*array-type*は、1次配列型(12.1を参照)である必要があります。メソッド呼び出しでは、パラメーター配列は指定した配列型の引数を1つ指定するか、配列要素型の引数を0以上指定できます。パラメーター配列については10.6.1.4で詳細に説明します。

*parameter-array*は、省略可能なパラメーターの後に指定できますが、既定値を持つことはできません。*parameter-array*に対応する引数を省略すると、空の配列が作成されます。

さまざまな種類のパラメーターの例を次に示します。

```
public void M(
    ref int      i,
    decimal     d,
    bool        b = false,
    bool?       n = false,
    string      s = "Hello",
    object      o = null,
    T           t = default(T),
    params int[] a
) {}
```

Mの*formal-parameter-list*において、*i*は必須の*ref* パラメーター、*d*は必須の値パラメーター、*b*、*s*、および*t*は省略可能な値パラメーター、*a*はパラメーター配列です。

メソッドの宣言では、パラメーター、型パラメーター、およびローカル変数用に別個の宣言空間が作成されます。メソッドの型パラメーターリストと仮パラメーターリスト、およびメソッドの*block*内のローカル変数宣言によって、この宣言空間に名前が導入されます。メソッド宣言空間に同じ名前の2つのメンバーがあるとエラーになります。メソッド宣言空間と入れ子になった宣言空間のローカル変数宣言空間に同じ名前の要素が格納されるとエラーになります。

メソッド呼び出し(7.6.5.1を参照)によって、その呼び出しに固有なメソッドの仮パラメーターとローカル変数のコピーが作成されます。また、呼び出しの引数リストにより、値または変数参照が新規作成された仮パラメーターに代入されます。メソッドの*block*内では、*simple-name*式(7.6.2を参照)の識別子で仮パラメーターを参照できます。

仮パラメーターには次の4種類があります。

- 値パラメーター。修飾子を使わずに宣言されます。
- 参照パラメーター。`ref`修飾子を使って宣言されます。
- 出力パラメーター。`out`修飾子を使って宣言されます。
- パラメーター配列。`params`修飾子を使って宣言されます。

3.6で説明しているように、`ref`修飾子および`out`修飾子はメソッドのシグネチャの一部ですが、`params`修飾子はシグネチャの一部ではありません。

#### 10.6.1.1 値パラメーター

修飾子なしで宣言されたパラメーターは値パラメーターです。値パラメーターは、メソッドの呼び出しで与えられた対応する引数から初期値を取得するローカル変数に対応します。

仮パラメーターが値パラメーターの場合、メソッド呼び出しで対応する引数は、仮パラメーターの型に暗黙的に変換可能な(0を参照)式である必要があります。

メソッドを使うと、値パラメーターに新たな値を代入できます。値パラメーターへの代入によって影響を受けるのは、値パラメーターによって表されるローカルの格納場所だけです。メソッドの呼び出しで与えられる実際の引数には影響しません。

#### 10.6.1.2 参照パラメーター

`ref`修飾子を使って宣言されたパラメーターは参照パラメーターです。参照パラメーターは値パラメーターとは異なり、新規の格納場所を作成しません。代わりに、参照パラメーターはメソッドの呼び出しの引数として与えられる変数と同じ格納場所を表します。

仮パラメーターが参照パラメーターの場合、メソッドの呼び出しで対応する引数を構成するのは、順に、`ref`キーワードと、仮パラメーターと同じ型の*variable-reference*(5.3.3を参照)です。変数は、参照パラメーターとして渡すことができるようになる前に、明示的に代入する必要があります。

参照パラメーターは、メソッドの中で常に明示的に代入されると見なされます。

反復子(10.14を参照)として宣言されたメソッドには、参照パラメーターを含めることができません。

次の例を参照してください。

```
using System;
class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("i = {0}, j = {1}", i, j);
    }
}
```

この例では、次のように出力されます。

i = 2, j = 1

`Swap` を `Main` で呼び出す場合、`x` が `i` を表し、`y` が `j` を表します。したがって、呼び出しによって `i` と `j` の値が交換されます。

参照パラメーターをとるメソッドでは、複数の名前で同じ格納場所を表すことができます。次に例を示します。

```
class A
{
    string s;
    void F(ref string a, ref string b) {
        s = "One";
        a = "Two";
        b = "Three";
    }
    void G() {
        F(ref s, ref s);
    }
}
```

`G` における `F` の呼び出しが、`a` と `b` の両方に `s` への参照を渡しています。したがって、この呼び出しでは、`s`、`a`、および `b` はすべて同じ格納場所を参照し、3つの代入はすべてインスタンス フィールド `s` を変更します。

#### 10.6.1.3 出力パラメーター

`out` 修飾子を使って宣言されたパラメーターは出力パラメーターです。出力パラメーターは参照パラメーターと同様に、新規の格納場所を作成しません。代わりに、出力パラメーターはメソッドの呼び出しの引数として与えられる変数と同じ格納場所を表します。

仮パラメーターが出力パラメーターの場合、メソッドの呼び出しで対応する引数を構成するのは、順に、`out` キーワードと、仮パラメーターと同じ型の *variable-reference* (5.3.3 を参照) です。変数は出力パラメーターとして渡すことができるようになる前に明示的に代入する必要はありませんが、変数が出力パラメーターとして渡された呼び出しの後で、その変数は明示的に代入されたと見なされます。

メソッド内では、出力パラメーターは初期状態でローカル変数と同様に未代入と見なされ、値が使用される前に明示的に代入される必要があります。

ただし、メソッドの各出力パラメーターは、メソッドが戻る以前に明示的に代入されている必要があります。

部分メソッド (10.2.7 を参照) または反復子 (10.14 を参照) として宣言されたメソッドには、出力パラメーターを含めることができません。

出力パラメーターは通常、複数の戻り値を生成するメソッドで使用されます。次に例を示します。

```
using System;
```

```

class Test
{
    static void SplitPath(string path, out string dir, out string name) {
        int i = path.Length;
        while (i > 0) {
            char ch = path[i - 1];
            if (ch == '\\' || ch == '/' || ch == ':') break;
            i--;
        }
        dir = path.Substring(0, i);
        name = path.Substring(i);
    }

    static void Main() {
        string dir, name;
        SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
        Console.WriteLine(dir);
        Console.WriteLine(name);
    }
}

```

この例では、次のように出力されます。

```
c:\Windows\System\
hello.txt
```

`dir` 変数と `name` 変数は `SplitPath` に渡される前は未代入にできますが、呼び出しの後で明示的に代入されると見なされます。

#### 10.6.1.4 パラメーター配列

`params` 修飾子を使って宣言されたパラメーターはパラメーター配列です。仮パラメーター リストにパラメーター配列が含まれる場合、パラメーター配列はリストの最後のパラメーターであり、1 次元配列型である必要があります。たとえば、型 `string[]` および型 `string[][]` をパラメーター配列の型として使用できますが、型 `string[,]` は使用できません。`params` 修飾子に `ref` 修飾子や `out` 修飾子を組み合わせることはできません。

パラメーター配列を使うと、メソッドの呼び出しで次の 2 つの方法のいずれかで引数を指定できます。

- パラメーター配列に指定される引数には、パラメーター配列型に暗黙的に変換可能な (0 を参照) 1 つの式を使用できます。この場合、パラメーター配列は値パラメーターとまったく同じように機能します。
- また、呼び出しはパラメーター配列に 0 以上の引数を指定できます。各引数は、パラメーター配列の要素型に暗黙的に変換可能な (0 を参照) 式です。この場合、呼び出しによって、引数の数に対応した長さを持つパラメーター配列型のインスタンスが作成され、配列インスタンスの要素が指定の引数值で初期化され、新規作成された配列インスタンスが実際の引数として使用されます。

呼び出しでの可変数の引数を使用できることを除けば、パラメーター配列は、同じ型の値パラメーター (10.6.1.1 を参照) と同等です。

次の例を参照してください。

```
using System;
```

```

class Test
{
    static void F(params int[] args) {
        Console.WriteLine("Array contains {0} elements:", args.Length);
        foreach (int i in args)
            Console.WriteLine("{0}", i);
        Console.WriteLine();
    }

    static void Main() {
        int[] arr = {1, 2, 3};
        F(arr);
        F(10, 20, 30, 40);
        F();
    }
}

```

この例では、次のように出力されます。

```

Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:

```

`F` の最初の呼び出しでは、単純に配列 `a` を値パラメーターとして渡しています。`F` の 2 回目の呼び出しでは、指定の要素数を持つ 4 つの要素の `int[]` が自動的に作成され、配列インスタンスが値パラメーターとして渡されます。同様に、`F` の 3 回目の呼び出しでは、要素が 0 の `int[]` が作成され、そのインスタンスが値パラメーターとして渡されます。2 番目と 3 番目の呼び出しは、次のように記述するのとまったく同じです。

```

F(new int[] {10, 20, 30, 40});
F(new int[] {});

```

オーバーロードの解決法を実行する場合、パラメーター配列を持つメソッドは通常形式と展開形式 (7.5.3.1 を参照) のどちらにも適用できます。メソッドの展開形式を使用できるのは、メソッドの通常形式を適用できない場合と、展開形式と同じシグネチャを持つ適用可能なメソッドが同じ型で宣言されていない場合だけです。

次の例を参照してください。

```

using System;
class Test
{
    static void F(params object[] a) {
        Console.WriteLine("F(object[])");
    }

    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object a0, object a1) {
        Console.WriteLine("F(object,object)");
    }

    static void Main() {
        F();
        F(1);
        F(1, 2);
        F(1, 2, 3);
        F(1, 2, 3, 4);
    }
}

```

この例では、次のように出力されます。

```
F();
F(object[]);
F(object,object);
F(object[]);
F(object[]);
```

この例では、パラメーター配列を持つメソッドの2つの展開形式が、通常のメソッドとして既にクラスに含まれています。このため、これらの展開形式はオーバーロードの解決法の実行時には考慮されず、1回目と3回目のメソッドの呼び出しでは通常のメソッドが選択されます。クラスでパラメーター配列を持つメソッドを呼び出す場合は、展開形式を通常形式として含めることはほとんどありません。そうすることで、パラメーター配列を持つメソッドの展開形式が呼び出されるときに実行される、配列インスタンスの配置を回避できます。

パラメーター配列の型が `object[]` である場合は、メソッドの通常形式と、単一の `object` パラメーターに展開された形式があいまいになる可能性があります。このあいまいさの原因は、`object[]` 自体が暗黙的に型 `object` に変換できることにあります。ただし、必要に応じてキャストを挿入することで解決できるので、あいまいさがあっても問題はありません。

次の例を参照してください。

```
using System;
class Test
{
    static void F(params object[] args) {
        foreach (object o in args) {
            Console.WriteLine(o.GetType().FullName);
            Console.WriteLine(" ");
        }
        Console.WriteLine();
    }

    static void Main() {
        object[] a = {1, "Hello", 123.456};
        object o = a;
        F(a);
        F((object)a);
        F(o);
        F((object[])o);
    }
}
```

この例では、次のように出力されます。

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

`F` の最初と最後の呼び出しでは、引数の型からパラメーターの型(両方とも型 `object[]`)への暗黙の型変換が存在するので、`F` の通常形式を適用できます。このため、オーバーロードの解決法では `F` の通常形式が選択され、引数は通常の値パラメーターとして渡されます。2回目と3回目の呼び出しでは、引数の型からパラメーターの型への暗黙の型変換が存在しないため、`F` の通常形式は適用可能ではありません(`object` 型は `object[]` 型に暗黙的に変換できません)。しかし、`F` の展開形式は適用可能であるため、オーバーロード解決で展開形式が選択されます。結果として、呼び出しによって1つの要素 `object[]` が作成され、その配列の1つの要素は指定の引数値で初期化されます。引数値自体が `object[]` への参照です。

## 10.6.2 静的メソッドとインスタンス メソッド

メソッドの宣言に `static` 修飾子が含まれる場合、そのメソッドは静的メソッドと呼ばれます。

`static` 修飾子が存在しない場合、そのメソッドはインスタンス メソッドと呼ばれます。

静的メソッドは特定のインスタンスでは機能しないので、静的メソッドで `this` を参照するとコンパイルエラーになります。

インスタンス メソッドはクラスの指定のインスタンスで機能し、そのインスタンスは `this` (7.6.7 を参照) としてアクセスできます。

メソッドが `E.M` 形式の *member-access* (7.6.4 を参照) で参照される場合、`M` が静的メソッドであれば `E` は `M` を含む型を表す必要があり、`M` がインスタンス メソッドであれば、`E` は `M` を含む型のインスタンスを表す必要があります。

静的メンバーとインスタンス メンバーの違いについては、10.3.7 を参照してください。

## 10.6.3 仮想メソッド

インスタンス メソッドの宣言に `virtual` 修飾子が含まれる場合、そのメソッドは "仮想メソッド" と呼ばれます。`virtual` 修飾子が存在しない場合、そのメソッドは非仮想メソッドと呼ばれます。

非仮想メソッドの実装は 1 種類だけです。メソッドの呼び出し元が、宣言元のクラスのインスタンスであるか、派生クラスのインスタンスであるかに関係なく、実装は同じです。対照的に、仮想メソッドの実装は、派生クラスに取り替えることができます。継承仮想メソッドの実装に取り替える処理は、メソッドの "オーバーライド" (10.6.4 を参照) と呼ばれます。

仮想メソッドの呼び出しでは、呼び出しが行われるインスタンスの "実行時の型" により、実際に呼び出されるメソッド実装が決定されます。非仮想メソッドの呼び出しでは、インスタンスの "コンパイル時の型" が決定要因です。正確には、`N` という名前のメソッドが、コンパイル時の型 `C` および実行時の型 `R` を持つインスタンスで、引数リスト `A` によって呼び出されると (`R` は、`C` または `C` から派生したクラス)、呼び出しあは次のように処理されます。

- 最初に、オーバーロードの解決法は `C`、`N`、および `A` に適用され、`C` で宣言および継承される一連のメソッドからメソッド `M` が選択されます。これは 7.6.5.1 で説明します。
- 次に、`M` が非仮想メソッドである場合は `M` が呼び出されます。
- それ以外の場合、`M` は仮想メソッドであり、`R` に応じて `M` の最派生実装が呼び出されます。

クラスによって宣言または継承された仮想メソッドのそれぞれについて、そのクラスに関するメソッドの "最派生実装" が存在します。クラス `R` に関する仮想メソッド `M` の最派生実装は、次のように決定されます。

- `R` に `M` の `virtual` 宣言が含まれる場合は、これが `M` の最派生実装です。
- それ以外の場合で、`R` に `M` の `override` が含まれるときは、これが `M` の最派生実装です。
- それ以外の場合、`R` に関連する `M` の最派生実装は、`R` の直接基底クラスに関連する `M` の最派生実装と同じです。

仮想メソッドと非仮想メソッドの違いを次の例で示します。

```
using System;
```

```

class A
{
    public void F() { Console.WriteLine("A.F"); }
    public virtual void G() { Console.WriteLine("A.G"); }
}

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

class Test
{
    static void Main()
    {
        B b = new B();
        A a = b;
        a.F();
        b.F();
        a.G();
        b.G();
    }
}

```

この例では、**A** は非仮想メソッド **F** および仮想メソッド **G** を導入します。クラス **B** には、"新しい" 非仮想メソッドの **F** があります。したがって、継承した **F** は "隠ぺい" され、継承メソッドの **G** も "オーバーライド" されます。この例では、次のように出力されます。

```

A.F
B.F
B.G
B.G

```

**a.G()** ステートメントでは、**A.G** ではなく **B.G** が呼び出されます。実際に呼び出すメソッド実装を決定するのは、インスタンスの実行時の型 (**B**) であり、インスタンスのコンパイル時型 (**A**) ではないためです。

メソッドを使って、継承されたメソッドを隠ぺいできるので、同じシグネチャを持つ複数の仮想メソッドを 1 つのクラスに含めることができます。最派生メソッド以外のすべてが隠ぺいされるので、あいまい性の問題は発生しません。次に例を示します。

```

using System;

class A
{
    public virtual void F() { Console.WriteLine("A.F"); }
}

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

class C: B
{
    new public virtual void F() { Console.WriteLine("C.F"); }
}

class D: C
{
    public override void F() { Console.WriteLine("D.F"); }
}

```

```

class Test
{
    static void Main() {
        D d = new D();
        A a = d;
        B b = d;
        C c = d;
        a.F();
        b.F();
        c.F();
        d.F();
    }
}

```

**C** クラスと **D** クラスは、同じシグネチャを持つ 2 つの仮想メソッドを含みます。1 つは **A** により導入されたメソッド、もう 1 つは **C** により導入されたメソッドです。**C** により導入されたメソッドは、**A** から継承されたメソッドを隠します。したがって、**D** でのオーバーライド宣言は、**C** により導入されたメソッドをオーバーライドします。**D** が **A** により導入されたメソッドをオーバーライドすることはできません。この例では、次のように出力されます。

```

B.F
B.F
D.F
D.F

```

隠されているメソッドの下位の派生型を通じて **D** のインスタンスにアクセスすることで、隠されている仮想メソッドを呼び出すことができます。

#### 10.6.4 オーバーライド メソッド

インスタンス メソッドの宣言に **override** 修飾子が含まれる場合、そのメソッドは "オーバーライド メソッド" と呼ばれます。オーバーライド メソッドは、同じシグネチャを持つ継承仮想メソッドをオーバーライドします。仮想メソッドの宣言は新しいメソッドを "導入" しますが、オーバーライド メソッドの宣言は、そのメソッドの新しい実装を提供して、既存の継承仮想メソッドを "特化" します。

**override** 宣言によってオーバーライドされるメソッドは、"オーバーライドされる基本メソッド" と呼ばれます。クラス **C** で宣言されたオーバーライド メソッド **M** の場合、オーバーライドされる基本 メソッドは、**C** の各基底クラス型を調べることで決定されます。この検索は、特定基底クラス型で、型引数の代入後に **M** と同じシグネチャを持つ最低 1 つのアクセス可能なメソッドが見つかるまで、**C** の直接基底クラス型から順次、連続する各直接基底クラス型に対して実行されます。オーバーライドされる基本メソッドを見つけるために、**public**、**protected**、**protected internal**、または **internal** であり、**C** と同じプログラムで宣言されているメソッドは、アクセス可能と見なされます。

オーバーライドされる宣言に対して次の項目のすべてが **True** でない場合は、コンパイル エラーが発生します。

- オーバーライドされる基本メソッドを上で説明した方法で見つけることができる。
- このようにオーバーライドされる基本メソッドは 1 つのみです。この制約は、基底クラス型が構築された型で、型引数の代入により 2 つのメソッドのシグネチャが同じになる場合にのみ効果があります。
- オーバーライドされる基本メソッドは、仮想メソッド、抽象メソッド、またはオーバーライド メソッドです。つまり、オーバーライドされる基本メソッドは静的メソッドや非仮想メソッドではありません。

- オーバーライドされる基本メソッドがシールメソッドではない。
- オーバーライドメソッドとオーバーライドされる基本メソッドの戻り値の型が同じ。
- オーバーライド宣言とオーバーライドされる基本メソッドの宣言されたアクセシビリティが同じ。つまり、オーバーライド宣言で仮想メソッドのアクセシビリティを変更できない。ただし、オーバーライドされる基本メソッドが内部的に保護され、そのメソッドがオーバーライドメソッドが格納されるアセンブリ以外のアセンブリで宣言されている場合、オーバーライドメソッドの宣言されたアクセシビリティは保護する必要があります。
- オーバーライド宣言は、`type-parameter-constraints-clauses` を指定しません。代わりに、オーバーライドされる基本メソッドから制約を継承します。オーバーライドされるメソッドの型パラメーターである制約は、継承された制約の型引数で置換することができます。このため、値型やシール型など、明示的に指定されている場合に無効な制約が生成される場合があります。

以下の例は、オーバーライド規則がジェネリック クラスに対してどのように機能するかを示しています。

```
abstract class C<T>
{
    public virtual T F() {...}
    public virtual C<T> G() {...}
    public virtual void H(C<T> x) {...}
}
class D: C<string>
{
    public override string F() ... // Ok
    public override C<string> G() ... // Ok
    public override void H(C<T> x) ... // Error, should be C<string>
}
class E<T,U>: C<U>
{
    public override U F() ... // Ok
    public override C<U> G() ... // Ok
    public override void H(C<T> x) ... // Error, should be C<U>
}
```

オーバーライド宣言では、*base-access* (7.6.8 を参照) を使用して、オーバーライドされた基本メソッドにアクセスできます。次に例を示します。

```
class A
{
    int x;
    public virtual void PrintFields()
    {
        Console.WriteLine("x = {0}", x);
    }
}
class B: A
{
    int y;
```

```

    public override void PrintFields() {
        base.PrintFields();
        Console.WriteLine("y = {0}", y);
    }
}

```

B での `base.PrintFields()` の呼び出しは、A で宣言された `PrintFields` メソッドを呼び出します。`base-access` は、仮想呼び出しの機構を無効にし、基本メソッドを単に非仮想メソッドとして扱います。B での呼び出しを `((A)this).PrintFields()` として記述したとしても、B で宣言された `PrintFields` メソッドが再帰的に呼び出され、A で宣言されたメソッドは呼び出されません。`PrintFields` は仮想メソッドであり、`((A)this)` の実行時の型は B であるためです。

あるメソッドで別のメソッドをオーバーライドする唯一の方法は、`override` 修飾子を含めることです。その他の場合はすべて、継承メソッドと同じシグネチャを持つメソッドが、継承メソッドを隠ぺいするだけです。次に例を示します。

```

class A
{
    public virtual void F() {}
}

class B: A
{
    public virtual void F() {} // Warning, hiding inherited F()
}

```

B の `F` メソッドには `override` 修飾子が含まれないため、A の `F` メソッドはオーバーライドされません。この場合、B の `F` メソッドが A のメソッドを隠ぺいし、宣言に `new` 修飾子が含まれないため警告が報告されます。

次に例を示します。

```

class A
{
    public virtual void F() {}
}

class B: A
{
    new private void F() {} // Hides A.F within body of B
}

class C: B
{
    public override void F() {} // Ok, overrides A.F
}

```

B の `F` メソッドは、A から継承された仮想の `F` メソッドを隠ぺいします。B の `new F` メソッドはプライベートアクセスを持つため、そのスコープに含まれるのは B のクラス本体だけで、C は含まれません。このため、C での `F` の宣言では、A から継承された `F` をオーバーライドできます。

## 10.6.5 シール メソッド

インスタンス メソッドの宣言に `sealed` 修飾子が含まれる場合、そのメソッドは "シール メソッド" と呼ばれます。インスタンス メソッドの宣言に `sealed` 修飾子が含まれる場合は、`override` 修飾子も含まれている必要があります。`sealed` 修飾子を使うと、派生クラスでメソッドをこれ以上オーバーライドしないように設定できます。

次の例を参照してください。

```
using System;
```

```

class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }
    public virtual void G() {
        Console.WriteLine("A.G");
    }
}
class B: A
{
    sealed override public void F() {
        Console.WriteLine("B.F");
    }
    override public void G() {
        Console.WriteLine("B.G");
    }
}
class C: B
{
    override public void G() {
        Console.WriteLine("C.G");
    }
}

```

クラス **B** は 2 つのオーバーライドメソッドを提供します。その 1 つは、**sealed** 修飾子を持つ **F** メソッド、もう 1 つは **sealed** 修飾子を持たない **G** メソッドです。**B** がシールされた **modifier** を使用すると、**C** は **F** をこれ以上オーバーライドできなくなります。

## 10.6.6 抽象メソッド

インスタンス メソッドの宣言に **abstract** 修飾子が含まれる場合、そのメソッドは "**抽象メソッド**" と呼ばれます。抽象メソッドは暗黙的に仮想メソッドであるにもかかわらず、**virtual** 修飾子を付けることができません。

抽象メソッドの宣言により、新規の仮想メソッドが導入されますが、そのメソッドの実装は提供されません。代わりに、メソッドをオーバーライドすることで独自の実装を提供するには、非抽象派生クラスが必要です。抽象メソッドは実際の実装を提供しないため、抽象メソッドの *method-body* を構成するのはセミコロンだけです。

抽象メソッドの宣言は、抽象クラス内でだけ許可されます (10.1.1.1 を参照)。

次に例を示します。

```

public abstract class Shape
{
    public abstract void Paint(Graphics g, Rectangle r);
}
public class Ellipse: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawEllipse(r);
    }
}

```

```
public class Box: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawRect(r);
    }
}
```

`Shape` クラスは、そのオブジェクト自体を塗りつぶすことができる、幾何学形状オブジェクトの抽象表記法を定義します。`Paint` メソッドは有効な既定の実装がないので、抽象メソッドです。`Ellipse` クラスと `Box` クラスが、`Shape` の具体的な実装です。これらのクラスは抽象クラスではないので、`Paint` メソッドをオーバーライドするために必要であり、実際の実装を提供します。

*base-access* (7.6.8 を参照) が抽象メソッドを参照するとコンパイルエラーになります。次に例を示します。

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

class B: A
{
    public override void F() {
        base.F();           // Error, base.F is abstract
    }
}
```

`base.F()` 呼び出しは抽象メソッドを参照するため、コンパイルエラーが報告されます。

抽象メソッドの宣言は、仮想メソッドをオーバーライドするために使うことができます。これにより、抽象クラスは派生クラス内でメソッドを強制的に再実装し、メソッドの元の実装は使用できなくなります。次に例を示します。

```
using System;
class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }
}

abstract class B: A
{
    public abstract override void F();
}

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

クラス `A` は仮想メソッドを宣言し、クラス `B` は抽象メソッドでこのメソッドをオーバーライドします。また、クラス `C` は抽象メソッドをオーバーライドして独自の実装を提供します。

## 10.6.7 外部メソッド

メソッドの宣言に `extern` 修飾子が含まれる場合、そのメソッドは "外部メソッド" と呼ばれます。外部メソッドは、一般的には C# 以外の言語を使って外部的に実装されます。外部メソッドの宣言は

実際の実装を提供しないので、外部メソッドの *method-body* を構成するのはセミコロンだけです。外部メソッドをジェネリックとすることはできません。

通常、`extern` 修飾子は `DllImport` 属性 (17.5.1 を参照) と組み合わせて使用されます。これにより、外部メソッドはダイナミック リンク ライブラリ (DLL: Dynamic Link Library) によって実装されます。実行環境は、他の機構をサポートして、外部メソッドの実装を実現している場合があります。

外部メソッドに `DllImport` 属性が含まれる場合は、メソッドの宣言に `static` 修飾子も含める必要があります。`extern` 修飾子と `DllImport` 属性の使用例を次に示します。

```
using System.Text;
using System.Security.Permissions;
using System.Runtime.InteropServices;

class Path
{
    [DllImport("kernel32", SetLastError=true)]
    static extern bool CreateDirectory(string name, SecurityAttribute sa);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool RemoveDirectory(string name);

    [DllImport("kernel32", SetLastError=true)]
    static extern int GetCurrentDirectory(int bufsize, StringBuilder buf);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool SetCurrentDirectory(string name);
}
```

## 10.6.8 部分メソッド

メソッドの宣言に `partial` 修飾子が含まれる場合、そのメソッドは "部分メソッド" と呼ばれます。部分メソッドは部分型 (10.2 を参照) のメンバーとしてのみ宣言でき、いくつかの制約があります。部分メソッドについては、10.2.7 で詳細に説明します。

## 10.6.9 拡張メソッド

メソッドの最初のパラメーターに `this` 修飾子が含まれる場合、そのメソッドは "拡張メソッド" と呼ばれます。拡張メソッドは、入れ子になつてない非ジェネリックの静的クラスでのみ宣言できます。拡張メソッドの最初のパラメーターには、`this` 以外の修飾子を含めることができず、パラメーター型をポインター型にすることはできません。

以下に 2 つの拡張メソッドを宣言する静的クラスの例を示します。

```
public static class Extensions
{
    public static intToInt32(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;
    }
}
```

拡張メソッドは通常の静的メソッドです。さらに、その外側の静的クラスがスコープ内にある場合、インスタンスメソッドの呼び出し構文(7.6.5.2を参照)を使い、最初の引数に受信式を使用して、拡張メソッドを呼び出すことができます。

次のプログラムでは、上記のように宣言された拡張メソッドを使用しています。

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in strings.Slice(1, 2)) {
            Console.WriteLine(s.ToInt32());
        }
    }
}
```

拡張メソッドとして宣言されているので、`string[]`に `Slice` メソッドを使用でき、`string` に `ToInt32` メソッドを使用できます。このプログラムは、通常の静的メソッドの呼び出しを使用する以下のプログラムと同じ意味を持ちます。

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in Extensions.Slice(strings, 1, 2)) {
            Console.WriteLine(Extensions.ToInt32(s));
        }
    }
}
```

## 10.6.10 メソッド本体

メソッドの宣言の *method-body* は、*block* またはセミコロンのどちらかで構成されます。

抽象メソッドおよび外部メソッドの宣言は、メソッドの実装を提供しないので、メソッドの本体を構成するのはセミコロンだけです。他のメソッドの場合、メソッド本体は、メソッド呼び出し時に実行されるステートメントを含むブロック(8.2を参照)です。

戻り値の型が `void` の場合、メソッドの戻り値の型は `void` で、メソッドが非同期の場合、戻り値の型は `System.Threading.Tasks.Task` です。それ以外の場合、非同期でないメソッドの結果の型は戻り値の型であり、戻り値の型が `System.Threading.Tasks.Task<T>` の非同期メソッドの戻り値の型は `T` です。

メソッドの結果の型が `void` の場合、そのメソッドの本体内の `return` ステートメント(8.9.4を参照)で式を指定することはできません。`void` メソッドのメソッド本体の実行が通常どおり完了する(制御がメソッド本体の末尾を離れる)と、メソッドは現在の呼び出し元に返るだけです。

メソッドの結果の型が `void` でない場合、そのメソッドの本体内の各 `return` ステートメントでは、結果の型に暗黙的に変換できる式を指定する必要があります。値を返すメソッドのメソッド本体の終了点は、到達不可能である必要があります。つまり、値を返すメソッドでは、制御がメソッド本体の末尾に到達して終了することはできません。

次に例を示します。

```
class A
{
    public int F() {} // Error, return value required
```

```

public int G() {
    return 1;
}

public int H(bool b) {
    if (b) {
        return 1;
    }
    else {
        return 0;
    }
}
}

```

この場合、値を返す `F` メソッドは、制御がメソッド本体の末尾に到達できるため、コンパイルエラーになります。`G` メソッドと `H` メソッドは、可能性のあるすべての実行パスが、戻り値を指定する `return` ステートメントで終了するので適切です。

### 10.6.11 メソッドのオーバーロード

メソッドのオーバーロードの解決規則については、7.5.2 を参照してください。

## 10.7 プロパティ

"プロパティ" は、オブジェクトまたはクラスの特性に対するアクセスを提供するメンバーです。プロパティの例としては、文字列の長さ、フォントのサイズ、ウィンドウのキャプション、顧客の名前などがあります。プロパティは、フィールドを自然な形で拡張したものと考えることができます。どちらも関連する型を持つ名前付きメンバーであり、フィールドとプロパティにアクセスする構文も同じです。ただし、フィールドとは異なり、プロパティには記憶場所の指定はありません。代わりに、プロパティには、値を読み取ったり書き込んだりするときに実行されるステートメントを指定する "アクセサー" があります。したがって、プロパティはオブジェクトの属性の読み書きにアクションを関連付けるための機構を提供し、さらに、そのオブジェクトの属性を計算できます。

プロパティは *property-declaration* を使用して宣言されます。

```

property-declaration:
  attributesopt property-modifiersopt type member-name { accessor-declarations }

property-modifiers:
  property-modifier
  property-modifiers property-modifier

property-modifier:
  new
  public
  protected
  internal
  private
  static
  virtual
  sealed
  override
  abstract
  extern

```

```
member-name:  
  identifier  
  interface-type . identifier
```

*property-declaration* には、属性(17を参照)のセット、4つのアクセス修飾子の有効な組み合わせ(10.3.5を参照)、**new**修飾子(10.3.4を参照)、**static**修飾子(10.6.2を参照)、**virtual**修飾子(10.6.3を参照)、**override**修飾子(10.6.4を参照)、**sealed**修飾子(10.6.5を参照)、**abstract**修飾子(10.6.6を参照)、および**extern**修飾子(10.6.7を参照)を指定できます。

プロパティ宣言で有効な修飾子の組み合わせは、メソッド宣言(10.6を参照)と同じです。

プロパティ宣言の*type*は、その宣言によって導入されるプロパティの型を指定し、*member-name*はプロパティの名前を指定します。プロパティが明示的なインターフェイス メンバー実装ではない場合、*member-name*は単に*identifier*です。明示的なインターフェイス メンバー実装(13.4.1を参照)の場合、*member-name*は順に、*interface-type*、".", および*identifier*で構成されます。

プロパティの*type*は、少なくともプロパティ自体と同程度にアクセス可能である必要があります(3.5.4を参照)。

*accessor-declarations* はプロパティのアクセサー(10.7.2を参照)を宣言します。また、必ず "{" トークンと "}" トークンで囲む必要があります。アクセサーは、プロパティの読み取りおよび書き込みに関連付ける実行可能なステートメントを指定します。

プロパティにアクセスするための構文がフィールドの構文と同じであっても、プロパティは変数として分類されません。したがって、引数 **ref** または引数 **out** としてプロパティを渡すことはできません。

プロパティの宣言に **extern** 修飾子が含まれる場合、そのプロパティは "外部プロパティ" と呼ばれます。外部プロパティ宣言は実際の実装を提供するものではないため、その各 *accessor-declarations* はセミコロンだけで構成されます。

### 10.7.1 静的プロパティとインスタンス プロパティ

プロパティの宣言に **static** 修飾子が含まれる場合、そのプロパティは "静的プロパティ" と呼ばれます。**static** 修飾子が存在しない場合、そのプロパティは "インスタンス プロパティ" と呼ばれます。

静的プロパティは特定のインスタンスと関連付けられていないので、静的プロパティのアクセサーで **this** を参照するとコンパイルエラーになります。

インスタンス プロパティはクラスのインスタンスに関連付けられており、そのインスタンスには、プロパティのアクセサーで **this**(7.6.7を参照)としてアクセスできます。

プロパティが **E.M** 形式の *member-access*(7.6.4を参照)で参照される場合、Mが静的プロパティであればEはMを含む型を表す必要があります、Mがインスタンス プロパティであれば、EはMを含む型のインスタンスを表す必要があります。

静的メンバーとインスタンス メンバーの違いについては、10.3.7を参照してください。

### 10.7.2 アクセサー

プロパティの *accessor-declarations* は、そのプロパティの読み取りおよび書き込みに関連付ける実行可能なステートメントを指定します。

```

accessor-declarations:
  get-accessor-declaration set-accessor-declarationopt
  set-accessor-declaration get-accessor-declarationopt

get-accessor-declaration:
  attributesopt accessor-modifieropt get accessor-body

set-accessor-declaration:
  attributesopt accessor-modifieropt set accessor-body

accessor-modifier:
  protected
  internal
  private
  protected internal
  internal protected

accessor-body:
  block
  ;

```

アクセサーの宣言は、*get-accessor-declaration* と *set-accessor-declaration* のいずれか一方、または両方で構成されます。各アクセサーの宣言は、**get** トークンまたは **set** トークンと、省略可能な *accessor-modifier* および *accessor-body* から構成されます。

*accessor-modifier* の使用には、次のような制限があります。

- *accessor-modifier* は、インターフェイスまたは明示的なインターフェイス メンバーの実装では使用できません。
- **override** 修飾子を持たないプロパティまたはインデクサーでは、*accessor-modifier* はそのプロパティまたはインデクサーが **get** アクセサーと **set** アクセサーの両方を持つ場合に限り、これらのアクセサーのいずれかに対してのみ使用できます。
- **override** 修飾子を含むプロパティまたはインデクサーでは、アクセサーはオーバーライドされるアクセサーの *accessor-modifier* (存在する場合) と一致する必要があります。
- *accessor-modifier* は、プロパティまたはインデクサー自体の宣言済みアクセシビリティよりも厳密に制限されたアクセシビリティを宣言する必要があります。厳密には、次のように処理されます。
  - プロパティまたはインデクサーが宣言済みアクセシビリティ **public** を持つ場合は、*accessor-modifier* は、**protected internal**、**internal**、**protected**、または **private** のいずれかになります。
  - プロパティまたはインデクサーが宣言済みアクセシビリティ **protected internal** を持つ場合は、*accessor-modifier* は、**internal**、**protected**、または **private** のいずれかになります。
  - プロパティまたはインデクサーが宣言済みアクセシビリティ **internal** または **protected** を持つ場合は、*accessor-modifier* は **private** である必要があります。
  - プロパティまたはインデクサーが宣言済みアクセシビリティ **private** を持つ場合は、*accessor-modifier* は使用されません。

**abstract** プロパティと **extern** プロパティの場合、指定された各アクセサーの *accessor-body* はセミコロンだけです。非 **abstract** プロパティと非 **extern** プロパティは "自動実装プロパティ" にすることができます。この場合、**get** アクセサーと **set** アクセサーの両方をセミコロンの本体 (10.7.3 を参照) で

指定する必要があります。その他の非 `abstract` プロパティと非 `extern` プロパティのアクセサーの場合、`accessor-body` は、対応するアクセサーの呼び出し時に実行されるステートメントを指定する `block` です。

`get` アクセサーは、プロパティ型の戻り値を持つパラメーターなしのメソッドに対応します。プロパティが代入のターゲットとしてではなく、式で参照される場合は、プロパティの `get` アクセサーが呼び出され、プロパティの値 (7.1.1 を参照) が計算されます。`get` アクセサーの本体は、10.6.10 で説明されるように、値を返すメソッドの規則に従う必要があります。特に、`get` アクセサーの本体に含まれるすべての `return` ステートメントは、プロパティの型に暗黙的に変換可能な式を指定する必要があります。さらに、`get` アクセサーの終了点は、到達不可能である必要があります。

`set` アクセサーは、プロパティの型の単一値パラメーターと `void` 型の戻り値を持つメソッドに対応します。`set` アクセサーの暗黙的なパラメーターの名前は常に `value` です。プロパティが代入のターゲットとして (7.17 を参照)、または `++` や `--` のオペランド (7.6.9 および 7.7.5 を参照) として参照される場合、`set` アクセサーは、新たな値を提供する (7.17.1 を参照) 引数 (代入の右側の値または `++` 演算子や `--` 演算子のオペランド) と共に呼び出されます。`set` アクセサーの本体は、10.6.10 で説明されるように、`void` メソッドの規則に従う必要があります。特に、`set` アクセサー内の `return` ステートメントで式を指定することはできません。`set` アクセサーは `value` という名前のパラメーターを暗黙的に持つので、`set` アクセサーのローカル変数宣言または定数宣言にその名前があるとコンパイルエラーになります。

`get` アクセサーと `set` アクセサーの有無に基づいて、プロパティは次のように分類されます。

- `get` アクセサーと `set` アクセサーの両方を含むプロパティは、"読み取り/書き込み" プロパティと呼ばれます。
- `get` アクセサーだけを持つプロパティは、"読み取り専用" プロパティと呼ばれます。読み取り専用プロパティに対して代入を行うと、コンパイルエラーが発生します。
- `set` アクセサーだけを持つプロパティは、"書き込み専用" プロパティと呼ばれます。代入のターゲットとして以外に、式の中で書き込み専用プロパティを参照するとコンパイルエラーになります。

次に例を示します。

```
public class Button: Control
{
    private string caption;
    public string Caption {
        get {
            return caption;
        }
        set {
            if (caption != value) {
                caption = value;
                Repaint();
            }
        }
    }
    public override void Paint(Graphics g, Rectangle r) {
        // Painting code goes here
    }
}
```

`Button` コントロールは、パブリックの `Caption` プロパティを宣言します。`Caption` プロパティの `get` アクセサーは、プライベートの `caption` フィールドに格納された文字列を返します。`set` アクセサーは、新しい値が現在の値と異なるかどうかをチェックします。異なる場合は、新しい値を格納し、コントロールを再描画します。多くの場合、プロパティは上記のパターンをたどります。`get` アクセサーはプライベート フィールドに格納された値を返すだけです。`set` アクセサーはプライベート フィールドを変更し、オブジェクトの状態を完全に更新するために必要な追加のアクションを実行します。

上の `Button` クラスを使って、次の例に `Caption` プロパティの使用方法を示します。

```
Button okButton = new Button();
okButton.Caption = "OK";           // Invokes set accessor
string s = okButton.Caption;       // Invokes get accessor
```

ここで、`set` アクセサーはプロパティに値を代入することで呼び出され、`get` アクセサーは式の中でプロパティを参照することで呼び出されます。

プロパティの `get` アクセサーと `set` アクセサーは独立したメンバーではなく、プロパティとは別にアクセサーを宣言することはできません。このため、読み取り/書き込みプロパティの 2 つのアクセサーが異なるアクセシビリティを持つことはできません。次の例を参照してください。

```
class A
{
    private string name;
    public string Name {           // Error, duplicate member name
        get { return name; }
    }
    public string Name {           // Error, duplicate member name
        set { name = value; }
    }
}
```

これは、1 つの読み取り/書き込みプロパティを宣言していません。読み取り専用と書き込み専用の 2 つのプロパティを同じ名前で宣言しています。同じクラスに宣言された 2 つのメンバーが同じ名前を持つことはできないので、上の例ではコンパイルエラーが発生します。

派生クラスが継承プロパティと同名でプロパティを宣言する場合、派生プロパティは継承プロパティを読み込みと書き込みの両方の点で隠れます。次に例を示します。

```
class A
{
    public int P {
        set {...}
    }
}
class B : A
{
    new public int P {
        get {...}
    }
}
```

`B` の `P` プロパティは、読み取りと書き込みの両方に関して、`A` の `P` プロパティを隠します。したがって、次のステートメントのようになります。

```
B b = new B();
b.P = 1;           // Error, B.P is read-only
((A)b).P = 1;    // Ok, reference to A.P
```

**B** 内の読み取り専用の **P** プロパティが **A** 内の書き込み専用の **P** プロパティを隠ぺいするので、**b.P** への代入によりコンパイルエラーが発生します。ただし、キャストを使うと、隠ぺいされた **P** プロパティにアクセスできます。

パブリック フィールドとは異なり、プロパティはオブジェクトの内部状態とパブリック インターフェイスが独立しています。次に例を示します。

```
class Label
{
    private int x, y;
    private string caption;

    public Label(int x, int y, string caption) {
        this.x = x;
        this.y = y;
        this.caption = caption;
    }

    public int X {
        get { return x; }
    }

    public int Y {
        get { return y; }
    }

    public Point Location {
        get { return new Point(x, y); }
    }

    public string Caption {
        get { return caption; }
    }
}
```

ここで、**Label** クラスは **x** と **y** という 2 つの **int** フィールドを使用して、その場所を格納します。場所は **X** プロパティと **Y** プロパティとして、および型 **Point** の **Location** プロパティとしてパブリックに公開されています。**Label** の将来のバージョンで、内部的に **Point** として場所を格納する方が便利になった場合は、クラスのパブリック インターフェイスに影響を与えることなく変更できます。

```
class Label
{
    private Point location;
    private string caption;

    public Label(int x, int y, string caption) {
        this.location = new Point(x, y);
        this.caption = caption;
    }

    public int X {
        get { return location.X; }
    }

    public int Y {
        get { return location.Y; }
    }

    public Point Location {
        get { return location; }
    }
}
```

```

    public string Caption {
        get { return caption; }
    }
}

```

しかし、`x` および `y` が `public readonly` フィールドであった場合は、`Label` クラスに同様の変更を加えることはできません。

プロパティから状態を公開するのは、フィールドを直接公開する場合と比較して必ずしも効率が落ちるわけではありません。特に、プロパティが非仮想プロパティで、少量のコードしか格納されていない場合は、実行環境によってアクセサーへの呼び出しがアクセサーの実際のコードに置き換えられることがあります。このプロセスは "インライン展開" と呼ばれ、プロパティの大きな柔軟性を保ちながら、プロパティへのアクセスをフィールドアクセスと同程度に効率化します。

`get` アクセサーの呼び出しはフィールド値の読み取りと概念的に同じなので、`get` アクセサーの使用が他に影響を及ぼすようなプログラミングスタイルは不適切と見なされます。次に例を示します。

```

class Counter
{
    private int next;
    public int Next {
        get { return next++; }
    }
}

```

`Next` プロパティの値は、プロパティがこれまでにアクセスを受けた回数に依存します。したがって、プロパティへのアクセスが明らかな影響を及ぼします。このプロパティは、代わりにメソッドとして実装する必要があります。

`get` アクセサーの "副作用をなくす" といっても、`get` アクセサーがフィールドに格納される値を単に返すように記述しなければならないという意味ではありません。実際、`get` アクセサーは多くの場合、複数のフィールドにアクセスしたり、メソッドを呼び出したりすることで、プロパティの値を計算します。ただし、適切に設計された `get` アクセサーが、オブジェクトの状態に目立つ変化を起こすアクションを実行することはできません。

プロパティを使うと、リソースが最初に参照されるまでリソースの初期化を遅らせることができます。次に例を示します。

```

using System.IO;
public class Console
{
    private static TextReader reader;
    private static TextWriter writer;
    private static TextWriter error;

    public static TextReader In {
        get {
            if (reader == null) {
                reader = new StreamReader(Console.OpenStandardInput());
            }
            return reader;
        }
    }
}

```

```

public static TextWriter Out {
    get {
        if (writer == null) {
            writer = new StreamWriter(Console.OpenStandardOutput());
        }
        return writer;
    }
}
public static TextWriter Error {
    get {
        if (error == null) {
            error = new StreamWriter(Console.OpenStandardError());
        }
        return error;
    }
}
}

```

`Console` クラスには、`In`、`Out`、および `Error` という 3 つのプロパティが含まれ、それぞれ標準入力、標準出力、エラーデバイスを表します。これらのメンバーをプロパティとして公開することで、`Console` クラスはそれらが実際に使われるまで初期化を遅らせることができます。たとえば、`Out` プロパティを最初に参照するときは次のようなときです。

```
Console.Out.WriteLine("hello, world");
```

出力デバイス用に、基になる `TextWriter` が作成されます。ただし、アプリケーションが `In` プロパティと `Error` プロパティを参照しない場合、これらのデバイスに対するオブジェクトは作成されません。

### 10.7.3 自動実装プロパティ

プロパティを自動実装プロパティとして指定する場合、プロパティに隠れたバッキングフィールドが自動的に利用できるようになり、そのバッキングフィールドに読み取りと書き込みを行うようにアクセサーが実装されます。

次に例を示します。

```

public class Point {
    public int X { get; set; } // automatically implemented
    public int Y { get; set; } // automatically implemented
}

```

これは次の宣言と同じです。

```

public class Point {
    private int x;
    private int y;
    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}

```

バッキングフィールドにアクセスできないため、それを包含している型の中でも、読み取りと書き込みを行うことができる原因是プロパティアクセサーを利用する場合のみです。つまり、自動実装される読み取り専用プロパティまたは書き込み専用プロパティが意味をなさず、認められないことになります。ただし、各アクセサーのアクセスレベルを変えて設定することができます。したがって、プライベートなバッキングフィールドの読み取り専用プロパティの効果をこのようにまねることができます。

```
public class ReadOnlyPoint {
    public int X { get; private set; }
    public int Y { get; private set; }
    public ReadOnlyPoint(int x, int y) { X = x; Y = y; }
}
```

この制約により、自動実装されたプロパティを持つ構造体型の確実な代入は、プロパティ自身の代入に構造体の確実な代入が必要になるため、構造体の標準コンストラクターを使用しなければ実現できないことも意味します。つまり、ユーザー定義のコンストラクターが既定コンストラクターを呼び出す必要があることになります。

#### 10.7.4 アクセシビリティ

アクセサーが *accessor-modifier* を持つ場合、アクセサーのアクセシビリティ ドメイン (3.5.2 を参照) は、*accessor-modifier* の宣言済みアクセシビリティを使用して決定されます。アクセサーが *accessor-modifier* を持たない場合、アクセサーのアクセシビリティ ドメインは、プロパティまたはインデクサーの宣言済みアクセシビリティから決定されます。

*accessor-modifier* が存在するかどうかは、メンバー検索 (7.3 を参照) にもオーバーロード解決にも影響しません (7.5.3 を参照)。アクセスのコンテキストとは関係なく、常にプロパティまたはインデクサーの修飾子により、バインドするプロパティまたはインデクサーが決定されます。

特定のプロパティまたはインデクサーが選択されると、関連する特定のアクセサーのアクセシビリティ ドメインを使用して、その使用が有効かどうかが決定されます。

- 値として使用する場合 (7.1.1 を参照) は、**get** アクセサーが存在し、アクセス可能である必要があります。
- 単純代入 (7.17.1 を参照) のターゲットとして使用する場合は、**set** アクセサーが存在し、アクセス可能である必要があります。
- 複合代入 (7.17.2 を参照) のターゲット、または ++ 演算子や -- 演算子 (7.5.9、7.6.5 を参照) のターゲットとして使用する場合は、**get** アクセサーと **set** アクセサーの両方が存在し、アクセス可能である必要があります。

次の例では、**set** アクセサーのみを呼び出すコンテキストの場合でも、プロパティ **A.Text** はプロパティ **B.Text** によって隠れています。これに対して、プロパティ **B.Count** はクラス **M** にアクセス可能ではなく、代わりにアクセス可能なプロパティ **A.Count** が使用されます。

```
class A
{
    public string Text {
        get { return "hello"; }
        set { }
    }
    public int Count {
        get { return 5; }
        set { }
    }
}
class B: A
{
    private string text = "goodbye";
    private int count = 0;
```

```

new public string Text {
    get { return text; }
    protected set { text = value; }
}
new protected int Count {
    get { return count; }
    set { count = value; }
}
}
class M
{
    static void Main() {
        B b = new B();
        b.Count = 12;           // Calls A.Count set accessor
        int i = b.Count;        // Calls A.Count get accessor
        b.Text = "howdy";       // Error, B.Text set accessor not accessible
        string s = b.Text;      // Calls B.Text get accessor
    }
}

```

インターフェイスの実装に使用するアクセサーに、*accessor-modifier* が存在しない場合があります。インターフェイスの実装に使用するアクセサーが 1 つのみの場合、他のアクセサーは *accessor-modifier* を使って次のように宣言できます。

```

public interface I
{
    string Prop { get; }
}

public class C: I
{
    public Prop {
        get { return "April"; }    // Must not have a modifier here
        internal set {...}        // Ok, because I.Prop has no set accessor
    }
}

```

### 10.7.5 仮想、シール、オーバーライド、抽象の各アクセサー

**virtual** プロパティ宣言は、そのプロパティのアクセサーが仮想アクセサーであることを指定します。**virtual** 修飾子は、読み取り/書き込みプロパティの両方のアクセサーに適用されます。読み取り/書き込みプロパティの片方のアクセサーだけを仮想アクセサーにすることはできません。ただし、アクセサーの 1 つが **private** の場合、**virtual** と同時に **private** であることは意味をなさないため、**virtual** 修飾子はそのアクセサーに対して無視されます。

**abstract** プロパティの宣言は、そのプロパティのアクセサーが仮想アクセサーであり、アクセサーは実際に実装されないことを指定します。代わりに、プロパティをオーバーライドすることでアクセサーの独自の実装を提供するには、非抽象派生クラスが必要です。抽象プロパティ宣言のアクセサーは実際の実装を提供するものではないため、その *accessor-body* はセミコロンだけで構成されます。

**abstract** 修飾子と **override** 修飾子の両方を含むプロパティ宣言は、そのプロパティが抽象プロパティで、基本プロパティをオーバーライドすることを指定します。この場合のプロパティのアクセサーも抽象アクセサーです。

抽象プロパティの宣言は抽象クラス (10.1.1.1 を参照) 内でしかできません。継承仮想プロパティのアクセサーは、**override** ディレクティブを指定するプロパティ宣言を含めることで、派生クラス内でオーバーライドできます。これは、"オーバーライドするプロパティの宣言" と呼ばれます。オーバーライドするプロパティの宣言は、派生クラスの抽象プロパティ宣言の直下で記述される必要があります。

バーライドするプロパティの宣言では新規プロパティは宣言されません。代わりに、既存の仮想プロパティのアクセサーに対する実装を特化するだけです。

オーバーライドするプロパティの宣言では、継承プロパティとまったく同じアクセシビリティ、修飾子、型、および名前を指定する必要があります。継承プロパティが 1 つのアクセサーだけを保有する場合(つまり、継承プロパティが読み取り専用または書き込み専用である場合)、オーバーライドするプロパティにはそのアクセサーだけを含める必要があります。継承プロパティが 2 つのアクセサーを保有する場合(つまり、継承プロパティが読み取り/書き込みである場合)、オーバーライドするプロパティには 1 つのアクセサーを含めることも、2 つのアクセサーを含めることもできます。

オーバーライドするプロパティの宣言には、`sealed` 修飾子を含めることができます。この修飾子を使うと、派生クラスでプロパティをこれ以上オーバーライドしないように設定できます。シールプロパティのアクセサーもシールされます。

宣言と呼び出し構文の違いを除けば、仮想、シール、オーバーライド、抽象の各アクセサーは、仮想、シール、オーバーライド、および抽象の各メソッドとまったくおなじように動作します。つまり、10.6.3、10.6.4、10.6.5、および 10.6.6 で説明されている規則は、アクセサーが次の対応する形式のメソッドになる場合と同じように適用されます。

- `get` アクセサーは、プロパティの型の戻り値、および包含するプロパティと同じ修飾子を持つパラメーターなしのメソッドに対応します。
- `set` アクセサーは、プロパティの型の単一値パラメーター、戻り値の型 `void`、および包含するプロパティと同じ修飾子を持つメソッドに対応します。

次に例を示します。

```
abstract class A
{
    int y;
    public virtual int X {
        get { return 0; }
    }
    public virtual int Y {
        get { return y; }
        set { y = value; }
    }
    public abstract int Z { get; set; }
}
```

`X` は仮想の読み取り専用プロパティ、`Y` は仮想の読み取り/書き込みプロパティ、`Z` は抽象の読み取り/書き込みプロパティです。`Z` は抽象プロパティなので、包含クラス `A` も抽象クラスとして宣言する必要があります。

`A` から派生するクラスを次に示します。

```
class B: A
{
    int z;
    public override int X {
        get { return base.X + 1; }
    }
    public override int Y {
        set { base.Y = value < 0? 0: value; }
    }
}
```

```

    public override int z {
        get { return z; }
        set { z = value; }
    }
}

```

ここで、`X`、`Y`、および`Z`の宣言は、プロパティの宣言をオーバーライドしています。各プロパティの宣言では、対応する継承プロパティのアクセシビリティ修飾子、型、および名前が正確に一致しています。`X`の`get`アクセサーと`Y`の`set`アクセサーは、`base`キーワードを使用して継承アクセサーにアクセスします。`Z`の宣言は、両方の抽象アクセサーをオーバーライドします。このため、`B`に未解決の抽象関数メンバーは存在せず、`B`は非抽象クラスになることができます。

プロパティを`override`として宣言する場合、オーバーライドされるアクセサーは、オーバーライドする側のコードからアクセスできる必要があります。また、プロパティ自体またはインデクサー自体の宣言済みアクセシビリティと、アクセサーの宣言済みアクセシビリティの両方が、オーバーライドされたメンバーおよびアクセサーの宣言済みアクセシビリティと一致する必要があります。次に例を示します。

```

public class B
{
    public virtual int P {
        protected set {...}
        get {...}
    }
}

public class D: B
{
    public override int P {
        protected set {...}           // Must specify protected here
        get {...}                   // Must not have a modifier here
    }
}

```

## 10.8 イベント

"イベント"は、オブジェクトやクラスが通知を発行できるようにするためのメンバーです。クライアントは、"イベントハンドラー"を指定することで、イベントに対して実行可能コードをアタッチできます。

イベントは`event-declaration`を使用して宣言されます。

```

event-declaration:
  attributesopt event-modifiersopt event type variable-declarators ;
  attributesopt event-modifiersopt event type member-name { event-accessor-declarations }

event-modifiers:
  event-modifier
  event-modifiers event-modifier

```

*event-modifier:*

**new**  
**public**  
**protected**  
**internal**  
**private**  
**static**  
**virtual**  
**sealed**  
**override**  
**abstract**  
**extern**

*event-accessor-declarations:*

*add-accessor-declaration* *remove-accessor-declaration*  
*remove-accessor-declaration* *add-accessor-declaration*

*add-accessor-declaration:*

*attributes<sub>opt</sub>* **add** *block*

*remove-accessor-declaration:*

*attributes<sub>opt</sub>* **remove** *block*

*event-declaration* には、属性(17 を参照)のセット、4つのアクセス修飾子の有効な組み合わせ(10.3.5を参照)、**new** 修飾子(10.3.4 を参照)、**static** 修飾子(10.6.2 を参照)、**virtual** 修飾子(10.6.3 を参照)、**override** 修飾子(10.6.4 を参照)、**sealed** 修飾子(10.6.5 を参照)、**abstract** 修飾子(10.6.6 を参照)、および **extern** 修飾子(10.6.7 を参照)を指定できます。

イベント宣言で有効な修飾子の組み合わせは、メソッド宣言(10.6 を参照)と同じです。

イベント宣言の *type* は *delegate-type*(4.2 を参照)である必要があり、その *delegate-type* は少なくともイベント自体と同程度にアクセス可能である必要があります(3.5.4 を参照)。

イベント宣言には、*event-accessor-declaration* を含めることができます。ただし、非 **extern** イベントと非 **abstract** イベントについてこの宣言が含まれていない場合は、コンパイラによって自動的に指定されます(10.8.1 を参照)。**extern** イベントの場合、アクセサーは外部から提供されます。

*event-accessor-declaration* を省略するイベント宣言は、*variable-declarator* ごとに1つ以上のイベントを定義します。属性および修飾子は、*event-declaration* などで宣言されるすべてのメンバーに適用されます。

*event-declaration* に **abstract** 修飾子と中かっこ({})で区切られた *event-accessor-declaration* を同時に含めると、コンパイルエラーになります。

イベント宣言に **extern** 修飾子が含まれる場合、そのイベントは“外部イベント”と呼ばれます。外部イベント宣言は実際の実装を提供するものではないため、**extern** 修飾子と *event-accessor-declaration* の両方を含めるとエラーになります。

**abstract** または **external** 修飾子を使用したイベント宣言の *variable-declarator* に *variable-initializer* を含めると、コンパイルエラーになります。

イベントは、**+=** 演算子および **-=** 演算子の左辺のオペランドで使用できます(7.17.3 を参照)。**+=** 演算子はイベントにイベントハンドラーを追加する場合に使用し、**-=** 演算子はイベントのイベントハ

ドラーを削除する場合に使用します。イベントのアクセス修飾子は、イベントでの演算が許容されるコンテキストを管理します。

`+=` 演算子と `-=` 演算子は、イベントを宣言する型の外側のイベントで許容される唯一の演算なので、外部コードはイベントハンドラーを追加および削除できますが、イベントハンドラーの基になるリストを取得または変更することはできません。

`x += y` または `x -= y` という形式の演算では、`x` がイベントで、`x` の宣言を含む型の外側で参照が発生すると、演算結果は `x` の型で代入後の `x` の値になるのではなく、`void` 型になります。この規則により、外部コードはイベントの基になるデリゲートを間接的に検査できません。

`Button` クラスのインスタンスにイベントハンドラーを追加する方法を次のコードで示します。

```
public delegate void EventHandler(object sender, EventArgs e);
public class Button: Control
{
    public event EventHandler Click;
}
public class LoginDialog: Form
{
    Button OkButton;
    Button CancelButton;
    public LoginDialog() {
        OkButton = new Button(...);
        OkButton.Click += new EventHandler(OkButtonClick);
        CancelButton = new Button(...);
        CancelButton.Click += new EventHandler(CancelButtonClick);
    }
    void OkButtonClick(object sender, EventArgs e) {
        // Handle OkButton.Click event
    }
    void CancelButtonClick(object sender, EventArgs e) {
        // Handle CancelButton.Click event
    }
}
```

ここで、`LoginDialog` インスタンス コンストラクターは、2つの `Button` インスタンスを作成し、`Click` イベントにイベントハンドラーを追加します。

### 10.8.1 フィールドのように使用するイベント

イベントの宣言を含むクラスまたは構造体のプログラム テキスト内では、特定のイベントをフィールドのように使用できます。イベントをフィールドのように使用するには、イベントが `abstract` または `extern` ではないこと、`event-accessor-declaration` を明示的に含まないことが条件です。これに該当するイベントは、フィールドを許容するすべてのコンテキストで使用できます。フィールドには、イベントに追加されたイベントハンドラーのリストを参照するデリゲート (15 を参照) が含まれています。イベントハンドラーが追加されていない場合、フィールドには `null` が含まれています。

次に例を示します。

```
public delegate void EventHandler(object sender, EventArgs e);
public class Button: Control
{
    public event EventHandler Click;
```

```

protected void OnClick(EventArgs e) {
    if (click != null) click(this, e);
}
public void Reset() {
    click = null;
}
}

```

**Button** クラス内では、`click` がフィールドとして使用されます。例が示すように、フィールドはデリゲートの呼び出し式で検査、変更、および使用できます。**Button** クラスの `onClick` メソッドにより、`click` イベントが "発生" します。イベント発生の表記は、イベントによって表されるデリゲートの呼び出しとまったく同じです。このため、イベント発生用の特別な言語構成要素はありません。デリゲートの呼び出しへは、そのデリゲートが非 NULL であることを確認するチェックの後に行われます。

**Button** クラスの宣言の外側では、`click` メンバーを使用できるのは、次のように `+=` 演算子と `-=` 演算子の左側だけです。

```
b.click += new EventHandler(...);
```

これは、`click` イベントの呼び出しリストにデリゲートを追加します。

```
b.click -= new EventHandler(...);
```

これは、`click` イベントの呼び出しリストからデリゲートを削除します。

フィールドのように使用するイベントをコンパイルすると、デリゲートを保持するストレージがコンパイラによって自動的に作成され、デリゲートフィールドのイベントハンドラーを追加または削除するイベントのアクセサーが作成されます。追加または削除操作はスレッドセーフで、インスタンスイベントであればそのイベントを含むオブジェクトのロックを保持し(8.12 を参照)、静的イベントであればそのイベントの型オブジェクトのロックを保持することができますが(7.6.10.6 を参照)、必須ではありません。

したがって、インスタンスイベントの宣言は次の形式をとります。

```

class X
{
    public event D Ev;
}

```

この宣言をコンパイルすると、次のようになります。

```

class X
{
    private D __Ev; // field to hold the delegate
    public event D Ev {
        add {
            /* add the delegate in a thread safe way */
        }
        remove {
            /* remove the delegate in a thread safe way */
        }
    }
}

```

クラス `X` 内では、`+=` 演算子と `-=` 演算子の左側の `Ev` を参照すると、`add` アクセサーおよび `remove` アクセサーが呼び出されます。`Ev` への他のすべての参照をコンパイルすると、隠しフィールド `__Ev` が

代わりに参照されます (7.6.4 を参照)。"\_\_Ev" という名前は任意です。隠しフィールドには、任意の名前を付けることができます。また、名前を付けなくてもかまいません。

### 10.8.2 イベント アクセサー

イベント宣言では通常、上記の `Button` の例のように、*event-accessor-declaration* を省略します。この宣言を省略する状況としては、1 イベントにつき 1 フィールドのストレージコストが許容されない場合があります。その場合は、クラスに *event-accessor-declaration* を含めて、イベントハンドラーのリストを格納する専用の機構を使用できます。

イベントの *event-accessor-declarations* は、イベントハンドラーの追加および削除に関連付ける実行可能なステートメントを指定します。

アクセサーの宣言は、*add-accessor-declaration* と *remove-accessor-declaration* で構成されます。各アクセサーの宣言は、順に、`add` トークンまたは `remove` トークン、および `block` で構成されます。*add-accessor-declaration* に関連付けられた `block` は、イベントハンドラーを追加すると実行されるステートメントを指定します。*remove-accessor-declaration* に関連付けられた `block` は、イベントハンドラーを削除すると実行されるステートメントを指定します。

*add-accessor-declaration* と *remove-accessor-declaration* は、イベントの種類の単一値パラメーターと `void` 型の戻り値を持つメソッドに対応します。イベントアクセサーの暗黙的なパラメーターの名前は `value` です。イベントをイベント割り当てで使用する場合は、適切なイベントアクセサーが使用されます。具体的には、代入演算子が `+=` の場合は `add` アクセサーが使用され、代入演算子が `-=` の場合は `remove` アクセサーが使用されます。どちらの場合も、代入演算子の右側のオペランドは、イベントアクセサーの引数として使用されます。*add-accessor-declaration* と *remove-accessor-declaration* のブロックは、10.6.10 で説明されているように、`void` メソッドの規則に従う必要があります。特に、アクセサー宣言のブロック内の `return` ステートメントで式を指定することはできません。

イベントアクセサーは `value` という名前のパラメーターを暗黙的に保有するので、イベントアクセサーで宣言されるローカル変数または定数にその名前があるとコンパイルエラーになります。

次に例を示します。

```
class Control: Component
{
    // Unique keys for events
    static readonly object mouseDownEventKey = new object();
    static readonly object mouseUpEventKey = new object();

    // Return event handler associated with key
    protected Delegate GetEventHandler(object key) {...}

    // Add event handler associated with key
    protected void AddEventHandler(object key, Delegate handler) {...}

    // Remove event handler associated with key
    protected void RemoveEventHandler(object key, Delegate handler) {...}

    // MouseDown event
    public event MouseEventHandler MouseDown {
        add { AddEventHandler(mouseDownEventKey, value); }
        remove { RemoveEventHandler(mouseDownEventKey, value); }
    }

    // MouseUp event
    public event MouseEventHandler MouseUp {
        add { AddEventHandler(mouseUpEventKey, value); }
        remove { RemoveEventHandler(mouseUpEventKey, value); }
    }
}
```

```
// Invoke the MouseUp event
protected void OnMouseUp(MouseEventArgs args) {
    MouseEventHandler handler;
    handler = (MouseEventHandler)GetEventHandler(mouseUpEventKey);
    if (handler != null)
        handler(this, args);
}
```

`Control` クラスは、イベントの内部ストレージ機構を実装します。`AddEventHandler` メソッドは、デリゲート値とキーを関連付け、`GetEventHandler` メソッドはキーに現在関連付けられているデリゲートを返します。また、`RemoveEventHandler` メソッドは、指定されたイベントのイベントハンドラーとしてデリゲートを削除します。基になるストレージ機構は、`null` デリゲート値をキーに関連付けるためのコストがないために、未処理のイベントはストレージを消費しないものとして設計されています。

### 10.8.3 静的イベントとインスタンス イベント

イベント宣言に `static` 修飾子が含まれる場合、そのイベントは "静的イベント" と呼ばれます。`static` 修飾子が存在しない場合、そのイベントは "インスタンスイベント" と呼ばれます。

静的イベントは特定のインスタンスと関連付けられていないので、静的イベントのアクセサーで `this` を参照するとコンパイルエラーになります。

インスタンスイベントはクラスのインスタンスに関連付けられており、このインスタンスには、イベントのアクセサーで `this` (7.6.7 を参照) としてアクセスできます。

イベントが `E.M` 形式の *member-access* (7.6.4 を参照) で参照される場合、`M` が静的イベントであれば `E` は `M` を含む型を表す必要があり、`M` がインスタンスイベントであれば `E` は `M` を含む型のインスタンスを表す必要があります。

静的メンバーとインスタンスメンバーの違いについては、10.3.7 を参照してください。

### 10.8.4 仮想、シール、オーバーライド、抽象の各アクセサー

`virtual` イベント宣言は、そのイベントのアクセサーが仮想アクセサーであることを指定します。`virtual` 修飾子は、イベントの両方のアクセサーに適用されます。

`abstract` イベント宣言は、そのイベントのアクセサーが仮想アクセサーであり、アクセサーは実際には実装されないことを指定します。代わりに、イベントをオーバーライドすることでアクセサーの独自の実装を提供するには、非抽象派生クラスが必要です。抽象イベント宣言は実際の実装を提供するものではないため、中かっこ ({} ) で区切られた *event-accessor-declaration* を指定することはできません。

`abstract` 修飾子と `override` 修飾子の両方を含むイベント宣言は、そのイベントが抽象イベントで、基本イベントをオーバーライドすることを指定します。この場合のイベントのアクセサーも抽象アクセサーです。

抽象イベントの宣言は、抽象クラス内でだけ許可されます (10.1.1.1 を参照)。

継承仮想イベントのアクセサーは、`override` 修飾子を指定するイベント宣言を含めることで、派生クラス内でオーバーライドできます。これは、"オーバーライドするイベントの宣言" と呼ばれます。オーバーライドするイベントの宣言では、新規イベントは宣言されません。代わりに、既存の仮想イベントのアクセサーに対する実装を特化するだけです。

オーバーライドするイベントの宣言では、オーバーライドされるイベントとまったく同じアクセシビリティ、修飾子、型、および名前を指定する必要があります。

オーバーライドするイベントの宣言には、**sealed** 修飾子を含めることができます。この修飾子を使うと、派生クラスでイベントをこれ以上オーバーライドしないように設定できます。シールイベントのアクセサーもシールされます。

オーバーライドするイベントの宣言に **new** 修飾子を含めるとコンパイルエラーになります。

宣言と呼び出し構文の違いを除けば、仮想、シール、オーバーライド、抽象の各アクセサーは、仮想、シール、オーバーライド、および抽象の各メソッドとまったくおなじように動作します。つまり、10.6.3、10.6.4、10.6.5、および 10.6.6 で説明されている規則は、アクセサーが次の対応する形式のメソッドになる場合と同じように適用されます。各アクセサーは、イベントの型の単一値パラメーター、戻り値の型 **void**、および包含するイベントと同じ修飾子を持つメソッドに対応します。

## 10.9 インデクサー

"インデクサー" は、配列と同じ方法でオブジェクトにインデックスを作成できるようにするためのメンバーです。インデクサーは *indexer-declaration* を使用して宣言されます。

```
indexer-declaration:  
    attributesopt indexer-modifiersopt indexer-declarator { accessor-declarations }  
  
indexer-modifiers:  
    indexer-modifier  
    indexer-modifiers indexer-modifier  
  
indexer-modifier:  
    new  
    public  
    protected  
    internal  
    private  
    virtual  
    sealed  
    override  
    abstract  
    extern  
  
indexer-declarator:  
    type this [ formal-parameter-list ]  
    type interface-type . this [ formal-parameter-list ]
```

*indexer-declaration* には、属性(17 を参照)のセット、4 つのアクセス修飾子の有効な組み合わせ(10.3.5 を参照)、**new** 修飾子(10.3.4 を参照)、**virtual** 修飾子(10.6.3 を参照)、**override** 修飾子(10.6.4 を参照)、**sealed** 修飾子(10.6.5 を参照)、**abstract** 修飾子(10.6.6 を参照)、および **extern** 修飾子(10.6.7 を参照)を指定できます。

インデクサー宣言で有効な修飾子の組み合わせは、インデクサー宣言では静的修飾子が許可されていない点を除けば、メソッド宣言(10.6 を参照)と同じです。

**virtual**、**override**、および **abstract** の各修飾子は同時に使用できませんが、例外が 1 つあります。抽象インデクサーによって仮想インデクサーをオーバーライドする場合は、**abstract** 修飾子と **override** 修飾子を同時に使用できます。

インデクサーの宣言の *type* は、宣言で導入されるインデクサーの要素型を指定します。インデクサーが明示的なインターフェイス メンバーの実装でない場合、*type* の後にはキーワード **this** を指定します。明示的なインターフェイス メンバー実装の場合、*type* は、順に、*interface-type*、". ."、およびキーワード **this** で構成されます。他のメンバーとは異なり、インデクサーにユーザー定義の名前はありません。

*formal-parameter-list* は、インデクサーのパラメーターを指定します。インデクサーの仮パラメーターリストは、メソッド(10.6.1を参照)の仮パラメーターリストに対応します。ただし、1つ以上のパラメーターを指定する必要があり、**ref** パラメーター修飾子と **out** パラメーター修飾子は使用できません。

インデクサーの *type* および *formal-parameter-list* で参照されている各型は、少なくともインデクサー自体と同程度にアクセス可能である必要があります(3.5.4を参照)。

*accessor-declarations*(10.7.2を参照)はインデクサーのアクセサーを宣言します。また、必ず "{" トークンと "}" トークンで囲む必要があります。アクセサーは、インデクサー要素の読み取りおよび書き込みに関連付ける実行可能なステートメントを指定します。

インデクサー要素にアクセスするための構文が、配列要素にアクセスするための構文と同じである場合でも、インデクサー要素は変数としては分類されません。したがって、**ref** パラメーターまたは **out** パラメーターとしてインデクサー要素を渡すことはできません。

インデクサーの仮パラメーターリストは、そのインデクサーのシグネチャ(3.6を参照)を定義します。特に、インデクサーのシグネチャは、番号と仮パラメーターの型で構成されています。仮パラメーターの要素型と名前は、インデクサーのシグネチャに含まれません。

インデクサーのシグネチャは、同じクラスで宣言されている他のすべてのインデクサーのシグネチャと異なっている必要があります。

インデクサーとプロパティの概念はよく似ていますが、次の点で異なります。

- プロパティは名前で識別されますが、インデクサーはシグネチャで識別されます。
- プロパティには *simple-name*(7.6.2を参照)または *member-access*(7.6.4を参照)を通じてアクセスしますが、インデクサー要素には *element-access*(7.6.6.2を参照)を通じてアクセスします。
- プロパティは **static** メンバーになることができますが、インデクサーは常にインスタンス メンバーです。
- プロパティの **get** アクセサーは、パラメーターのないメソッドに対応しますが、インデクサーの **get** アクセサーは、インデクサーと同じ仮パラメーターリストを持つメソッドに対応します。
- プロパティの **set** アクセサーは、**value** という名前の1つのパラメーターを持つメソッドに対応しますが、インデクサーの **set** アクセサーは、インデクサーと同じ仮パラメーターリストに加えて **value** という名前のパラメーターを持つメソッドに対応します。
- インデクサー アクセサーで、インデクサー パラメーターと同じ名前のローカル変数を宣言するとコンパイル エラーになります。
- オーバーライドするプロパティの宣言で、継承プロパティには構文 **base.P** を使ってアクセスします。ここで **P** はプロパティ名です。オーバーライドするインデクサーの宣言で、継承インデクサーには構文 **base[E]** を使ってアクセスします。ここで **E** はコンマ区切りの式リストです。

これらの違いを除けば、10.7.2 と 10.7.3 で定義されるすべての規則がインデクサー アクセサーとプロパティ アクセサーに適用されます。

インデクサーの宣言に `extern` 修飾子が含まれる場合、そのインデクサーは "外部インデクサー" と呼ばれます。外部インデクサー宣言は実際の実装を提供するものではないため、その各 *accessor-declarations* はセミコロンだけで構成されます。

インデクサーを実装する `BitArray` クラスを宣言する例を次に示します。このインデクサーは、ビット配列の各ビットにアクセスします。

```
using System;
class BitArray
{
    int[] bits;
    int length;
    public BitArray(int length) {
        if (length < 0) throw new ArgumentException();
        bits = new int[((length - 1) >> 5) + 1];
        this.length = length;
    }
    public int Length {
        get { return length; }
    }
    public bool this[int index] {
        get {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            return (bits[index >> 5] & 1 << index) != 0;
        }
        set {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            if (value) {
                bits[index >> 5] |= 1 << index;
            }
            else {
                bits[index >> 5] &= ~(1 << index);
            }
        }
    }
}
```

`BitArray` クラスのインスタンスが消費するメモリ量は、対応する `bool[]` より少なく（後者が各値に 1 バイト消費するのに対し、前者は各値に 1 ビット）、`bool[]` と同じ演算を実行できます。

`BitArray` と標準的な "古い" アルゴリズムを使って、1 から指定した最大値までの素数の数を計算する `CountPrimes` クラスを次に示します。

```

class CountPrimes
{
    static int Count(int max) {
        BitArray flags = new BitArray(max + 1);
        int count = 1;
        for (int i = 2; i <= max; i++) {
            if (!flags[i]) {
                for (int j = i * 2; j <= max; j += i) flags[j] = true;
                count++;
            }
        }
        return count;
    }

    static void Main(string[] args) {
        int max = int.Parse(args[0]);
        int count = Count(max);
        Console.WriteLine("Found {0} primes between 1 and {1}", count, max);
    }
}

```

`BitArray` の要素にアクセスするための構文は `bool[]` の場合とまったく同じです。

2つのパラメーターを持つインデクサーを保有する  $26 \times 10$  のグリッドクラスの例を次に示します。最初のパラメーターは A ~ Z の範囲の大文字または小文字、2 番目のパラメーターは 0 ~ 9 の範囲の整数である必要があります。

```

using System;
class Grid
{
    const int NumRows = 26;
    const int NumCols = 10;

    int[,] cells = new int[NumRows, NumCols];

    public int this[char c, int col] {
        get {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            return cells[c - 'A', col];
        }
        set {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            cells[c - 'A', col] = value;
        }
    }
}

```

### 10.9.1 インデクサーのオーバーロード

インデクサーのオーバーロードの解決規則については、7.5.2 を参照してください。

## 10.10 演算子

"演算子" は、クラスのインスタンスに適用できる式演算子の意味を定義するメンバーです。演算子は *operator-declaration* を使用して宣言されます。

```

operator-declaration:
    attributesopt operator-modifiers operator-declarator operator-body

operator-modifiers:
    operator-modifier
    operator-modifiers operator-modifier

operator-modifier:
    public
    static
    extern

operator-declarator:
    unary-operator-declarator
    binary-operator-declarator
    conversion-operator-declarator

unary-operator-declarator:
    type operator overloadable-unary-operator ( type identifier )

overloadable-unary-operator: one of
    +   -   !   ~   ++   --   true   false

binary-operator-declarator:
    type operator overloadable-binary-operator ( type identifier , type identifier )

overloadable-binary-operator:
    +
    -
    *
    /
    %
    &
    |
    ^
    <<
    right-shift
    ==
    !=
    >
    <
    >=
    <=

conversion-operator-declarator:
    implicit operator type ( type identifier )
    explicit operator type ( type identifier )

operator-body:
    block
    ;

```

オーバーロード可能な演算子には、単項演算子(10.10.1を参照)、二項演算子(10.10.2を参照)、および変換演算子(10.10.3を参照)があります。

演算子の宣言に `extern` 修飾子が含まれる場合、その演算子は "外部演算子" と呼ばれます。外部演算子は実際の実装を提供するものではないため、その *operator-body* はセミコロンだけで構成されます。その他のすべての演算子の場合、*operator-body* を構成するのは、演算子呼び出し時に実行するステートメントを指定する *block* です。演算子の *block* は、10.6.10で説明されるように、値を返すメソッドの規則に従う必要があります。

すべての演算子の宣言には次の規則が適用されます。

- 演算子の宣言には、`public` 修飾子と `static` 修飾子を両方とも含める必要があります。
- 演算子のパラメーターは値パラメーター(§ 5.1.4を参照)である必要があります。演算子の宣言に `ref` パラメーターまたは `out` パラメーターを含めるとコンパイルエラーになります。
- 演算子のシグネチャ(10.10.1、10.10.2、10.10.3を参照)は、同じクラスで宣言されている他のすべての演算子のシグネチャと異なっている必要があります。
- 演算子宣言で参照されるすべての型は、少なくとも演算子自体と同程度にアクセス可能である必要があります(3.5.4を参照)。
- 1つの演算子宣言内で同じ修飾子を複数回使用すると、エラーになります。

各演算子カテゴリには、以降の各トピックで説明するように、追加の制限事項があります。

基底クラスで宣言された演算子は、他のメンバーと同様に、派生クラスによって継承されます。演算子の宣言では、その演算子が宣言されるクラスまたは構造体を演算子のシグネチャに含める必要があるので、基底クラスで宣言された演算子を派生クラスで宣言された演算子で非表示にすることはできません。したがって、演算子の宣言で `new` 修飾子が必要になることはなく、許容もされません。

単項演算子および二項演算子の詳細については、7.3を参照してください。

変換演算子の詳細については、6.4を参照してください。

### 10.10.1 単項演算子。

単項演算子の宣言には次の規則が適用されます。ここで `T` は、演算子の宣言を格納するクラスまたは構造体のインスタンス型を表します。

- `+`、`-`、`!`、または`~`の各単項演算子は、型 `T` または `T?` の单一のパラメーターを受け取る必要があります、任意の型を返すことができます。
- `++` または `--` の各単項演算子は、型 `T` または `T?` の单一のパラメーターを受け取り、同じ型またはその型から派生する型を返す必要があります。
- `true` または `false` の各単項演算子は、型 `T` または `T?` の单一のパラメーターを受け取る必要があります、戻り値の型は `bool` である必要があります。

単項演算子のシグネチャは、演算子トークン(`+`、`-`、`!`、`~`、`++`、`--`、`true`、または`false`)と、1つの仮パラメーターの型で構成されます。戻り値の型は、単項演算子のシグネチャに含まれません。また、仮パラメーターの名前も含まれません。

**true** 単項演算子と **false** 単項演算子は、ペアで宣言する必要があります。クラスがこれらの演算子の一方だけを宣言すると、コンパイルエラーになります。**true** 演算子と **false** 演算子の詳細については、7.12.2 および 7.20 を参照してください。

整数ベクタークラスの **operator ++** の実装とその後の使用について、次の例で示します。

```
public class IntVector
{
    public IntVector(int length) {...}
    public int Length {...}           // read-only property
    public int this[int index] {...} // read-write indexer
    public static IntVector operator ++(IntVector iv) {
        IntVector temp = new IntVector(iv.Length);
        for (int i = 0; i < iv.Length; i++)
            temp[i] = iv[i] + 1;
        return temp;
    }
}
class Test
{
    static void Main()
    {
        IntVector iv1 = new IntVector(4); // vector of 4 x 0
        IntVector iv2;
        iv2 = iv1++; // iv2 contains 4 x 0, iv1 contains 4 x 1
        iv2 = ++iv1; // iv2 contains 4 x 2, iv1 contains 4 x 2
    }
}
```

演算子は、オペランドに 1 を足すことで生成される値を返します。これは、後置インクリメント演算子と後置デクリメント演算子 (7.6.9 を参照)、および前置インクリメント演算子と前置デクリメント演算子 (7.7.5 を参照) の場合とまったく同じです。C++ と異なり、この方法では直接オペランドの値を変更する必要はありません。実際は、オペランド値を変更すると後置インクリメント演算子の標準セマンティクスに違反します。

## 10.10.2 二項演算子。

二項演算子の宣言には次の規則が適用されます。ここで **T** は、演算子の宣言を格納するクラスまたは構造体のインスタンス型を表します。

- シフトなしの二項演算子は、少なくとも 1 つが型 **T** または **T?** の 2 つのパラメーターを受け取る必要があります、任意の型を返すことができます。
- << または >> の各二項演算子は、最初が型 **T** または **T?** で 2 番目が型 **int** または **int?** の 2 つのパラメーターを受け取る必要があります、任意の型を返すことができます。

二項演算子のシグネチャは、演算子トークン (+、-、\*、/、%、&、|、^、<<、>>、==、!=、>、<、>=、または <=) と、2 つの仮パラメーターの型で構成されます。仮パラメーターの戻り値の型と名前は、二項演算子のシグネチャに含まれません。

特定の二項演算子はペアで宣言する必要があります。ペアの一方の演算子を宣言する場合は、ペアの他方の演算子に一致する宣言が存在する必要があります。2 つの演算子の宣言 **R<sub>1</sub> op<sub>1</sub>(P<sub>1</sub>, Q<sub>1</sub>)** と **R<sub>2</sub> op<sub>2</sub>(P<sub>2</sub>, Q<sub>2</sub>)** は、戻り値型 **R<sub>1</sub>** と **R<sub>2</sub>** の間、オペランド型 **P<sub>1</sub>** と **P<sub>2</sub>** の間、オペランド型 **Q<sub>1</sub>** と **Q<sub>2</sub>** の間に恒等変換が存在する場合に一致します。ペアで宣言する必要があるのは、次の二項演算子です。

- operator ==** および **operator !=**

- `operator >` および `operator <`
- `operator >=` および `operator <=`

### 10.10.3 変換演算子

変換演算子の宣言により、"ユーザー一定義変換" (6.4 を参照) が導入されます。この変換は、暗黙的および明示的な定義済み変換を拡張します。

`implicit` キーワードを含む変換演算子の宣言では、ユーザー一定義の暗黙の型変換が導入されます。暗黙的な変換は、関数メンバー呼び出し、キャスト式、代入など、さまざまな状況で実行します。これは 0 で詳細に説明します。

`explicit` キーワードを含む変換演算子の宣言では、ユーザー一定義の明示的な型変換が導入されます。明示的な変換は、キャスト式で実行できます。詳細については、6.2 を参照してください。

変換演算子は、変換演算子のパラメーター型で表される変換元の型から、変換演算子の戻り値の型で表される変換先の型に変換します。

変換元の型  $S_0$  と変換先の型  $T_0$  が指定されている場合、 $S_0$  または  $T_0$  が `null` 許容型であれば、 $S_0$  と  $T_0$  は基になる型を参照できますが、それ以外の場合、 $S_0$  と  $T_0$  はそれぞれ  $S$  と  $T$  に等しくなります。クラスまたは構造体において変換元の型  $S_0$  から変換先の型  $T_0$  への変換を宣言できるのは、以下の条件がすべて満たされている場合だけです。

- $S_0$  と  $T_0$  の型が異なる。
- $S_0$  または  $T_0$  が、演算子の宣言が行われるクラスまたは構造体の型である。
- $S_0$  と  $T_0$  のどちらも *interface-type* ではない。
- ユーザー一定義の変換を除き、 $S_0$  から  $T_0$  への変換も  $T_0$  から  $S_0$  への変換も存在しない。

これらの規則が検討されるとき、 $S_0$  または  $T_0$  に関連付けられている型パラメーターは、他の型と継承関係のない一意の型であると見なされ、それらの型パラメーターに対する制約は無視されます。

次に例を示します。

```
class C<T> { ... }
class D<T>: C<T>
{
    public static implicit operator C<int>(D<T> value) { ... } // ok
    public static implicit operator C<string>(D<T> value) { ... } // ok
    public static implicit operator C<T>(D<T> value) { ... } // Error
}
```

最初の 2 つの演算子宣言は、10.9.3 の理由により  $T$  と `int` と `string` がそれぞれ無関係の一意の型と見なされるため、許可されます。しかし、3 番目の演算子は、`C<T>` が `D<T>` の基底クラスであるためエラーになります。

2 番目の規則から、変換演算子は、演算子の宣言が行われるクラスまたは構造体の型を変換先または変換元とする必要があることがわかります。たとえば、クラスまたは構造体の型 `C` で、`C` から `int` または `int` から `C` への変換は定義できますが、`int` から `bool` への変換は定義できません。

定義済みの変換を直接再定義することはできません。したがって、変換演算子は、`object` を変換元または変換先とする変換は許容されません。これは、`object` とその他すべての型の間に暗黙的および明示的な変換が既に存在しているためです。同様に、変換元の型と変換先の型は、どちらかの基本型になることはできません。これは、変換が既に存在しているためです。

ただし、ジェネリック型の演算子を宣言することにより、特定の型引数に対して、定義済み変換として既に存在する変換が指定されることは許されます。次に例を示します。

```
struct Convertible<T>
{
    public static implicit operator Convertible<T>(T value) {...}
    public static explicit operator T(Convertible<T> value) {...}
}
```

ここで、`object` 型を `T` の型引数として指定すると、2 番目の演算子は既に存在する変換（任意の型から `object` 型への明示的でもある暗黙の型変換が存在します）を宣言します。

2 つの型の間に定義済み変換が存在する場合、それらの型の間のユーザー定義変換は無視されます。具体的には、以下の警告があります。

- 型 `S` から型 `T` への定義済みの暗黙の型変換（6.1 を参照）が存在する場合、`S` から `T` へのすべての（暗黙または明示的な）ユーザー定義変換は無視されます。
- 型 `S` から型 `T` への定義済みの明示的な変換（6.2 を参照）が存在する場合、`S` から `T` へのすべてのユーザー定義の明示的な変換は無視されます。さらに、次の点に注意してください。
  - `T` がインターフェイス型の場合、`S` から `T` へのユーザー定義の暗黙の型変換は無視されます。
  - それ以外の場合、`S` から `T` へのユーザー定義の暗黙の型変換が考慮されます。

`object` 以外のすべての型では、上の `Convertible<T>` 型によって宣言された演算子は定義済みの変換と衝突しません。次に例を示します。

```
void F(int i, Convertible<int> n) {
    i = n;                                // Error
    i = (int)n;                            // User-defined explicit conversion
    n = i;                                // User-defined implicit conversion
    n = (Convertible<int>)i;              // User-defined implicit conversion
}
```

ただし、`object` 型では、1 つの例外を除くすべての場合において、定義済みの変換によってユーザー定義変換が隠ぺいされます。

```
void F(object o, Convertible<object> n) {
    o = n;                                // Pre-defined boxing conversion
    o = (object)n;                          // Pre-defined boxing conversion
    n = o;                                // User-defined implicit conversion
    n = (Convertible<object>)o;            // Pre-defined unboxing conversion
}
```

ユーザー定義の変換で、*interface-type* を変換元または変換先にすることはできません。特に、この制限により、*interface-type* に変換するときにユーザー定義の変換が実行されないこと、および変換中のオブジェクトが指定の *interface-type* を実際に実装する場合にだけ *interface-type* への変換が成功することが保証されます。

変換演算子のシグネチャは、変換元の型と変換先の型で構成されます。これは、メンバーの戻り値の型がシグネチャに含まれる、唯一の形式です。変換演算子が `implicit` であるか `explicit` であるか

の分類は、演算子のシグネチャには含まれません。したがって、クラスまたは構造体では、同じ変換元と変換先の型を使用して `implicit` と `explicit` の両方の変換演算子を宣言することはできません。

通常、ユーザー定義の暗黙的な変換は、例外を決してスローせず、情報を失わないように設計する必要があります。ユーザー定義の変換が例外を発生したり（変換元の引数が範囲外であるなど）、情報を失ったりする場合は（上位ビットを破棄するなど）、その変換は明示的な変換として定義する必要があります。

次に例を示します。

```
using System;
public struct Digit
{
    byte value;
    public Digit(byte value) {
        if (value < 0 || value > 9) throw new ArgumentException();
        this.value = value;
    }
    public static implicit operator byte(Digit d) {
        return d.value;
    }
    public static explicit operator Digit(byte b) {
        return new Digit(b);
    }
}
```

`Digit` から `byte` への変換は、例外をスローせず情報も失われないので、暗黙的です。しかし、`byte` から `Digit` への変換は、`Digit` が `byte` の取りうる値のサブセットを表すことができるだけなので、明示的です。

## 10.11 インスタンス コンストラクター

"インスタンス コンストラクター" は、クラスのインスタンスを初期化するために必要なアクションを実装するメンバーです。インスタンス コンストラクターは *constructor-declaration* を使用して宣言されます。

```
constructor-declaration:
    attributesopt constructor-modifiersopt constructor-declarator constructor-body

constructor-modifiers:
    constructor-modifier
    constructor-modifiers constructor-modifier

constructor-modifier:
    public
    protected
    internal
    private
    extern

constructor-declarator:
    identifier ( formal-parameter-listopt ) constructor-initializeropt

constructor-initializer:
    : base ( argument-listopt )
    : this ( argument-listopt )
```

*constructor-body*:

*block*

;

*constructor-declaration* には、*attributes* (17 を参照) の集合、4 つのアクセス修飾子の有効な組み合わせ (10.3.5 を参照)、および **extern** (10.6.7 を参照) 修飾子を指定できます。コンストラクター宣言には、同じ修飾子を複数回含めることはできません。

*constructor-declarator* の *identifier* は、そのインスタンス コンストラクターが宣言されるクラスの名前を指定する必要があります。他の名前を指定した場合は、コンパイル エラーが発生します。

インスタンス コンストラクターの省略可能な *formal-parameter-list* は、メソッドの *formal-parameter-list* (10.6 を参照) と同じ規則に従う必要があります。仮パラメーターリストは、インスタンス コンストラクターのシグネチャ (3.6 を参照) を定義し、オーバーロードの解決法 (7.5.2 を参照) が呼び出し内で特定のインスタンス コンストラクターを選択するプロセスを管理します。

インスタンス コンストラクターの *formal-parameter-list* で参照されている各型は、少なくとも、コンストラクター自体と同程度にアクセスできる必要があります (3.5.4 を参照)。

省略可能な *constructor-initializer* は、このインスタンス コンストラクターの *constructor-body* で指定されたステートメントを実行する前に呼び出す、別のインスタンス コンストラクターを指定します。これは 10.11.1 で詳細に説明します。

コンストラクター宣言に **extern** 修飾子が含まれる場合、そのコンストラクターは "外部コンストラクター" と呼ばれます。外部コンストラクター宣言は実際の実装を提供するものではないため、その *constructor-body* はセミコロンだけで構成されます。他のすべてのコンストラクターの場合、*constructor-body* は、クラスの新しいインスタンスを初期化するステートメントを指定する *block* で構成されます。これは、戻り値の型が **void** (10.6.10 を参照) であるインスタンス メソッドの *block* に完全に一致します。

インスタンス コンストラクターは継承されません。このため、クラスは、そのクラスで実際に宣言されたもの以外のインスタンス コンストラクターを保有しません。クラスにインスタンス コンストラクターの宣言がまったく含まれない場合は、既定のインスタンス コンストラクターが自動的に指定されます (10.11.4 を参照)。

インスタンス コンストラクターは、*object-creation-expression* (7.6.10.1 を参照) により、および *constructor-initializer* を通じて呼び出されます。

### 10.11.1 コンストラクター初期化子

すべてのインスタンス コンストラクター (**object** クラスのコンストラクターを除く) は、*constructor-body* の直前にある別のインスタンス コンストラクターの呼び出しを暗黙的に含みます。暗黙的に呼び出すコンストラクターは、*constructor-initializer* によって次のように決定されます。

- **base(*argument-list<sub>opt</sub>*)** 形式のインスタンス コンストラクター初期化子は、直接基底クラスからインスタンス コンストラクターを呼び出します。コンストラクターは、*argument-list* とオーバーロードの解決規則 (7.5.3 を参照) を使って選択されます。候補インスタンス コンストラクターの集合は、直接基底クラスに含まれるすべてのアクセス可能なインスタンス コンストラクターから構成されます。または直接基底クラスでインスタンス コンストラクターが宣言されていない場合は、既定のコンストラクター (10.11.4 を参照) から構成されます。このセットが空である場合は、または 1 つの最適なインスタンス コンストラクターを識別できなかった場合は、コンパイル エラーが発生します。

- `this(argument-listopt)` 形式のインスタンス コンストラクター初期化子は、そのクラス自体からインスタンス コンストラクターを呼び出します。コンストラクターは、*argument-list* とオーバーロードの解決規則 (7.5.3 を参照) を使って選択されます。呼び出す候補のインスタンス コンストラクターのセットは、そのクラス自体で宣言された、アクセス可能なすべてのインスタンス コンストラクターです。このセットが空である場合、または 1 つの最適なインスタンス コンストラクターを識別できなかった場合は、コンパイル エラーが発生します。インスタンス コンストラクターの宣言に、そのコンストラクター自体を呼び出すコンストラクター初期化子が含まれる場合は、コンパイル エラーが発生します。

インスタンス コンストラクターにコンストラクター初期化子がない場合は、`base()` 形式のコンストラクター初期化子が暗黙的に指定されます。したがって、インスタンス コンストラクターの宣言は次の形式をとります。

```
c(...)
```

これは、次の形式とまったく同じです。

```
c(...): base()
```

インスタンス コンストラクターの宣言の *formal-parameter-list* で指定されるパラメーターのスコープには、その宣言のコンストラクター初期化子が含まれます。したがって、コンストラクターのパラメーターにアクセスするためにコンストラクター初期化子を使うことができます。次に例を示します。

```
class A
{
    public A(int x, int y) {}
}
class B: A
{
    public B(int x, int y): base(x + y, x - y) {}
}
```

インスタンス コンストラクター初期化子は、作成中のインスタンスにはアクセスできません。したがって、コンストラクター初期化子の引数式で `this` を参照するとコンパイル エラーになります。同様に、引数式で *simple-name* を使用してインスタンス メンバーを参照するとコンパイル エラーになります。

### 10.11.2 インスタンス変数初期化子

インスタンス コンストラクターにコンストラクター初期化子が指定されていない場合、または `base(...)` 形式のコンストラクター初期化子が指定されている場合は、そのコンストラクターにより、そのクラスで宣言されたインスタンス フィールドの *variable-initializer* で指定された初期化が暗黙的に実行されます。これは、コンストラクターへのエントリ直後、および直接基底クラスのコンストラクターの暗黙的な呼び出しの前に実行される一連の代入に対応します。変数初期化子は、クラス宣言に出てくる順番どおりに実行されます。

### 10.11.3 コンストラクターの実行

変数初期化子は代入ステートメントに変換され、これらの代入ステートメントは基底クラスのインスタンス コンストラクターの呼び出し前に実行されます。この実行順序により、インスタンスにアクセスするステートメントが実行される前に、変数初期化子によってすべてのインスタンス フィールドの初期化が保証されます。

次に例を示します。

```

using System;
class A
{
    public A() {
        PrintFields();
    }
    public virtual void PrintFields() {}
}
class B: A
{
    int x = 1;
    int y;
    public B() {
        y = -1;
    }
    public override void PrintFields() {
        Console.WriteLine("x = {0}, y = {1}", x, y);
    }
}

```

`new B()` を使用して `B` のインスタンスを作成すると、次の出力が生成されます。

```
x = 1, y = 0
```

基底クラスのインスタンス コンストラクターが呼び出される前に変数初期化子が実行されるため、`x` の値は 1 です。ただし、`y` の値は 0 (`int` の既定値) です。`y` への代入は基底クラスのコンストラクターが値を返した後に実行されるためです。

インスタンス変数初期化子とコンストラクター初期化子は、*constructor-body* の前に自動挿入されるステートメントとして考えると便利です。次の例を参照してください。

```

using System;
using System.Collections;
class A
{
    int x = 1, y = -1, count;
    public A() {
        count = 0;
    }
    public A(int n) {
        count = n;
    }
}
class B: A
{
    double sqrt2 = Math.Sqrt(2.0);
    ArrayList items = new ArrayList(100);
    int max;
    public B(): this(100) {
        items.Add("default");
    }
    public B(int n): base(n - 1) {
        max = n;
    }
}

```

これには、複数の変数初期化子が含まれ、両方の形式 (`base` と `this`) のコンストラクター初期化子も含まれます。この例は、次に示すコードに対応しています。各コメントは、自動挿入されるステートメントを示しています。自動挿入されるコンストラクターの呼び出しに使用される構文は有効ではなく、この機構を説明するためだけのものです。

```

using System.Collections;
class A
{
    int x, y, count;
    public A() {
        x = 1;                                // Variable initializer
        y = -1;                               // Variable initializer
        object();                             // Invoke object() constructor
        count = 0;
    }
    public A(int n) {
        x = 1;                                // Variable initializer
        y = -1;                               // Variable initializer
        object();                             // Invoke object() constructor
        count = n;
    }
}
class B: A
{
    double sqrt2;
    ArrayList items;
    int max;
    public B(): this(100) {                  // Invoke B(int) constructor
        B(100);
        items.Add("default");
    }
    public B(int n): base(n - 1) {
        sqrt2 = Math.Sqrt(2.0);           // Variable initializer
        items = new ArrayList(100);       // Variable initializer
        A(n - 1);                      // Invoke A(int) constructor
        max = n;
    }
}

```

#### 10.11.4 既定のコンストラクター

クラスにインスタンス コンストラクターの宣言がまったく含まれない場合は、既定のインスタンス コンストラクターが自動的に指定されます。既定のコンストラクターは、直接基底クラスのパラメーターなしのコンストラクターを呼び出すだけです。クラスが抽象クラスの場合、既定コンストラクターの宣言されたアクセシビリティは `protected` です。それ以外の場合、既定コンストラクターの宣言されたアクセシビリティは `public` です。したがって、既定コンストラクターの宣言は常に次の形式をとります。

```
protected C(): base() {}
```

または

```
public C(): base() {}
```

ここで、`C` はクラスの名前です。オーバーロードの解決策で基底クラスのコンストラクター初期化子の一意の最適な候補が決定できない場合、コンパイル時のエラーになります。

次に例を示します。

```
class Message
{
    object sender;
    string text;
}
```

クラスにはインスタンス コンストラクターの宣言が含まれないため、既定のコンストラクターが使用されます。したがって、次の例とまったく同等です。

```
class Message
{
    object sender;
    string text;
    public Message(): base() {}
}
```

#### 10.11.5 プライベート コンストラクター

クラス `T` がプライベートインスタンス コンストラクターだけを宣言する場合は、`T` のプログラム テキストの外部から `T` の派生クラスを作成したり、`T` のインスタンスを直接作成したりすることはできません。このため、クラスに含まれるのが静的メンバーだけで、クラスをインスタンス化しない場合は、空のプライベートインスタンス コンストラクターを追加することでインスタンス化を防止できます。次に例を示します。

```
public class Trig
{
    private Trig() {} // Prevent instantiation
    public const double PI = 3.14159265358979323846;
    public static double Sin(double x) {...}
    public static double Cos(double x) {...}
    public static double Tan(double x) {...}
}
```

`Trig` クラスは、関連するメソッドと定数をグループ化しますが、インスタンス化はしません。したがって、1つの空のプライベートインスタンス コンストラクターが宣言されます。既定コンストラクターが自動生成されないようにするために、少なくとも1つのインスタンス コンストラクターを宣言する必要があります。

#### 10.11.6 省略可能なインスタンス コンストラクター パラメーター

コンストラクター初期化子の `this(...)` 形式は通常、省略可能なインスタンス コンストラクターパラメーターを実装するためにオーバーロードと共に使用されます。次に例を示します。

```
class Text
{
    public Text(): this(0, 0, null) {}
    public Text(int x, int y): this(x, y, null) {}
    public Text(int x, int y, string s) {
        // Actual constructor implementation
    }
}
```

先頭から2つのインスタンス コンストラクターは、欠けている引数用の既定値を提供するだけです。どちらも `this(...)` コンストラクター初期化子を使用して、3番目のインスタンス コンストラクターを呼び出します。このコンストラクターが、新しいインスタンスの初期化作業を実際に行います。

その効果として、以下に示すように、パラメーターを省略して、コンストラクターを呼び出すことができます。

```
Text t1 = new Text();           // Same as Text(0, 0, null)
Text t2 = new Text(5, 10);      // Same as Text(5, 10, null)
Text t3 = new Text(5, 20, "Hello");
```

## 10.12 静的コンストラクター

"*静的コンストラクター*" は、クローズクラス型を初期化するために必要なアクションを実装するメンバーです。静的コンストラクターは *static-constructor-declaration* を使用して宣言されます。

*static-constructor-declaration:*

*attributes<sub>opt</sub>* *static-constructor-modifiers* *identifier* ( ) *static-constructor-body*

*static-constructor-modifiers:*

*extern<sub>opt</sub>* **static**  
**static** *extern<sub>opt</sub>*

*static-constructor-body:*

*block*  
;

*static-constructor-declaration* には、*attributes* (17 を参照) の集合と **extern** 修飾子 (10.6.7 を参照) を含めることができます。

*static-constructor-declaration* の *identifier* は、静的コンストラクターが宣言されるクラスの名前を指定する必要があります。他の名前を指定した場合は、コンパイルエラーが発生します。

静的コンストラクター宣言に **extern** 修飾子が含まれる場合、その静的コンストラクターは "**外部静的コンストラクター**" と呼ばれます。外部静的コンストラクター宣言は実際の実装を提供するものではないため、その *static-constructor-body* はセミコロンだけで構成されます。他のすべての静的コンストラクター宣言の場合、*static-constructor-body* は、クラスを初期化するために実行するステートメントを指定する *block* で構成されます。これは、戻り値の型が **void** (10.6.10 を参照) である静的メソッドの *method-body* に完全に一致します。

静的コンストラクターは継承されず、直接呼び出すこともできません。

指定のアプリケーションドメインでクローズクラス型の静的コンストラクターを実行できるのは、1回だけです。静的コンストラクターは、次のどちらかのイベントがアプリケーションドメインで最初に発生したときに実行されます。

- クラス型のインスタンスが作成されたとき
- クラス型のいずれかの静的メンバーが参照されたとき

**Main** メソッド (3.1 を参照) が含まれているクラスで実行が開始される場合は、**Main** メソッドが呼び出される前にそのクラスの静的コンストラクターが実行されます。

新しいクローズクラス型を初期化するには、まず、特定のクローズ型の静的フィールド (10.5.1 を参照) の新しいセットを作成します。各静的フィールドを既定値 (5.2 を参照) に初期化します。次に、それらの静的フィールドに対して、静的フィールドの初期化子 (10.4.5.1 を参照) を実行します。最後に、静的コンストラクターを実行します。

次の例を参照してください。

```
using System;
class Test
{
    static void Main() {
        A.F();
        B.F();
    }
}
class A
{
    static A() {
        Console.WriteLine("Init A");
    }
    public static void F() {
        Console.WriteLine("A.F");
    }
}
class B
{
    static B() {
        Console.WriteLine("Init B");
    }
    public static void F() {
        Console.WriteLine("B.F");
    }
}
```

出力は次のようにになります。

```
Init A
A.F
Init B
B.F
```

このように出力されるのは、**A** の静的コンストラクターの実行が **A.F** の呼び出しによって発生し、**B** の静的コンストラクターの実行が **B.F** の呼び出しによって発生するためです。

循環依存関係を構築すると、変数初期化子を持つ静的フィールドを既定値の状態で確認できます。

次の例を参照してください。

```
using System;
class A
{
    public static int X;
    static A() {
        X = B.Y + 1;
    }
}
class B
{
    public static int Y = A.X + 1;
    static B() {}
    static void Main() {
        Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);
    }
}
```

この例では、次のように出力されます。

```
X = 1, Y = 2
```

`Main` メソッドを実行するために、システムは最初に `B.Y` の初期化子を実行してから、クラス `B` の静的コンストラクターを実行します。`A.X` の値が参照されているため、`Y` の初期化子によって `A` の静的コンストラクターが実行されます。続いて、`A` の静的コンストラクターが `X` の値を計算し、`Y` の既定値 `0` をフェッチします。したがって、`A.X` は `1` に初期化されます。`A` の静的フィールド初期化子と静的コンストラクターの実行が完了すると、`Y` の初期値の計算に戻ります。計算結果は `2` になります。

静的コンストラクターは、各クローズ構築クラス型に対して 1 回だけ実行されるため、コンパイル時に制約 (10.1.5 を参照) でチェックできない型パラメーターを実行時にチェックする場所として適しています。たとえば、次の型は、静的コンストラクターを使用して型引数を `enum` にします。

```
class Gen<T> where T: struct
{
    static Gen() {
        if (!typeof(T).IsEnum) {
            throw new ArgumentException("T must be an enum");
        }
    }
}
```

## 10.13 デストラクター

"デストラクター" は、クラスのインスタンスを消滅させるために必要なアクションを実装するメンバーです。デストラクターは `destructor-declaration` を使用して宣言されます。

```
destructor-declaration:
    attributesopt externopt ~ identifier ( ) destructor-body
destructor-body:
    block
    ;
```

`destructor-declaration` には `attributes` (17 を参照) の集合を含めることができます。

`destructor-declarator` の `identifier` は、そのデストラクターが宣言されるクラスの名前を指定する必要があります。他の名前を指定した場合は、コンパイルエラーが発生します。

デストラクターの宣言に `extern` 修飾子が含まれる場合、そのデストラクターは "外部デストラクター" と呼ばれます。外部デストラクター宣言は実際の実装を提供するものではないため、その `destructor-body` はセミコロンだけで構成されます。他のすべてのデストラクターの場合、`destructor-body` は、クラスのインスタンスを破棄するために実行するステートメントを指定する `block` で構成されます。`destructor-body` は、戻り値の型が `void` (10.6.10 を参照) であるインスタンスメソッドの `method-body` に完全に一致します。

デストラクターは継承されません。このため、クラスは、そのクラスで実際に宣言されたもの以外のデストラクターを保有しません。

デストラクターにパラメーターは不要なので、オーバーロードできません。このため、クラスが保有できるデストラクターは 1 つだけです。

デストラクターは自動的に呼び出されるものであり、明示的に呼び出すことはできません。どのコードもそのインスタンスを使用できなくなると、インスタンスは消滅する資格を得ます。インスタンスのデストラクターの実行は、インスタンスが消滅の資格を得た後はいつでも発生する可能性があります。インスタンスを消滅させると、そのインスタンスの継承チェーン内のデストラクターが最派生か

ら最低派生まで順に呼び出されます。デストラクターは、任意のスレッドで実行できます。デストラクターを実行する場合と方法を管理する規則の詳細については、3.9 を参照してください。

出力例は次のとおりです。

```
using System;
class A
{
    ~A() {
        Console.WriteLine("A's destructor");
    }
}
class B: A
{
    ~B() {
        Console.WriteLine("B's destructor");
    }
}
class Test
{
    static void Main() {
        B b = new B();
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
```

結果

```
B's destructor
A's destructor
```

これは、継承チェイン内のデストラクターが最派生から最低派生まで順に呼び出されるためです。

デストラクターは、`System.Object` の仮想メソッドである `Finalize` をオーバーライドすることによって実装されます。C# プログラムでは、このメソッドをオーバーライドしたり、直接そのメソッドやオーバーライドを呼び出すことはできません。たとえば、次のプログラムがあるとします。

```
class A
{
    override protected void Finalize() {} // error
    public void F() {
        this.Finalize(); // error
    }
}
```

このプログラムでは、2 つのエラーが発生します。

コンパイラはこのメソッド、およびそのオーバーライドがまったく存在していないかのように動作します。したがって、プログラムを次のようにします。

```
class A
{
    void Finalize() {} // permitted
```

これは有効であり、上のメソッドは `System.Object` の `Finalize` メソッドを隠ぺいします。

例外がデストラクターからスローされたときの動作については、16.3 を参照してください。

## 10.14 反復子

反復子ブロック (7.5 を参照) を使用して実装された関数メンバー (8.2 を参照) は "反復子" と呼ばれます。

反復子ブロックは、対応する関数メンバーの戻り値の型が列挙子インターフェイス (10.14.1 を参照) のいずれか、または列挙可能なインターフェイス (10.14.2 を参照) のいずれかである限り、関数メンバーの本体として使用できます。反復子ブロックは *method-body*、*operator-body*、または *accessor-body* になることができますが、イベント、インスタンス コンストラクター、静的コンストラクター、およびデストラクターは反復子として実装できません。

関数メンバーが反復子ブロックを使用して実装された場合は、関数メンバーの仮パラメーターリストに `ref` パラメーターまたは `out` パラメーターを指定するとコンパイルエラーになります。

### 10.14.1 列挙子インターフェイス

"列挙子インターフェイス" は、非ジェネリックインターフェイス

`System.Collections.IEnumerator` と、ジェネリックインターフェイス

`System.Collections.Generic.IEnumerable<T>` のすべてのインスタンス化です。説明を簡潔にするために、この章ではこれらのインターフェイスをそれぞれ `IEnumerator` および `IEnumerable<T>` と呼びます。

### 10.14.2 列挙可能なインターフェイス

"列挙可能なインターフェイス" は、非ジェネリックインターフェイス

`System.Collections.IEnumerable` と、ジェネリックインターフェイス

`System.Collections.Generic.IEnumerable<T>` のすべてのインスタンス化です。説明を簡潔にするために、この章ではこれらのインターフェイスをそれぞれ `IEnumerable` および `IEnumerable<T>` と呼びます。

### 10.14.3 yield 型

反復子は、すべて同じ型の一連の値を生成します。この型は、反復子の "*yield* 型" と呼ばれます。

- `IEnumerator` または `IEnumerable` を返す反復子の *yield* 型は `object` です。
- `IEnumerator<T>` または `IEnumerable<T>` を返す反復子の *yield* 型は `T` です。

### 10.14.4 列挙子オブジェクト

列挙子インターフェイス型を返す関数メンバーを反復子ブロックを使用して実装した場合は、関数メンバーを呼び出しても反復子ブロックのコードはすぐに実行されません。代わりに、"列挙子オブジェクト" が作成されて返されます。このオブジェクトは、反復子ブロックに指定されたコードをカプセル化し、列挙子オブジェクトの `MoveNext` メソッドを呼び出すと反復子ブロックのコードを実行します。列挙子オブジェクトには、次のような特性があります。

- 列挙子オブジェクトは、`IEnumerator` および `IEnumerator<T>` を実装します (`T` は反復子の *yield* 型)。
- `System.IDisposable` を実装します。
- 列挙子オブジェクトは、関数メンバーに渡された引数値 (存在する場合) のコピーおよびインスタンス値によって初期化されます。

- 列挙子オブジェクトには、**before**、**running**、**suspended**、および**after** の 4 つの状態があり、初期状態は **before** 状態です。

列挙子オブジェクトは、一般的にはコンパイラによって生成された列挙子クラスのインスタンスで、反復子ブロック内にコードをカプセル化し、列挙子インターフェイスを実行します。しかし、他の方法による実装もあります。列挙子クラスがコンパイラによって生成された場合、そのクラスは、関数メンバーを含むクラスの直接的または間接的な入れ子になり、プライベートなアクセシビリティを持ち、コンパイル用に予約された名前 (2.4.2 を参照) を持ります。

列挙子オブジェクトは、上記で指定されていないインターフェイスも実装できます。

この後のセクションでは、列挙子オブジェクトによって提供される、**IEnumerable** インターフェイスの実装および **IEnumerable<T>** インターフェイスの実装の **MoveNext**、**Current**、および **Dispose** メンバーの厳密な動作について説明します。

列挙子オブジェクトは、**IEnumerator.Reset** メソッドをサポートしません。このメソッドを呼び出されると、**System.NotSupportedException** がスローされます。

#### 10.14.4.1 MoveNext メソッド

列挙子オブジェクトの **MoveNext** メソッドは、反復子ブロックのコードをカプセル化します。

**MoveNext** メソッドを呼び出すと、反復子ブロックのコードが実行され、列挙子オブジェクトの **Current** プロパティが適切に設定されます。**MoveNext** によって実行されるアクションは、**MoveNext** が呼び出されたときの列挙子オブジェクトの状態によって異なります。

- 列挙子オブジェクトの状態が **before** の場合は、**MoveNext** を呼び出すと次のように処理されます。
    - 状態を **running** に変更します。
    - 反復子ブロックのパラメーター (**this** を含む) を、列挙子オブジェクトの初期化時に保存された引数値とインスタンス値に初期化します。
    - 反復子ブロックを、先頭から実行が中断されるまで実行します (下記を参照)。
  - 列挙子オブジェクトの状態が **running** の場合、**MoveNext** の呼び出しの結果は未指定です。
  - 列挙子オブジェクトの状態が **suspended** の場合は、**MoveNext** を呼び出すと次のように処理されます。
    - 状態を **running** に変更します。
    - すべてのローカル変数およびパラメーター (**this** を含む) を、反復子ブロックの実行が前回中断されたときに保存された値に復元します。これらの変数が参照するオブジェクトの内容は、それまでの **MoveNext** の呼び出しによって変更されている場合があります。
    - 実行の中断を引き起こした **yield return** ステートメントのすぐ後の反復子ブロックの実行を再開し、実行が中断されるまで継続します (下記を参照)。
  - 列挙子オブジェクトの状態が **after** の場合は、**MoveNext** を呼び出すと **false** が返されます。
- MoveNext** による反復子ブロックの実行は、**yield return** ステートメント、**yield break** ステートメント、反復子ブロックの最後、および反復子ブロックの外側に伝達される例外のスローという 4 つおりの方法で中断されます。
- yield return** ステートメントがある場合 (8.14 を参照) は、次のように処理されます。

- ステートメントに指定されている式を評価し、暗黙的に `yield` 型に変換し、列挙子オブジェクトの `Current` プロパティに代入します。
- 反復子本体の実行を中断します。すべてのローカル変数およびパラメーター (`this` を含む) の値と、この `yield return` ステートメントの位置を保存します。`yield return` ステートメントが 1 つ以上の `try` ブロック内にある場合、この時点では関連付けられた `finally` ブロックは実行されません。
- 列挙子オブジェクトの状態が `suspended` に変更されます。
- `MoveNext` メソッドは呼び出し元に `true` を返し、反復処理が正常に次の値に進んだことを示します。
- `yield break` ステートメントがある場合 (8.14 を参照) は、次のように処理されます。
  - `yield break` ステートメントが 1 つ以上の `try` ブロック内にある場合は、関連付けられた `finally` ブロックは実行されません。
  - 列挙子オブジェクトの状態が `after` に変更されます。
  - `MoveNext` メソッドは呼び出し元に `false` を返し、反復処理が完了したことを示します。
- 反復子本体の最後に達すると、次の処理が行われます。
  - 列挙子オブジェクトの状態が `after` に変更されます。
  - `MoveNext` メソッドは呼び出し元に `false` を返し、反復処理が完了したことを示します。
- 例外がスローされ、反復子ブロックの外側に伝達されると、次の処理が行われます。
  - 例外の伝達により、反復子本体の適切な `finally` ブロックが実行されていることになります。
  - 列挙子オブジェクトの状態が `after` に変更されます。
  - 例外の伝達が `MoveNext` メソッドの呼び出し元まで続行されます。

#### 10.14.4.2 Current プロパティ

列挙子オブジェクトの `Current` プロパティは、反復子ブロックの `yield return` ステートメントの影響を受けます。

列挙子オブジェクトが `suspended` 状態の場合、`Current` の値は以前の `MoveNext` の呼び出しで設定された値になります。列挙子オブジェクトが `before`、`running`、または `after` の場合、`Current` にアクセスした結果は未指定です。

`yield` 型が `object` 以外である反復子では、列挙子オブジェクトの `IEnumerable` 実装を通じて `Current` にアクセスした結果は、列挙子オブジェクトの `IEnumerator<T>` 実装を通じて `Current` にアクセスした結果を `object` にキャストした場合に対応します。

#### 10.14.4.3 Dispose メソッド

`Dispose` メソッドを使用して列挙子オブジェクトを `after` 状態にすることにより、反復処理をクリーンアップできます。

- 列挙子オブジェクトの状態が `before` の場合は、`Dispose` を呼び出すと状態が `after` に変わります。
- 列挙子オブジェクトの状態が `running` の場合、`Dispose` の呼び出しの結果は未指定です。

- 列挙子オブジェクトの状態が **suspended** の場合は、**Dispose** を呼び出すと次のように処理されます。
  - 状態を **running** に変更します。
  - 最後に実行した **yield return** ステートメントを **yield break** ステートメントと見なして、**finally** ブロックを実行します。この処理によって例外がスローされ、反復子本体の外側に伝達された場合は、列挙子オブジェクトの状態が **after** に設定され、例外は **Dispose** メソッドの呼び出し元に伝達されます。
  - 状態を **after** に変更します。
- 列挙子オブジェクトの状態が **after** の場合は、**Dispose** を呼び出しても効果はありません。

#### 10.14.5 列挙可能なオブジェクト

列挙可能なインターフェイス型を返す関数メンバーを反復子ブロックを使用して実装した場合は、関数メンバーを呼び出しても反復子ブロックのコードはすぐに実行されません。代わりに、"列挙可能なオブジェクト" が作成されて返されます。列挙可能なオブジェクトの **GetEnumerator** メソッドは、反復子ブロック内に指定されているコードをカプセル化する列挙子オブジェクトを返し、反復子ブロック内のコードは、列挙子オブジェクトの **MoveNext** メソッドを呼び出すと実行されます。列挙可能なオブジェクトには、次のような特性があります。

- 列挙子オブジェクトは、**IEnumerable** および **IEnumerable<T>** を実装します (**T** は反復子の **yield** 型)。
- 列挙子オブジェクトは、関数メンバーに渡された引数値(存在する場合)のコピーおよびインスタンス値によって初期化されます。

列挙可能なオブジェクトは、一般的にはコンパイラによって生成された列挙可能なクラスのインスタンスで、反復子ブロック内にコードをカプセル化し、列挙可能なインターフェイスを実行します。しかし、他の方法による実装もあります。列挙可能クラスがコンパイラによって生成された場合、そのクラスは、関数メンバーを含むクラスの直接的または間接的な入れ子になり、プライベートなアクセシビリティを持ち、コンパイル用に予約された名前(2.4.2 を参照)を持ちます。

列挙可能なオブジェクトは、上記で指定されていないインターフェイスも実装できます。特に、列挙可能なオブジェクトは **IEnumerator** と **IEnumerator<T>** を実装して、列挙可能なオブジェクトと列挙子オブジェクトの両方の働きをする場合があります。このような実装では、最初は列挙可能なオブジェクトの **GetEnumerator** メソッドが呼び出されて列挙可能なオブジェクトが返されます。引き続き列挙可能なオブジェクトの **GetEnumerator**(存在する場合)を呼び出すと、列挙可能なオブジェクトのコピーが返されます。したがって、返される列挙子の状態はそれぞれ独立しており、1つの列挙子の変化は他の列挙子に影響しません。

##### 10.14.5.1 GetEnumerator メソッド

列挙可能なオブジェクトは、**IEnumerable** および **IEnumerable<T>** インターフェイスの **GetEnumerator** メソッドの実装を提供します。2つの **GetEnumerator** メソッドは、使用可能な列挙子オブジェクトを取得して返す共通の実装を共有します。列挙子オブジェクトは、列挙可能なオブジェクトが初期化されたときに保存された引数値とインスタンス値で初期化されますが、それ以外では、10.14.4 で説明されているように機能します。

### 10.14.6 実装例

ここでは、使用できる反復子の実装について、標準の C# 構造に基づいて説明します。ここで説明する実装は、Microsoft C# コンパイラの原理と同じ原理に基づきますが、必須の実装ではなく、また唯一可能な実装でもありません。

次の `Stack<T>` クラスは、反復子を使用する `GetEnumerator` メソッドを実装します。反復子によって、スタックの要素が上から下へ列挙されます。

```
using System;
using System.Collections;
using System.Collections.Generic;

class Stack<T>: IEnumerable<T>
{
    T[] items;
    int count;

    public void Push(T item) {
        if (items == null) {
            items = new T[4];
        }
        else if (items.Length == count) {
            T[] newItems = new T[count * 2];
            Array.Copy(items, 0, newItems, 0, count);
            items = newItems;
        }
        items[count++] = item;
    }

    public T Pop() {
        T result = items[--count];
        items[count] = default(T);
        return result;
    }

    public IEnumerator<T> GetEnumerator() {
        for (int i = count - 1; i >= 0; --i) yield return items[i];
    }
}
```

`GetEnumerator` メソッドは、次のように、反復子ブロック内にコードをカプセル化する、コンパイラにより生成された列挙子クラスのインスタンス化に書き換えることができます。

```
class Stack<T>: IEnumerable<T>
{
    ...
    public IEnumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }

    class __Enumerator1: IEnumerator<T>, IEnumerable
    {
        int __state;
        T __current;
        Stack<T> __this;
        int i;

        public __Enumerator1(Stack<T> __this) {
            this.__this = __this;
        }

        public T Current {
            get { return __current; }
        }
    }
}
```

```

        object IEnumator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            switch (__state) {
                case 1: goto __state1;
                case 2: goto __state2;
            }
            i = __this.count - 1;
            __loop:
            if (i < 0) goto __state2;
            __current = __this.items[i];
            __state = 1;
            return true;
            __state1:
            --i;
            goto __loop;
            __state2:
            __state = 2;
            return false;
        }

        public void Dispose() {
            __state = 2;
        }

        void IEnumator.Reset() {
            throw new NotSupportedException();
        }
    }
}

```

上記の書き換えでは、反復子ブロックのコードはステートマシンになり、列挙子クラスの `MoveNext` メソッドに配置されます。また、ローカル変数 `i` は列挙子オブジェクトのフィールドになり、`MoveNext` の複数の呼び出しにわたって存在し続けることができます。

次の例は、1 から 10 までの整数の単純な乗算テーブルを出力します。この例の `FromTo` メソッドは、列挙可能なオブジェクトを返し、反復子を使用して実装されます。

```

using System;
using System.Collections.Generic;
class Test
{
    static IEnumerable<int> FromTo(int from, int to) {
        while (from <= to) yield return from++;
    }

    static void Main() {
        IEnumerable<int> e = FromTo(1, 10);
        foreach (int x in e) {
            foreach (int y in e) {
                Console.WriteLine("{0},{1} {2}", x, y, x * y);
            }
            Console.WriteLine();
        }
    }
}

```

`FromTo` メソッドは、次のように、反復子ブロック内にコードをカプセル化する、コンパイラにより生成された列挙可能なクラスのインスタンス化に書き換えることができます。

```

using System;
using System.Threading;
using System.Collections;
using System.Collections.Generic;
class Test
{
    ...
    static IEnumerable<int> FromTo(int from, int to) {
        return new __Enumerable1(from, to);
    }
    class __Enumerable1:
        IEnumerable<int>, IEnumerable,
        IEnumerator<int>, Ienumerator
    {
        int __state;
        int __current;
        int __from;
        int from;
        int to;
        int i;
        public __Enumerable1(int __from, int to) {
            this.__from = __from;
            this.to = to;
        }
        public IEnumerator<int> GetEnumerator() {
            __Enumerable1 result = this;
            if (Interlocked.CompareExchange(ref __state, 1, 0) != 0) {
                result = new __Enumerable1(__from, to);
                result.__state = 1;
            }
            result.from = result.__from;
            return result;
        }
        IEnumerator IEnumerable.GetEnumerator() {
            return (IEnumerator)GetEnumerator();
        }
        public int Current {
            get { return __current; }
        }
        object IEnumerator.Current {
            get { return __current; }
        }
        public bool MoveNext() {
            switch (__state) {
                case 1:
                    if (from > to) goto case 2;
                    __current = from++;
                    __state = 1;
                    return true;
                case 2:
                    __state = 2;
                    return false;
                default:
                    throw new InvalidOperationException();
            }
        }
    }
}

```

```

        public void Dispose() {
            __state = 2;
        }
        void Ienumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}

```

列挙可能なクラスは列挙可能なインターフェイスと列挙子インターフェイスの両方を実装することで、列挙可能なインターフェイスと列挙子インターフェイスの両方として働くことができます。

`GetEnumerator` メソッドを初めて呼び出すと、列挙可能なオブジェクトそのものが返されます。引き続き列挙可能なオブジェクトの `GetEnumerator` (存在する場合) を呼び出すと、列挙可能なオブジェクトのコピーが返されます。したがって、返される列挙子の状態はそれぞれ独立しており、1つの列挙子の変化は他の列挙子に影響しません。`Interlocked.CompareExchange` メソッドを使用して、スレッドセーフ操作を実現できます。

`from` パラメーターと `to` パラメーターは、列挙可能なクラスのフィールドになります。`from` は反復子プロック内で変更されるため、各列挙子の `from` に指定される初期値を保持するために `__from` フィールドが追加されています。

`MoveNext` メソッドは、`__state` が 0 のときに呼び出すと `InvalidOperationException` をスローします。これにより、最初に `GetEnumerator` を呼び出してからでなければ、列挙可能オブジェクトを列挙子オブジェクトとして使用できないように保護されます。

次の例は、単純なツリークラスを示します。`Tree<T>` クラスは、反復子を使用する `GetEnumerator` メソッドを実装します。反復子は、ツリーの要素を infix 順に列挙します。

```

using System;
using System.Collections.Generic;
class Tree<T>: IEnumerable<T>
{
    T value;
    Tree<T> left;
    Tree<T> right;
    public Tree(T value, Tree<T> left, Tree<T> right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }
    public IEnumerator<T> GetEnumerator() {
        if (left != null) foreach (T x in left) yield x;
        yield value;
        if (right != null) foreach (T x in right) yield x;
    }
}
class Program
{
    static Tree<T> MakeTree<T>(T[] items, int left, int right) {
        if (left > right) return null;
        int i = (left + right) / 2;
        return new Tree<T>(items[i],
            MakeTree(items, left, i - 1),
            MakeTree(items, i + 1, right));
    }
}

```

## C# LANGUAGE SPECIFICATION

```
static Tree<T> MakeTree<T>(params T[] items) {
    return MakeTree(items, 0, items.Length - 1);
}
// The output of the program is:
// 1 2 3 4 5 6 7 8 9
// Mon Tue Wed Thu Fri Sat Sun
static void Main() {
    Tree<int> ints = MakeTree(1, 2, 3, 4, 5, 6, 7, 8, 9);
    foreach (int i in ints) Console.WriteLine("{0}", i);
    Console.WriteLine();
    Tree<string> strings = MakeTree(
        "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun");
    foreach (string s in strings) Console.WriteLine("{0}", s);
    Console.WriteLine();
}
```

`GetEnumerator` メソッドは、次のように、反復子ブロック内にコードをカプセル化する、コンパイラにより生成された列挙子クラスのインスタンス化に書き換えることができます。

```
class Tree<T>: IEnumerable<T>
{
    ...
    public IEnumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }
    class __Enumerator1 : IEnumerator<T>, IEnumerable
    {
        Node<T> __this;
        IEnumerator<T> __left, __right;
        int __state;
        T __current;
        public __Enumerator1(Node<T> __this) {
            this.__this = __this;
        }
        public T Current {
            get { return __current; }
        }
        object IEnumerator.Current {
            get { return __current; }
        }
        public bool MoveNext() {
            try {
                switch (__state) {
                    case 0:
                        __state = -1;
                        if (__this.left == null) goto __yield_value;
                        __left = __this.left.GetEnumerator();
                        goto case 1;
                    case 1:
                        __state = -2;
                        if (!__left.MoveNext()) goto __left_Dispose;
                        __current = __left.Current;
                        __state = 1;
                        return true;
                }
            }
            finally {
                __left?.Dispose();
            }
        }
        void __yield_value() {
            __current = __left.Current;
        }
        void __left_Dispose() {
            __left?.Dispose();
        }
    }
}
```

```

    __left_Dispose:
        __state = -1;
        __left.Dispose();

    __yield_value:
        __current = __this.value;
        __state = 2;
        return true;

    case 2:
        __state = -1;
        if (__this.right == null) goto __end;
        __right = __this.right.GetEnumerator();
        goto case 3;

    case 3:
        __state = -3;
        if (!__right.MoveNext()) goto __right_Dispose;
        __current = __right.Current;
        __state = 3;
        return true;

    __right_Dispose:
        __state = -1;
        __right.Dispose();

    __end:
        __state = 4;
        break;
    }

}

finally {
    if (__state < 0) Dispose();
}
return false;
}

public void Dispose() {
    try {
        switch (__state) {

    case 1:
    case -2:
        __left.Dispose();
        break;

    case 3:
    case -3:
        __right.Dispose();
        break;
    }
}
finally {
    __state = 4;
}
}

void Ienumerator.Reset() {
    throw new NotSupportedException();
}
}

```

コンパイラによって生成され、`foreach`ステートメントで使用される一時ファイルは、列挙子オブジェクトの`_left`フィールドおよび`_right`フィールドにリフトされます。列挙子オブジェクトの`state`フィールドは、例外がスローされた場合に正しい`Dispose()`メソッドが正しく呼び出され

るよう、厳密に更新されます。これは、単純な `foreach` ステートメントのコードに書き換えることはできません。

## 10.15 非同期関数

`async` 修飾子を持つメソッド(10.6 を参照)または匿名関数(7.15 を参照)は、**非同期関数**と呼ばれます。通常は、**非同期**という用語は、`async` 修飾子を持つすべての種類の関数の説明に使用されます。

非同期関数の仮パラメーター リストに `ref` パラメーターまたは `out` パラメーターを指定するとコンパイル時のエラーになります。

非同期メソッドの *return-type* は、`void`、またはタスク型である必要があります。タスク型は `System.Threading.Tasks.Task` および `System.Threading.Tasks.Task<T>` から構成されている型です。説明を簡潔にするために、この章ではこれらの型をそれぞれ `Task` および `Task<T>` と呼びます。タスク型を返す非同期メソッドは、タスクを返すと考えられます。

タスク型の正確な定義は実装で定義されますが、言語から見るとタスク型の状態は未完了、成功、または違反のいずれかです。違反タスクは、関連する例外を記録します。成功した `Task<T>` は型 `T` の結果を記録します。タスク型は待機可能であり、タスクを `await` 式のオペランドに指定できます(7.7.7 を参照)。

非同期関数呼び出しは、その本体で `await` 式(7.7.7)を使用して評価を中断することができます。後で再開デリゲートを使用して、中断した `await` 式の場所から評価を再開することができます。再開デリゲートは `System.Action` 型であり、呼び出されると、処理を中断した `await` 式から非同期関数呼び出しの評価が再開されます。非同期関数呼び出しの現在の呼び出し元は、関数呼び出しが中断されなかった場合は元の呼び出し元、それ以外の場合は再開デリゲートの最も新しい呼び出し元になります。

### 10.15.1 タスクを返す非同期関数の評価

タスクを返す非同期関数を呼び出すと、返されたタスクの型のインスタンスが生成されます。これを、非同期関数の戻りタスクと呼びます。タスクの初期状態は未完了です。

非同期関数本体は評価され、`await` 式に到達して中断されるか、または最後まで評価されて終了します。この時点で制御と戻りタスクは呼び出し元に返されます。

非同期関数の本体が終了すると、戻りタスクは未完了から次の状態に移ります。

- 関数本体が `return` ステートメントまたは本体の最後に到達して終了すると、結果値は戻りタスクに記録され、戻りタスクは成功状態になります。
- 関数本体が、キャッチされない例外(8.9.5 を参照)により終了すると、例外は戻りタスクに記録され、戻りタスクは違反状態になります。

### 10.15.2 void を返す非同期関数の評価

非同期関数の戻り値の型が `void` の場合、評価は前の場合とは異なり、タスクが返されないため、関数は現在のスレッドの同期コンテキストに完了と例外を伝えます。同期コンテキストの正確な定義は実装によって異なりますが、現在のスレッドが実行されている "場所" を表します。同期コンテキストは、`void` を返す非同期関数の評価が開始されたとき、正常に完了したとき、またはキャッチされない例外がスローされたときに通知を受け取ります。

これにより、コンテキストは、そのコンテキストで実行されている `void` を返す非同期関数の数を追跡し、発生する例外の反映方法を決定できます。

**Chapter** エラー! [ホーム] タブを使用して、ここに表示する文字列に Heading 1 を適用してください。 エラー! [ホーム] タブ

# 11. 構造体

構造体はクラスに似ていて、データ メンバーおよび関数メンバーを格納できるデータ構造を表します。クラスと異なるのは、構造体は値型であり、ヒープ割り当てを必要としません。構造体型の変数は構造体のデータを直接格納します。クラス型の変数はデータへの参照を格納します。後者はオブジェクトと呼ばれます。

構造体は、値セマンティクスを持つ小規模なデータ構造に特に役立ちます。構造体に適した例としては、複雑な数値、座標システム内の点、またはディクショナリ内のキー値のペアがあります。これらのデータ構造にとって重要な点は、データ メンバーが少ないこと、継承や参照 ID を使う必要がないこと、および代入により参照ではなく、値をコピーする値セマンティクスを使って、手軽に実装できることです。

4.1.4 で説明しているように、`int`、`double`、`bool` など C# により提供される単純型は実際にはすべて構造体型です。これらの定義済みの型が構造体であるのと同様に、構造体と演算子のオーバーロードを使って、C# 言語の "プリミティブ" な新規の型を実装できます。型についての 2 つの例をこの章の最後に示します (11.4 を参照)。

## 11.1 構造体の宣言

*struct-declaration* は、新しい構造体を宣言する *type-declaration* (9.6 を参照) です。

```
struct-declaration:
    attributesopt struct-modifiersopt partialopt struct identifier type-parameter-listopt
        struct-interfacesopt type-parameter-constraints-clausesopt struct-body ;opt
```

*struct-declaration* は、省略可能な *attributes* の集合 (17 を参照)、省略可能な *struct-modifiers* の集合 (11.1.1 を参照)、省略可能な *partial* 修飾子、キーワード `struct` と構造体を指定する *identifier*、省略可能な *type-parameter-list* 指定 (10.1.3 を参照)、省略可能な *struct-interfaces* 指定 (11.1.2 を参照)、省略可能な *type-parameters-constraints-clauses* 指定 (10.1.5 を参照)、*struct-body* (11.1.4 を参照)、および省略可能なセミコロンから構成されます。

### 11.1.1 構造体修飾子

*struct-declaration* には、オプションとして一連の構造体修飾子を含めることができます。

```
struct-modifiers:
    struct-modifier
    struct-modifiers struct-modifier

struct-modifier:
    new
    public
    protected
    internal
    private
```

1 つの構造体宣言内で同じ修飾子を複数回使用すると、コンパイルエラーになります。

構造体宣言の修飾子の意味は、クラス宣言 (10.1 を参照) の修飾子の意味と同じです。

### 11.1.2 Partial 修飾子

`partial` 修飾子は、その *struct-declaration* が部分型宣言であることを示します。外側の名前空間または型宣言内に同じ名前を持つ複数の部分構造体宣言がある場合、10.2 で説明されている規則に従って、合わせて 1 つの構造体宣言が形成されます。

### 11.1.3 構造体インターフェイス

構造体の宣言には、*struct-interfaces* 指定を含めることができます。この場合、構造体は指定のインターフェイス型を直接実装すると言います。

```
struct-interfaces:  
  : interface-type-list
```

インターフェイスの実装の詳細については、13.4 を参照してください。

### 11.1.4 構造体の本体

構造体の *struct-body* は、構造体のメンバーを定義します。

```
struct-body:  
  { struct-member-declarationsopt }
```

## 11.2 構造体のメンバー

構造体のメンバーは、*struct-member-declaration* によって導入されたメンバーと、`System.ValueType` 型から継承されたメンバーで構成されます。

```
struct-member-declarations:  
  struct-member-declaration  
  struct-member-declarations struct-member-declaration  
  
struct-member-declaration:  
  constant-declaration  
  field-declaration  
  method-declaration  
  property-declaration  
  event-declaration  
  indexer-declaration  
  operator-declaration  
  constructor-declaration  
  static-constructor-declaration  
  type-declaration
```

11.3 で示されている相違点を除いて、10.3 から 10.14 で説明しているクラス メンバーの説明は、構造体メンバーにも当てはまります。

### 11.3 クラスと構造体の違い

構造体とクラスにはいくつかの重要な違いがあります。

- 構造体は値型です (11.3.1 を参照)。
- すべての構造型は暗黙的に `System.ValueType` クラスから継承します (11.3.2 を参照)。
- 構造型の変数に値を代入すると、代入した値のコピーが作成されます (11.3.3 を参照)。

- 構造体の既定値は、すべての値型フィールドをそれぞれの既定値に、すべての参照型フィールドを `null` に設定することで生成される値です (11.3.4 を参照)。
- 構造体の型と `object` との変換には、ボックス化演算およびボックス化解除演算を使用します (11.3.5 を参照)。
- `this` の意味が構造体では異なります (7.6.7 を参照)。
- 構造体のインスタンス フィールドの宣言には、変数初期化子を含めることはできません (11.3.7 を参照)。
- 構造体では、パラメーターなしのインスタンス コンストラクターを宣言できません (11.3.8 を参照)。
- 構造体でデストラクターを宣言することはできません (11.3.9 を参照)。

### 11.3.1 値のセマンティクス

構造体は値型 (4.1 を参照) であり、値のセマンティクスを持つと言います。一方、クラスは参照型 (4.2 を参照) であり、参照のセマンティクスを持つと言います。

構造体型の変数は構造体のデータを直接格納します。クラス型の変数はデータへの参照を格納します。後者はオブジェクトと呼ばれます。構造体 `B` が型 `A` のインスタンス フィールドを格納し、`A` が構造体型のとき、`A` が `B` または `B` から構築された型である場合は、コンパイル エラーになります。`X` が型 `Y` のインスタンス フィールドを格納する場合、構造体 `X` は構造体 `Y` に **直接依存** します。この定義により、構造体が依存する構造体の完全な集合は、"**直接依存**" 関係の推移的閉包となります。次に例を示します。

```
struct Node
{
    int data;
    Node next; // error, Node directly depends on itself
}
```

この例は、`Node` がその所有する型のインスタンス フィールドを格納するため、エラーになります。別の例を示します。

```
struct A { B b; }
struct B { C c; }
struct C { A a; }
```

この例は、型 `A`、`B`、および `C` が相互に依存するため、エラーになります。

クラスでは、2 つの変数が同じオブジェクトを参照できるため、一方の変数に対する演算が他方の変数によって参照されるオブジェクトに影響を与えることができます。構造体では、各変数が独自のデータ コピーを保持し (`ref` パラメーターと `out` パラメーターの変数を除く)、一方の変数に対する演算が他方の変数に影響を与えることはできません。さらに、構造体は参照型ではないので、構造体型の値を `null` にすることはできません。

宣言を次のように行うとします。

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

```

public Point(int x, int y) {
    this.x = x;
    this.y = y;
}
}

```

ここで、次のコードがあります。

```

Point a = new Point(10, 10);
Point b = a;
a.x = 100;
System.Console.WriteLine(b.x);

```

これは値 10 を出力します。a を b に代入すると、値のコピーが作成されます。a.x への代入によって b が影響を受けることはありません。Point がクラスとして宣言されたとすると、a と b は同じオブジェクトを参照するため、出力は 100 になります。

### 11.3.2 継承

すべての構造体型は、暗黙的に System.ValueType クラスから継承し、このクラスは object クラスから継承します。構造体の宣言は、実装されるインターフェイスのリストを指定しますが、構造体の宣言で基底クラスを指定することはできません。

構造体型は抽象にはならず、常に暗黙的にシールされています。このため、abstract 修飾子と sealed 修飾子は、構造体の宣言の中では使用できません。

構造体では継承がサポートされないので、構造体メンバーの宣言されたアクセシビリティは、protected または protected internal にできません。

構造体内の関数メンバーを abstract または virtual に設定することはできません。override 修飾子でオーバーライドできるのは System.ValueType から継承されたメソッドだけです。

### 11.3.3 代入

構造体型の変数に値を代入すると、代入した値のコピーが作成されます。これは、クラス型の変数への代入とは異なります。クラス型変数への代入では参照がコピーされ、参照によって識別されるオブジェクトはコピーされません。

代入と同様に、構造体が値パラメーターとして渡される、または関数メンバーの結果として返される場合は、構造体のコピーが作成されます。構造体は、ref パラメーターまたは out パラメーターを使って、関数メンバーに参照で渡すことができます。

構造体のプロパティまたはインデクサーが代入の代入対象である場合は、そのプロパティまたはインデクサーのアクセスに関連付けられたインスタンス式は、変数として分類する必要があります。インスタンス式が値に分類される場合は、コンパイルエラーが発生します。これは 7.17.1 で詳細に説明しています。

### 11.3.4 既定値

5.2 で説明しているように、いくつかの種類の変数は、作成されるとき自動的に既定値に初期化されます。クラス型およびその他の参照型の変数の場合、この既定値は null です。構造体は null に設定できない値型なので、構造体の既定値は、すべての値型フィールドをその既定値に、およびすべての参照型フィールドを null に設定することで生成される値です。

上で宣言された Point 構造体の場合は次のようになります。

```
Point[] a = new Point[100];
```

これは、配列内の各 `Point` が、`x` フィールドと `y` フィールドを 0 に設定することで生成される値に初期化されます。

構造体の既定値は、構造体の既定のコンストラクター (4.1.2 を参照) によって返される値に対応します。クラスとは異なり、構造体はパラメーターなしのインスタンス コンストラクターを宣言できません。代わりに、各構造体はパラメーターなしのインスタンス コンストラクターを暗黙的に保有しています。このコンストラクターは、すべての値型フィールドをその既定値に、およびすべての参照型フィールドを `null` に設定することで生成される値を常に返します。

構造体は、既定の初期化状態を有効な状態と見なすように設計する必要があります。次に例を示します。

```
using System;
struct KeyValuePair
{
    string key;
    string value;
    public KeyValuePair(string key, string value) {
        if (key == null || value == null) throw new ArgumentException();
        this.key = key;
        this.value = value;
    }
}
```

ユーザー定義のインスタンス コンストラクターは、明示的に呼び出される場合だけ NULL 値が設定されません。`KeyValuePair` 変数に対して既定値の初期化が行われる場合は、`key` フィールドと `value` フィールドが `null` になり、構造体はこの状態を処理するための準備が必要になります。

### 11.3.5 ボックス化とボックス化解除

クラス型の値は、コンパイル時に参照を別の型として扱うだけで、型 `object`、またはそのクラスによって実装されるインターフェイス型に変換できます。同様に、型 `object` の値またはインターフェイス型の値は、参照を変更しなくてもクラス型に再変換できます。ただし、この場合は実行時の型チェックが当然必要になります。

構造体は参照型ではないので、これらの処理は構造体型によって異なる方法で実装されます。構造体型の値を型 `object` または構造体によって実装されるインターフェイス型に変換する場合は、ボックス化処理が発生します。同様に、型 `object` の値またはインターフェイス型の値を構造体型に再変換する場合は、ボックス化処理が発生します。クラス型に対する同じ処理と主に異なる点は、ボックス化により、ボックス化されたインスタンス内に構造体の値がコピーされること、およびボックス化解除により、ボックス化されたインスタンスから構造体の値がコピーされることです。したがって、ボックス化またはボックス化解除の処理後は、ボックス化解除された構造体への変更がボックス化された構造体に反映されません。

構造体型が `System.Object` (`Equals`、`GetHashCode`、`ToString` など) から継承した仮想メソッドをオーバーライドする場合は、構造体型のインスタンスを通じて仮想メソッドを呼び出してもボックス化は発生しません。これは、構造体が型パラメーターとして使用されている場合も同様であり、呼び出しへは、型パラメーター型のインスタンスを通じて行われます。次に例を示します。

```
using System;
struct Counter
{
    int value;
```

```

        public override string ToString() {
            value++;
            return value.ToString();
        }
    }

    class Program
    {
        static void Test<T>() where T: new() {
            T x = new T();
            Console.WriteLine(x.ToString());
            Console.WriteLine(x.ToString());
            Console.WriteLine(x.ToString());
        }

        static void Main() {
            Test<Counter>();
        }
    }
}

```

このプログラムの出力は次のようにになります。

```

1
2
3

```

`ToString` が本来とは異なる機能を持つ点に問題はありますが、この例は `x.ToString()` の 3 回の呼び出しでボックス化が発生しないことを示しています。

同様に、制約された型パラメーターのメンバーにアクセスするときに、暗黙的にボックス化が発生することはありません。たとえば、インターフェイス `ICounter` に、値を変更するためのメソッド `Increment` が含まれているとします。`ICounter` を制約として使用する場合、`Increment` メソッドの実装は `Increment` を呼び出した変数への参照付きで呼び出され、ボックス化されたコピーではありません。

```

using System;
interface ICounter
{
    void Increment();
}

struct Counter: ICounter
{
    int value;

    public override string ToString() {
        return value.ToString();
    }

    void ICounter.Increment() {
        value++;
    }
}

class Program
{
    static void Test<T>() where T: ICounter, new() {
        T x = new T();
        Console.WriteLine(x);
        x.Increment();           // Modify x
        Console.WriteLine(x);
        ((ICounter)x).Increment(); // Modify boxed copy of x
        Console.WriteLine(x);
    }
}

```

```
static void Main() {
    Test<Counter>();
}
```

`Increment` の最初の呼び出しによって変数 `x` の値が変更されます。これは、`Increment` の 2 回目の呼び出しとは異なります。2 回目の呼び出しでは、`x` のボックス化されたコピーの値が変更されます。したがって、このプログラムの出力は次のようにになります。

```
0
1
1
```

ボックス化とボックス化解除の詳細については、4.3 を参照してください。

### 11.3.6 this の概念

クラスのインスタンス コンストラクター、またはインスタンス関数メンバー内では、`this` は値として分類されます。このため、`this` を使うと、関数メンバーが呼び出されたインスタンスを参照できますが、クラスの関数メンバー内に `this` を割り当てるることはできません。

構造体のインスタンス コンストラクター内では、`this` は構造型の `out` パラメーターに対応し、構造体のインスタンス関数メンバー内では、`this` は構造型の `ref` パラメーターに対応します。どちらの場合も `this` は変数として分類され、`this` に代入するか、`ref` パラメーターまたは `out` パラメーターとして `this` を渡すことで、関数メンバーが呼び出された構造体全体を変更できます。

### 11.3.7 フィールド初期化子

11.3.4 で説明しているように、構造体の既定値は、すべての値型フィールドをそれぞれの既定値に、すべての参照型フィールドを `null` に設定することで生成される値で構成されています。このため、構造体でインスタンス フィールドの宣言を使って変数初期化子を含めることはできません。この制限はインスタンス フィールドにだけ適用されます。構造体の静的フィールドでは変数初期化子を含めることができます。

次の例を参照してください。

```
struct Point
{
    public int x = 1; // Error, initializer not permitted
    public int y = 1; // Error, initializer not permitted
}
```

これは、インスタンス フィールドの宣言に変数初期化子が含まれているためです。

### 11.3.8 コンストラクター

クラスとは異なり、構造体はパラメーターなしのインスタンス コンストラクターを宣言できません。代わりに、各構造体はパラメーターなしのインスタンス コンストラクターを暗黙的に保有しています。このコンストラクターは、すべての値型フィールドをその既定値に、およびすべての参照型フィールドを `null` に設定することで生成される値を常に返します (4.1.2 を参照)。構造体は、パラメーターを持つインスタンス コンストラクターを宣言できます。次に例を示します。

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

```
public Point(int x, int y) {
    this.x = x;
    this.y = y;
}
```

このように宣言するものとします。

```
Point p1 = new Point();
Point p2 = new Point(0, 0);
```

どちらのステートメントでも、`x` と `y` がゼロに初期化された `Point` が作成されます。

構造体のインスタンス コンストラクターには、`base(...)` 形式のコンストラクター初期化子を含めることはできません。

構造体のインスタンス コンストラクターがコンストラクター初期化子を指定していない場合、`this` 変数は構造体型の `out` パラメーターに対応しています。このため、`out` パラメーターと同様に、`this` はコンストラクターが戻るすべての場所で確実に代入されている必要があります (5.3 を参照)。構造体のインスタンス コンストラクターがコンストラクター初期化子を指定している場合、`this` 変数は構造体型の `ref` パラメーターに対応しています。このため、`ref` パラメーターと同様に、`this` はコンストラクター本体へのエントリで確実に代入されていると見なされます。インスタンス コンストラクターの実装例を次に示します。

```
struct Point
{
    int x, y;
    public int x {
        set { x = value; }
    }
    public int y {
        set { y = value; }
    }
    public Point(int x, int y) {
        X = x;           // error, this is not yet definitely assigned
        Y = y;           // error, this is not yet definitely assigned
    }
}
```

構築中の構造体のすべてのフィールドが確実に代入された状態になるまで、`X` プロパティと `Y` プロパティの `set` アクセサーを含むインスタンスのメンバー関数を呼び出すことはできません。ただし、`Point` が構造体ではなくクラスであった場合は、インスタンス コンストラクターの実装が許可されています。

### 11.3.9 デストラクター

構造体でデストラクターを宣言することはできません。

### 11.3.10 静的コンストラクター

構造体の静的コンストラクターは、クラスの規則のほとんどすべてに従います。構造体型の静的コンストラクターは、次のイベントがアプリケーション ドメインで最初に発生したときに実行されます。

- 構造体型の静的メンバーが参照される。
- 明示的に宣言された構造体型のコンストラクターが呼び出される。

構造体型の既定値の型 (11.3.4 を参照) が作成されても、静的コンストラクターは発生しません。この例として、配列内の要素の初期値があります。

## 11.4 構造体の例

次に、`struct` 型を使用して型を作成する 2 つの重要な例を示します。作成された型は、セマンティクスが変更されていますが、言語の定義済みの型と同様に使用できます。

### 11.4.1 データベースの整数型

下の `DBInt` 構造体は、`int` 型の値の完全な集合を表すことができる整数型を実装します。さらに、不明な値を示す状態を表すこともできます。これらの特性を備えた型は通常、データベースで使用されます。

```
using System;
public struct DBInt
{
    // The Null member represents an unknown DBInt value.
    public static readonly DBInt Null = new DBInt();
    // When the defined field is true, this DBInt represents a known value
    // which is stored in the value field. When the defined field is false,
    // this DBInt represents an unknown value, and the value field is 0.
    int value;
    bool defined;
    // Private instance constructor. Creates a DBInt with a known value.
    DBInt(int value) {
        this.value = value;
        this.defined = true;
    }
    // The IsNull property is true if this DBInt represents an unknown value.
    public bool IsNull { get { return !defined; } }
    // The value property is the known value of this DBInt, or 0 if this
    // DBInt represents an unknown value.
    public int value { get { return value; } }
    // Implicit conversion from int to DBInt.
    public static implicit operator DBInt(int x) {
        return new DBInt(x);
    }
    // Explicit conversion from DBInt to int. Throws an exception if the
    // given DBInt represents an unknown value.
    public static explicit operator int(DBInt x) {
        if (!x.defined) throw new InvalidOperationException();
        return x.value;
    }
    public static DBInt operator +(DBInt x) {
        return x;
    }
    public static DBInt operator -(DBInt x) {
        return x.defined ? -x.value : Null;
    }
}
```

```

public static DBInt operator +(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value + y.value: Null;
}

public static DBInt operator -(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value - y.value: Null;
}

public static DBInt operator *(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value * y.value: Null;
}

public static DBInt operator /(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value / y.value: Null;
}

public static DBInt operator %(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value % y.value: Null;
}

public static DBBool operator ==(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value == y.value: DBBool.Null;
}

public static DBBool operator !=(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value != y.value: DBBool.Null;
}

public static DBBool operator >(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value > y.value: DBBool.Null;
}

public static DBBool operator <(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value < y.value: DBBool.Null;
}

public static DBBool operator >=(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value >= y.value: DBBool.Null;
}

public static DBBool operator <=(DBInt x, DBInt y) {
    return x.defined && y.defined? x.value <= y.value: DBBool.Null;
}

public override bool Equals(object obj) {
    if (!(obj is DBInt)) return false;
    DBInt x = (DBInt)obj;
    return value == x.value && defined == x.defined;
}

public override int GetHashCode() {
    return value;
}

public override string ToString() {
    return defined? value.ToString(): "DBInt.Null";
}
}

```

## 11.4.2 データベースのブール型

下の DBBool 構造体は、3 つの値で構成される論理型を実装します。この型に指定できる値は、`DBBool.True`、`DBBool.False`、および `DBBool.Null` です。`Null` メンバーは、不明な値を示します。この構造体のように 3 つの値で構成される論理型は通常、データベースで使用されます。

```
using System;
```

## C# LANGUAGE SPECIFICATION

```
public struct DBBool
{
    // The three possible DBBool values.
    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
    public static readonly DBBool True = new DBBool(1);

    // Private field that stores -1, 0, 1 for False, Null, True.
    sbyte value;

    // Private instance constructor. The value parameter must be -1, 0, or 1.
    DBBool(int value) {
        this.value = (sbyte)value;
    }

    // Properties to examine the value of a DBBool. Return true if this
    // DBBool has the given value, false otherwise.
    public bool IsNull { get { return value == 0; } }
    public bool IsFalse { get { return value < 0; } }
    public bool IsTrue { get { return value > 0; } }

    // Implicit conversion from bool to DBBool. Maps true to DBBool.True and
    // false to DBBool.False.
    public static implicit operator DBBool(bool x) {
        return x? True: False;
    }

    // Explicit conversion from DBBool to bool. Throws an exception if the
    // given DBBool is Null, otherwise returns true or false.
    public static explicit operator bool(DBBool x) {
        if (x.value == 0) throw new InvalidOperationException();
        return x.value > 0;
    }

    // Equality operator. Returns Null if either operand is Null, otherwise
    // returns True or False.
    public static DBBool operator ==(DBBool x, DBBool y) {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value == y.value? True: False;
    }

    // Inequality operator. Returns Null if either operand is Null, otherwise
    // returns True or False.
    public static DBBool operator !=(DBBool x, DBBool y) {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value != y.value? True: False;
    }

    // Logical negation operator. Returns True if the operand is False, Null
    // if the operand is Null, or False if the operand is True.
    public static DBBool operator !(DBBool x) {
        return new DBBool(-x.value);
    }

    // Logical AND operator. Returns False if either operand is False,
    // otherwise Null if either operand is Null, otherwise True.
    public static DBBool operator &(DBBool x, DBBool y) {
        return new DBBool(x.value < y.value? x.value: y.value);
    }
```

```
// Logical OR operator. Returns True if either operand is True, otherwise
// Null if either operand is Null, otherwise False.
public static DBBool operator |(DBBool x, DBBool y) {
    return new DBBool(x.value > y.value? x.value: y.value);
}

// Definitely true operator. Returns true if the operand is True, false
// otherwise.
public static bool operator true(DBBool x) {
    return x.value > 0;
}

// Definitely false operator. Returns true if the operand is False, false
// otherwise.
public static bool operator false(DBBool x) {
    return x.value < 0;
}

public override bool Equals(object obj) {
    if (!(obj is DBBool)) return false;
    return value == ((DBBool)obj).value;
}

public override int GetHashCode() {
    return value;
}

public override string ToString() {
    if (value > 0) return "DBBool.True";
    if (value < 0) return "DBBool.False";
    return "DBBool.Null";
}
```



# 12. 配列

配列とは、算出されたインデックスを使用してアクセスされる複数の変数を含むデータ構造です。配列に含まれる変数は、配列の要素とも呼ばれ、すべて同じ型です。この型を配列の要素型と呼びます。

配列には、配列の各要素に関連付けられるインデックス数を決定するランクがあります。配列のランクは、配列の次元とも呼ばれます。ランクが 1 の配列は **1 次元配列** と呼ばれます。ランクが 1 次元配列を超える配列は **多次元配列** と呼ばれます。特定のサイズの多次元配列は、2 次元配列、3 次元配列などのように、そのサイズで呼ばれるのが一般的です。

配列の各次元には、関連付けられた長さがあります。この長さは 0 以上の整数になります。次元の長さは配列の型には含まれませんが、実行時に配列型のインスタンスが作成されるときに確立されます。次元の長さは、その次元のインデックスの有効範囲を決定します。長さ  $N$  の次元の場合、インデックスの有効範囲は  $0 \sim N - 1$  です。配列内の要素の合計数は、その配列の各次元の長さの積になります。配列に長さが 0 の次元が存在する場合、配列は空であることになります。

配列の要素型は、配列型を含めて、任意の型に設定できます。

## 12.1 配列型

配列型は、*non-array-type* の後に 1 つ以上の *rank-specifier* が続く形式で記述されます。

```

array-type:
    non-array-type rank-specifiers

non-array-type:
    type

rank-specifiers:
    rank-specifier
    rank-specifiers rank-specifier

rank-specifier:
    [ dim-separatorsopt ]

dim-separators:
    ,
    dim-separators ,

```

*non-array-type* は、それ自体が *array-type* ではない *type* です。

配列型のランクは、*array-type* 内で最も左の *rank-specifier* によって指定されます。*rank-specifier* は、その配列が、*rank-specifier* 内の "," トークンの数に 1 を足したランクを持つ配列であることを示します。

配列型の要素型は、最も左の *rank-specifier* を削除した型になります。

- $T[R]$  形式の配列型は、ランク  $R$  と非配列要素型  $T$  が指定された配列です。
- $T[R][R_1] \dots [R_N]$  形式の配列型は、ランク  $R$  と要素型  $T[R_1] \dots [R_N]$  が指定された配列です。

*rank-specifier* は、実際には、最後の非配列要素型の前まで、左から右に読み取られます。

`int[][][,]` は、`int` 型の 2 次元配列の 3 次元配列のさらに 1 次元配列です。

実行時には、配列型の値は `null`、またはその配列型のインスタンスへの参照になります。

### 12.1.1 System.Array 型

`System.Array` 型は、すべての配列型の抽象基本型です。任意の配列型から `System.Array` への暗黙の参照変換 (6.1.6 を参照) が存在し、`System.Array` から任意の配列型への明示的な参照変換 (6.2.4 を参照) も存在します。`System.Array` 自体は *array-type* ではありません。`System.Array` は、すべての *value-type* の派生元である *class-type* です。

実行時に、`System.Array` 型の値は、`null` または任意の配列型のインスタンスへの参照になります。

### 12.1.2 配列とジェネリック `IList` インターフェイス

1 次元配列 `T[]` は、インターフェイス `System.Collections.Generic.IList<T>` (短い形式は `IList<T>`) とその基本インターフェイスを実装します。それに応じて、`T[]` から `IList<T>` およびその基本インターフェイスへの暗黙の変換が存在します。また、`S[]` から `T` への暗黙の参照変換が存在する場合、`S[]` は `IList<T>` を実装し、`S[]` から `IList<T>` およびその基本インターフェイスへの暗黙の参照変換が存在します (6.1.6 を参照)。`S` から `T` への明示的な参照変換が存在する場合は、`S[]` から `IList<T>` およびその基本インターフェイスへの明示的な参照変換が存在します (6.2.4 を参照)。次に例を示します。

```
using System.Collections.Generic;
class Test
{
    static void Main()
    {
        string[] sa = new string[5];
        object[] oa1 = new object[5];
        object[] oa2 = sa;

        IList<string> lst1 = sa; // Ok
        IList<string> lst2 = oa1; // Error, cast needed
        IList<object> lst3 = sa; // Ok
        IList<object> lst4 = oa1; // Ok

        IList<string> lst5 = (IList<string>)oa1; // Exception
        IList<string> lst6 = (IList<string>)oa2; // Ok
    }
}
```

`lst2 = oa1` という代入を実行すると、`object[]` から `IList<string>` への変換は暗黙の変換ではなく明示的な変換であるため、コンパイルエラーになります。`(IList<string>)oa1` をキャストすると、`oa1` は `object[]` ではなく `string[]` を参照するため、実行時に例外がスローされます。ただし、`(IList<string>)oa2` をキャストすると、`oa2` は `string[]` を参照するため、実行時に例外がスローされません。

`S[]` から `IList<T>` への暗黙または明示的な参照変換が存在する場合は、`IList<T>` およびその基本インターフェイスから `S[]` への明示的な参照変換が存在します (6.2.4 を参照)。

配列型 `S[]` が `IList<T>` を実装すると、実装されたインターフェイスのメンバーによっては例外がスローされる場合があります。インターフェイスの実装の詳細な動作については、ここでは取り上げません。

## 12.2 配列の作成

配列インスタンスは *array-creation-expression* (7.6.10.4 を参照) により、または *array-initializer* (12.6 を参照) を含むフィールド宣言またはローカル変数宣言により作成されます。

配列インスタンスが作成されるときに、各次元のランクと長さが確立され、インスタンスの有効期間を通じて変わりません。つまり、既存の配列インスタンスのランクも、その次元の大きさも変更できません。

配列インスタンスは常に配列型です。 `System.Array` 型は、インスタンス化できない抽象型です。

*array-creation-expression* によって作成された配列の要素は、常に既定値に初期化されます (5.2 を参照)。

## 12.3 配列要素へのアクセス

配列要素にアクセスするには、 $A[I_1, I_2, \dots, I_N]$  形式の *element-access* 式 (7.6.6.1 を参照) を使用します。  $A$  は配列型の式であり、各  $I_x$  は `int`、`uint`、`long`、`ulong`、またはこれらの型の 1 つ以上に暗黙的に変換できる型の式です。配列要素へのアクセスの結果は、変数、つまりインデックスによって選択された配列要素になります。

配列の要素は、`foreach` ステートメントを使用して列挙できます (8.8.4 を参照)。

## 12.4 配列のメンバー

すべての配列型は、`System.Array` 型によって宣言されたメンバーを継承します。

## 12.5 配列の共変性

2 つの *reference-type*  $A$  と  $B$  について、 $A$  から  $B$  への暗黙の参照変換 (6.1.6 を参照) または明示的な参照変換 (6.2.4 を参照) が存在する場合、配列型  $A[R]$  から配列型  $B[R]$  への同じ変換が存在します。  $R$  は指定された任意の *rank-specifier* (いずれの配列型も同じ) です。この関係は配列の共変性と呼ばれます。  $B$  から  $A$  への暗黙の参照変換が存在する場合の配列の共変性は、配列型  $A[R]$  の値が、実際には配列型  $B[R]$  のインスタンスへの参照である可能性があることを意味します。

配列の共変性のため、参照型配列の要素への代入ではランタイム チェックが行われ、実際に許可されている型の値が配列要素に代入されるようにします (7.17.1 を参照)。次に例を示します。

```
class Test
{
    static void Fill(object[] array, int index, int count, object value) {
        for (int i = index; i < index + count; i++) array[i] = value;
    }

    static void Main() {
        string[] strings = new string[100];
        Fill(strings, 0, 100, "Undefined");
        Fill(strings, 0, 10, null);
        Fill(strings, 90, 10, 0);
    }
}
```

`Fill` メソッド内の `array[i]` への代入には、ランタイム チェックが暗黙的に含まれます。このため、`value` によって参照されるオブジェクトが、`null` または `array` の実際の要素型と互換性のあるインスタンスであることが保証されます。`Main` では、`Fill` の 2 回目までの呼び出しは成功します。しかし、3 回目の呼び出しへは、`array[i]` への最初の代入を実行するときに

`System.ArrayTypeMismatchException` がスローされます。この例外が発生するのは、ボックス化された `int` を `string` 配列に格納できないためです。

配列の共変性は、*value-type* の配列では起こりません。たとえば、`int[]` を `object[]` として処理できる変換は存在しません。

## 12.6 配列初期化子

配列初期化子は、フィールド宣言 (10.5 を参照)、ローカル変数宣言 (8.5.1 を参照)、および配列作成式 (7.6.10.4 を参照) で指定できます。

```

array-initializer:
{ variable-initializer-listopt }
{ variable-initializer-list , }

variable-initializer-list:
variable-initializer
variable-initializer-list , variable-initializer

variable-initializer:
expression
array-initializer

```

配列初期化子は、"{" トークンと "}" トークン内の "," で区切った一連の変数初期化子から構成されます。各変数初期化子は、式、または(多次元配列の場合は)入れ子になった配列初期化子です。

配列初期化子が使用されるコンテキストによって、初期化される配列の型が異なります。配列作成式では、配列型が初期化子の直前に置かれます。または、配列初期化子内の式から導かれます。フィールドや変数の宣言では、配列型は、宣言するフィールドまたは変数の型になります。配列初期化子をフィールドや変数の宣言で使用する場合は、次のようにになります。

```
int[] a = {0, 2, 4, 6, 8};
```

これは、次の配列作成式を簡略に表したものです。

```
int[] a = new int[] {0, 2, 4, 6, 8};
```

1 次元配列の場合の配列初期化子は、配列の要素型と互換性のある代入である、一連の式で構成する必要があります。各式は、インデックス 0 の要素から昇順に、配列要素を初期化します。配列初期化子内の式の数によって、作成される配列インスタンスの長さが決まります。たとえば、上の配列初期化子は、長さが 5 の `int[]` インスタンスを作成し、そのインスタンスを次の値で初期化します。

```
a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;
```

多次元配列の場合、配列初期化子は、配列の次元と同じ入れ子レベルを持つ必要があります。最も外側の入れ子レベルは左端の次元に対応し、最も内側の入れ子レベルは右端の次元に対応します。配列の各次元の長さは、配列初期化子内の対応する入れ子レベルに存在する要素の数によって決まります。入れ子になった配列初期化子の場合、それぞれの要素数は、同じレベルの他の配列初期化子と同じ数である必要があります。次に例を示します。

```
int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
```

この例では、次に示すように、左端の次元の長さが 5 で右端の次元の長さが 2 の 2 次元配列が作成されます。

```
int[,] b = new int[5, 2];
```

そして、次の値で配列初期化子が初期化されます。

```
b[0, 0] = 0; b[0, 1] = 1;  
b[1, 0] = 2; b[1, 1] = 3;  
b[2, 0] = 4; b[2, 1] = 5;  
b[3, 0] = 6; b[3, 1] = 7;  
b[4, 0] = 8; b[4, 1] = 9;
```

右端以外の次元の長さが 0 に指定されている場合、以降の次元も長さ 0 であると見なされます。次に例を示します。

```
int[,] c = {};
```

この例では、次に示すように、左端と右端の次元の長さがどちらも 0 である 2 次元配列が作成されます。

```
int[,] c = new int[0, 0];
```

配列作成式に明示的な次元の長さと配列初期化子が含まれる場合、その長さは定数式である必要があります。また、それぞれの入れ子レベルにある要素の数は、対応する次元の長さと一致する必要があります。次にいくつかの例を示します。

```
int i = 3;  
int[] x = new int[3] {0, 1, 2}; // OK  
int[] y = new int[i] {0, 1, 2}; // Error, i not a constant  
int[] z = new int[3] {0, 1, 2, 3}; // Error, length/initializer mismatch
```

上の例の *y* の初期化子は、次元の長さを表す式が定数でないため、コンパイル エラーになります。また、*z* の初期化子は、長さと初期化子内の要素数が一致しないため、コンパイル エラーになります。



# 13. インターフェイス

インターフェイスはコントラクトを定義します。インターフェイスを実装するクラスや構造体は、そのコントラクトと一致する必要があります。インターフェイスは複数の基本インターフェイスから継承できます。また、クラスや構造体は複数のインターフェイスを実装できます。

インターフェイスには、メソッド、プロパティ、イベント、およびインデクサーを含めることができます。インターフェイス自体は、インターフェイスが定義するメンバーの実装を提供しません。インターフェイスは、そのインターフェイスを実装するクラスまたは構造体が提供するメンバーを指定するだけです。

## 13.1 インターフェイスの宣言

*interface-declaration* は、新しいインターフェイス クラスを宣言する *type-declaration* (9.6 を参照) です。

```
interface-declaration:
    attributesopt interface-modifiersopt partialopt interface
        identifier variant-type-parameter-listopt interface-baseopt
        type-parameter-constraints-clausesopt interface-body ;opt
```

*interface-declaration* は、省略可能な *attributes* の集合 (17 を参照)、省略可能な *interface-modifiers* の集合 (13.1.1 を参照)、省略可能な *partial* 修飾子、キーワード **interface** とインターフェイスを指定する *identifier*、省略可能な *variant-type-parameter-list* 指定 (13.1.3 を参照)、省略可能な *interface-base* 指定 (13.1.4 を参照)、省略可能な *type-parameter-constraints-clauses* 指定 (10.1.5 を参照)、*interface-body* (13.1.5 を参照)、および省略可能なセミコロンから構成されます。

### 13.1.1 インターフェイス修飾子

*interface-declaration* には、一連のインターフェイス修飾子を含めることもできます。

```
interface-modifiers:
    interface-modifier
    interface-modifiers interface-modifier

interface-modifier:
    new
    public
    protected
    internal
    private
```

1 つのインターフェイス宣言内で同じ修飾子を複数回使用すると、コンパイル エラーになります。

**new** 修飾子を使用できるのは、クラス内で定義されたインターフェイスだけです。この修飾子は、インターフェイスが継承されたメンバーと同じ名前で隠ぺいするように指定します。詳細については、10.3.4 を参照してください。

**public**、**protected**、**internal**、および **private** の各修飾子は、インターフェイスのアクセシビリティを制御します。インターフェイスの宣言が行われるコンテキストによっては、これらの修飾子の一部は使用できません (3.5.1 を参照)。

### 13.1.2 Partial 修飾子

`partial` 修飾子は、その *interface-declaration* が部分型宣言であることを示します。外側の名前空間または型宣言内に同じ名前を持つ複数の部分インターフェイス宣言がある場合、10.2 で説明されている規則に従って、合わせて 1 つのインターフェイス宣言が形成されます。

### 13.1.3 バリアント型パラメーター リスト

バリアント型パラメーター リストを使用できるのは、インターフェイス型とデリゲート型だけです。通常の *type-parameter-lists* との違いは、それぞれの型パラメーターに省略可能な *variance-annotation* があることです。

```

variant-type-parameter-list:
< variant-type-parameters >

variant-type-parameters:
attributesopt variance-annotationopt type-parameter
variant-type-parameters , attributesopt variance-annotationopt type-parameter

variance-annotation:
in
out

```

変性の注釈が `out` の場合、型パラメーターは "共変" であると言います。変性の注釈が `in` の場合、型パラメーターは "反変" であると言います。変性の注釈がない場合、型パラメーターは "不变" であると言います。

次に例を示します。

```

interface C<out X, in Y, Z>
{
    X M(Y y);
    Z P { get; set; }
}

```

`X` は共変、`Y` は反変、`Z` は不变です。

#### 13.1.3.1 変性の安全性

型の型パラメーター リストに変性の注釈があると、型宣言内で型を使用できる場所が制限されます。

次のいずれかに該当する場合、型 `T` は "出力アンセーフ" です。

- `T` が反変の型パラメーターである場合
- `T` が配列型で、出力アンセーフな要素型を持つ場合
- `T` が、ジェネリック型 `S<X1, … Xk>` から構成されるインターフェイス型またはデリゲート型 `S<A1, … Ak>` であり、少なくとも 1 つの `Ai` について次のいずれかの条件が満たされる場合
  - `Xi` が共変または不变であり、`Ai` が出力アンセーフである。
  - `Xi` が反変または不变であり、`Ai` が入力セーフである。

次のいずれかに該当する場合、型 `T` は "入力アンセーフ" です。

- `T` が共変の型パラメーターである場合
- `T` が配列型で、入力アンセーフな要素型がある場合

- $T$  が、ジェネリック型  $S<X_1, \dots, X_k>$  から構成されるインターフェイス型またはデリゲート型  $S<A_1, \dots, A_k>$  であり、少なくとも 1 つの  $A_i$  について次のいずれかの条件が満たされる場合
  - $X_i$  が共変または不変であり、 $A_i$  が入力アンセーフである。
  - $X_i$  が反変または不変であり、 $A_i$  が出力アンセーフである。

出力アンセーフ型は出力位置では禁止され、入力アンセーフ型は入力位置では禁止されます。

型は、出力アンセーフでない場合は "出力セーフ" であり、入力アンセーフでない場合は "入力セーフ" です。

### 13.1.3.2 変性変換

変性の注釈の目的は、インターフェイス型とデリゲート型に、より厳密でない(ただしタイプセーフな)変換を提供することです。このために、暗黙変換(0を参照)と明示変換(6.2を参照)の定義では、次のように定義される変性変換の概念を使用します。

$T$  がバリアント型パラメーター  $T<X_1, \dots, X_n>$  で宣言されるインターフェイス型またはデリゲート型で、それぞれのバリアント型パラメーターについて  $X_i$  が次のいずれかの条件を満たす場合、型  $T<A_1, \dots, A_n>$  は型  $T<B_1, \dots, B_n>$  への変性変換です。

- $X_i$  が共変で、 $A_i$  から  $B_i$  への暗黙の参照変換または恒等変換が存在する
- $X_i$  が反変で、 $B_i$  から  $A_i$  への暗黙の参照変換または恒等変換が存在する
- $X_i$  が不変で、 $A_i$  から  $B_i$  への恒等変換が存在する

### 13.1.4 基本インターフェイス

1 つのインターフェイスは 0 個以上のインターフェイス型から継承できます。継承元のインターフェイスは、インターフェイスの明示的な基本インターフェイスと呼ばれます。1 つのインターフェイスが 1 つ以上の明示的な基本インターフェイスを持つ場合、そのインターフェイスの宣言では、インターフェイス識別子の後に、コロンおよびコンマ区切りの基本インターフェイス型のリストが続きます。

```
interface-base:  
  : interface-type-list
```

構築されたインターフェイス型では、明示的な基本インターフェイスは、ジェネリック型宣言の明示的な基本インターフェイス宣言を使用し、基本インターフェイス宣言の各 *type-parameter* を、構築された型の対応する *type-argument* に置き換えて形成されます。

インターフェイスの明示的な基本インターフェイスは、少なくともインターフェイス自体と同程度にアクセス可能である必要があります(3.5.4を参照)。たとえば、`public`インターフェイスの *interface-base* で `private`インターフェイスまたは `internal`インターフェイスを指定すると、コンパイル時のエラーになります。

インターフェイスが、直接または間接的に、そのインターフェイス自体から派生する場合もコンパイルエラーになります。

インターフェイスの 基本インターフェイスは、明示的な基本インターフェイス、および明示的な基本インターフェイスの基本インターフェイスです。つまり、基本インターフェイスのセットは、明示的な基本インターフェイス、その明示的な基本インターフェイスというように、明示的な基本インターフェイス

フェイスの推移的閉包になります。インターフェイスは、その基本インターフェイスのすべてのメンバーを継承します。次に例を示します。

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

`IComboBox` の基本インターフェイスは、`IControl`、`ITextBox`、および `IListBox` です。

つまり、上の例の `IComboBox` インターフェイスは、メンバー `SetText` と `SetItems` と同様に `Paint` も継承します。

インターフェイスのすべての基本インターフェイスは出力セーフである必要があります (13.1.3.1 を参照)。インターフェイスを実装するクラスや構造体も、そのインターフェイスの基本インターフェイスのすべてを暗黙的に実装します。

### 13.1.5 インターフェイスの本体

インターフェイスの *interface-body* は、インターフェイスのメンバーを定義します。

```
interface-body:
{ interface-member-declarationsopt }
```

### 13.2 インターフェイスのメンバー

インターフェイスのメンバーは、基本インターフェイスから継承したメンバーと、そのインターフェイス自体が宣言したメンバーです。

```
interface-member-declarations:
interface-member-declaration
interface-member-declarations interface-member-declaration

interface-member-declaration:
interface-method-declaration
interface-property-declaration
interface-event-declaration
interface-indexer-declaration
```

インターフェイス宣言では、0 個以上のメンバーを宣言できます。インターフェイスのメンバーは、メソッド、プロパティ、イベント、またはインデクサーである必要があります。インターフェイスは、定数、フィールド、演算子、インスタンス コンストラクター、デストラクター、または型をメンバーとして持つことはできません。また、インターフェイスには、どのような種類の静的メンバーも含めることができません。

すべてのインターフェイス メンバーは、暗黙的にパブリック アクセスになります。インターフェイス メンバーの宣言に修飾子を含めるとコンパイル エラーになります。特に、インターフェイス メン

バーの宣言には、`abstract`、`public`、`protected`、`internal`、`private`、`virtual`、`override`、または`static` の各修飾子を指定できません。

次の例を参照してください。

```
public delegate void StringListEvent(IStringList sender);
public interface IStringList
{
    void Add(string s);
    int Count { get; }
    event StringListEvent Changed;
    string this[int index] { get; set; }
}
```

この例は、可能な種類のメンバー(メソッド、プロパティ、イベント、およびインデクサー)を1つずつ持つインターフェイスを宣言します。

*interface-declaration* によって新しい宣言空間が作成され(3.3を参照)、*interface-declaration* のすぐ内側にある*interface-member-declaration*により、この宣言空間に新しいメンバーが導入されます。

*interface-member-declaration*には次の規則が適用されます。

- メソッドの名前は、同じインターフェイスで宣言されているすべてのプロパティおよびイベントの名前とは異なる名前である必要があります。さらに、メソッドのシグネチャ(3.6を参照)は同じインターフェイスで宣言されている他のすべてのメソッドのシグネチャと異なっている必要があります。同じインターフェイスで宣言される2つのメソッドに、`ref` と `out` の違いしかないシグネチャを指定することはできません。
- プロパティまたはイベントの名前は、同じインターフェイスで宣言されている他のすべてのメンバーの名前とは異なる名前である必要があります。
- インデクサーのシグネチャは、同じインターフェイスで宣言されている他のすべてのインデクサーのシグネチャとは異なるシグネチャである必要があります。

インターフェイスの継承されたメンバーは、そのインターフェイスの宣言空間には含まれません。そのため、インターフェイスは、継承されたメンバーと同じ名前やシグネチャを持つメンバーを宣言できます。このような宣言を行うと、派生インターフェイスのメンバーは、基本インターフェイスのメンバーを "隠ぺいする" と言われます。継承したメンバーを隠ぺいしてもエラーとは見なされませんが、コンパイラは警告を出します。警告を出さないようにするには、派生したインターフェイスメンバーの宣言に `new` 修飾子を含めて、派生したメンバーが基本メンバーを隠ぺいすることを示す必要があります。これは 3.7.1.2 で詳細に説明しています。

継承メンバーを隠ぺいしない宣言に `new` 修飾子が含まれる場合は、そのことを表す警告が出されます。`new` 修飾子を削除すると、この警告は出されません。

`object` クラスのメンバーは、厳密にはどのインターフェイスのメンバーでもありません(13.2を参照)。ただし、`object` クラスのメンバーは、インターフェイス型のメンバー検索を通じて利用できます(7.4を参照)。

### 13.2.1 インターフェイスのメソッド

インターフェイスのメソッドは、*interface-method-declaration*を使用して宣言します。

*interface-method-declaration:*  
 $\text{attributes}_{\text{opt}} \text{ new}_{\text{opt}} \text{ return-type } \text{identifier} \text{ type-parameter-list}$   
 $( \text{formal-parameter-list}_{\text{opt}} ) \text{ type-parameter-constraints-clauses}_{\text{opt}} ;$

インターフェイス メソッドの宣言の *attributes*、*return-type*、*identifier*、および *formal-parameter-list* は、クラスのメソッドの宣言の *attributes*、*return-type*、*identifier*、および *formal-parameter-list* と同じ意味を持ちます(10.6を参照)。インターフェイスのメソッドの宣言では、メソッド本体を指定できません。したがって、宣言は常にセミコロンで終わります。

インターフェイス メソッドの仮パラメーターの型は入力セーフである必要があり(13.1.3.1を参照)、戻り値の型は **void** または出力セーフである必要があります。さらに、メソッドの型パラメーターに対する各クラス型制約、インターフェイス型制約、および型パラメーター制約は、入力セーフである必要があります。

これらの規則によって、インターフェイスの共変または反変の使用がタイプセーフであることが確認されます。次に例を示します。

```
interface I<out T> { void M<U>() where U : T; }
```

*U* に対する型パラメーター制約として *T* を使用することは入力セーフでないため、これは無効です。

このような制限がない場合は、次のようにタイプセーフに違反する可能性があります。

```
class B {}
class D : B {}
class E : B {}
class C : I<D> { public void M<U>() {…} }
...
I<B> b = new C();
b.M<E>();
```

これは実際には *C.M<E>* の呼び出しです。ただし、その呼び出しには *D* から派生した *E* が必要であるため、この場合もタイプセーフに違反します。

### 13.2.2 インターフェイスのプロパティ

インターフェイスのプロパティは、*interface-property-declaration* を使用して宣言します。

*interface-property-declaration:*  
 $\text{attributes}_{\text{opt}} \text{ new}_{\text{opt}} \text{ type } \text{identifier} \{ \text{interface-accessors} \}$

*interface-accessors:*  
 $\text{attributes}_{\text{opt}} \text{ get } ;$   
 $\text{attributes}_{\text{opt}} \text{ set } ;$   
 $\text{attributes}_{\text{opt}} \text{ get } ; \text{ attributes}_{\text{opt}} \text{ set } ;$   
 $\text{attributes}_{\text{opt}} \text{ set } ; \text{ attributes}_{\text{opt}} \text{ get } ;$

インターフェイスのプロパティの宣言の *attributes*、*type*、および *identifier* は、クラスのプロパティの宣言の *attributes*、*type*、および *identifier* と同じ意味を持ちます(10.7を参照)。

インターフェイスのプロパティの宣言のアクセサーは、クラスのプロパティの宣言(10.7.2を参照)のアクセサーに対応します。ただし、アクセサー本体が常にセミコロンになることがあります。したがって、アクセサーは、プロパティが読み取り/書き込み、読み取り専用、または書き込み専用のいずれであるかを示すだけです。

インターフェイス プロパティの型は、*get* アクセサーがある場合には出力セーフである必要があり、*set* アクセサーがある場合は入力セーフである必要があります。

### 13.2.3 インターフェイスのイベント

インターフェイスのイベントは、*interface-event-declaration* を使用して宣言します。

```
interface-event-declaration:  
    attributesopt newopt event type identifier ;
```

インターフェイスのイベント宣言の *attributes*、*type*、および *identifier* は、クラスのイベントの宣言の *attributes*、*type*、および *identifier* と同じ意味を持ちます(10.8 を参照)。

インターフェイスイベントの型は入力セーフである必要があります。

### 13.2.4 インターフェイスのインデクサー

インターフェイスのインデクサーは、*interface-indexer-declaration* を使用して宣言します。

```
interface-indexer-declaration:  
    attributesopt newopt type this [ formal-parameter-list ] { interface-accessors }
```

インターフェイスのインデクサーの宣言の *attributes*、*type*、および *formal-parameter-list* は、クラスのインデクサーの宣言の *attributes*、*type*、および *formal-parameter-list* と同じ意味を持ちます(10.9 を参照)。

インターフェイスのインデクサーの宣言のアクセサーは、クラスのインデクサーの宣言(10.9 を参照)のアクセサーに対応します。ただし、アクセサー本体が常にセミコロンになることが異なります。したがって、アクセサーは、インデクサーが読み取り/書き込み、読み取り専用、または書き込み専用のいずれであるかを示すだけです。

インターフェイス インデクサーのすべての仮パラメーター型は、入力セーフである必要があります。また、**out** 仮パラメーターまたは **ref** 仮パラメーターは出力セーフである必要もあります。基になる実行プラットフォームの制限により、**out** パラメーターも入力セーフである必要があります。

インターフェイス インデクサーの型は、**get** アクセサーがある場合には出力セーフである必要があります、**set** アクセサーがある場合は入力セーフである必要があります。

### 13.2.5 インターフェイスのメンバーへのアクセス

インターフェイスのメンバーには、**I.M** および **I[A]** という形式の、メンバー アクセス(7.6.4 を参照)式およびインデクサー アクセス(7.6.6.2 を参照)式を使用してアクセスします。**I** はインターフェイスの型であり、**M** はそのインターフェイス型のメソッド、プロパティ、またはイベントです。また、**A** はインデクサー引数のリストです。

厳密に単一継承のインターフェイスの場合(継承チェーン内の各インターフェイスが、直接基本インターフェイスを1つだけ持つか、まったく持たない場合)、メンバー検索(7.4 を参照)、メソッドの呼び出し(7.6.5.1 を参照)、インデクサー アクセス(7.6.6.2 を参照)の各規則の効果は、クラスや構造体の場合とまったく同じです。つまり、下位階層の派生メンバーは、同じ名前またはシグネチャを持つ上位階層の派生メンバーを隠ぺいします。ただし、多重継承インターフェイスの場合は、2つ以上の無関係な基本インターフェイスが同じ名前またはシグネチャでメンバーを宣言すると、あいまいさが生じることがあります。ここでは、あいまいさが生じる例をいくつか示します。すべての場合において、あいまいさを解決するために明示的なキャストを使用できます。

次に例を示します。

```

interface IList
{
    int Count { get; set; }
}
interface ICounter
{
    void Count(int i);
}
interface IListCounter: IList, ICounter {}
class C
{
    void Test(IListCounter x) {
        x.Count(1);                // Error
        x.Count = 1;               // Error
        ((IList)x).Count = 1;     // Ok, invokes IList.Count.set
        ((ICounter)x).Count(1);   // Ok, invokes ICounter.Count
    }
}

```

最初の 2 つのステートメントは、`IListCounter` 内の `Count` のメンバー検索 (7.4 を参照) があいまいであるため、コンパイルエラーが発生します。例に示すように、あいまいさは、`x` を適切な基本インターフェイス型にキャストすることで解決できます。このようなキャストでは、実行時に負荷がかかりません。コンパイル時に、インスタンスをより上位階層の型として処理するだけです。

次に例を示します。

```

interface IInteger
{
    void Add(int i);
}
interface IDouble
{
    void Add(double d);
}
interface INumber: IInteger, IDouble {}
class C
{
    void Test(INumber n) {
        n.Add(1);                // Invokes IInteger.Add
        n.Add(1.0);              // Only IDouble.Add is applicable
        ((IInteger)n).Add(1);    // Only IInteger.Add is a candidate
        ((IDouble)n).Add(1);    // Only IDouble.Add is a candidate
    }
}

```

呼び出し `n.Add(1)` は、オーバーロードの解決規則 (7.5.3 を参照) を適用することにより、`IInteger.Add` を選択します。同様に、呼び出し `n.Add(1.0)` は `IDouble.Add` を選択します。明示的なキャストを挿入すると、候補となるメソッドが

次に例を示します。

```

interface IBase
{
    void F(int i);
}
interface ILeft: IBase
{
    new void F(int i);
}

```

```

interface IRight: IBase
{
    void G();
}

interface IDerived: ILeft, IRight {}

class A
{
    void Test(IDerived d) {
        d.F(1);           // Invokes ILeft.F
        ((IBase)d).F(1); // Invokes IBase.F
        ((ILeft)d).F(1); // Invokes ILeft.F
        ((IRight)d).F(1); // Invokes IBase.F
    }
}

```

`IBase.F` メンバーは、`ILeft.F` メンバーによって隠ぺいされます。このため、`IRight` を経るアクセスパスでは `IBase.F` が隠ぺいされていないように見えるにもかかわらず、`d.F(1)` の呼び出しによって `ILeft.F` が選択されます。

多重継承インターフェイスにおける隠ぺいの規則は明快です。メンバーがいずれかのアクセスパスで隠ぺいされる場合は、すべてのアクセスパスで隠ぺいされます。`IDerived` から `ILeft` を経て `IBase` までのアクセスパスが `IBase.F` を隠ぺいするため、このメンバーは、`IDerived` から `IRight` を経て `IBase` までのアクセスパスでも隠ぺいされます。

### 13.3 インターフェイス メンバーの完全修飾名

インターフェイス メンバーは、[完全修飾名](#)で参照されることがあります。インターフェイス メンバーの完全限定名は、順に、そのメンバーを宣言するインターフェイスの名前、ドット、そのメンバーの名前で構成されます。メンバーの完全限定名は、そのメンバーを宣言するインターフェイスを参照します。たとえば、次の宣言があるとします。

```

interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

```

`Paint` の完全修飾名は `IControl.Paint`、`SetText` の完全修飾名は `ITextBox.SetText` です。

上の例では、`Paint` を `ITextBox.Paint` として参照することはできません。

インターフェイスが名前空間に含まれる場合、インターフェイス メンバーの完全限定名には名前空間名が含まれます。次に例を示します。

```

namespace System
{
    public interface ICloneable
    {
        object Clone();
    }
}

```

上の例の `Clone` メソッドの完全修飾名は、`System.ICloneable.Clone` になります。

## 13.4 インターフェイスの実装

インターフェイスは、クラスおよび構造体で実装できます。クラスまたは構造体がインターフェイスを直接実装していることを示すために、そのクラスまたは構造体の基底クラスリストにインターフェイス識別子が含まれます。次に例を示します。

```
interface ICloneable
{
    object Clone();
}

interface IComparable
{
    int CompareTo(object other);
}

class ListEntry: ICloneable, IComparable
{
    public object Clone() {...}
    public int CompareTo(object other) {...}
}
```

インターフェイスを直接実装するクラスや構造体も、そのインターフェイスの基本インターフェイスのすべてを暗黙的に直接実装します。クラスまたは構造体が、基底クラスリスト内にすべての基本インターフェイスを明示的にリストしない場合でも、この暗黙的な実装は行われます。次に例を示します。

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

class TextBox: ITextBox
{
    public void Paint() {...}
    public void SetText(string text) {...}
}
```

上の例の `TextBox` クラスは、`IControl` と `ITextBox` の両方を実装します。

クラス `C` でインターフェイスを直接実装すると、`C` から派生したすべてのクラスでも、そのインターフェイスが暗黙的に実装されます。クラス宣言で指定する基本インターフェイスとして、構築されたインターフェイス型(4.4を参照)を使用できます。基本インターフェイスはそれだけでは型パラメータにはなりませんが、スコープ内で型パラメーターと一緒に使用できます。次のコードは、クラスによる構築された型の実装および拡張方法を示します。

```
class C<U,V> {}

interface I1<V> {}

class D: C<string,int>, I1<string> {}

class E<T>: C<int,T>, I1<T> {}
```

ジェネリック クラス宣言の基本インターフェイスは、13.4.2 の一意性の規則を満たす必要があります。

### 13.4.1 インターフェイス メンバーの明示的実装

インターフェイスを実装するために、クラスまたは構造体はインターフェイス メンバーの明示的実装を宣言できます。インターフェイス メンバーの明示的実装とは、インターフェイス メンバーの完全限定名を参照する、メソッド、プロパティ、イベント、またはインデクサーの宣言です。次に例を示します。

```
interface IList<T>
{
    T[] GetElements();
}

interface IDictionary<K,V>
{
    V this[K key];
    void Add(K key, V value);
}

class List<T>: IList<T>, IDictionary<int,T>
{
    T[] IList<T>.GetElements() {...}
    T IDictionary<int,T>.this[int index] {...}
    void IDictionary<int,T>.Add(int index, T value) {...}
}
```

上の例の `IDictionary<int,T>.this` および `IDictionary<int,T>.Add` はインターフェイス メンバーの明示的実装です。

場合によっては、インターフェイス メンバーの名前が、インターフェイス メンバーを実装するクラスに適していないことがあります。そのような場合は、インターフェイス メンバーの明示的実装を使用してインターフェイス メンバーを実装できます。たとえば、ファイルの抽象化を実装するクラスで、ファイルリソースを解放する効果がある `Close` メンバー関数を実装し、インターフェイス メンバーの明示的実装を使用して `IDisposable` インターフェイスの `Dispose` メソッドを実装する場合が考えられます。

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

class MyFile: IDisposable
{
    void IDisposable.Dispose() {
        Close();
    }

    public void Close() {
        // Do what's necessary to close the file
        System.GC.SuppressFinalize(this);
    }
}
```

メソッドの呼び出し、プロパティ アクセス、またはインデクサー アクセスにおいて、完全限定名でインターフェイス メンバーの明示的実装にアクセスすることはできません。インターフェイス メンバーの明示的実装にアクセスできるのは、インターフェイス インスタンスからだけです。また、その場合は単にメンバーネームで参照されます。

インターフェイス メンバーの明示的実装にアクセス修飾子を含めると、コンパイル エラーになります。`abstract`、`virtual`、`override`、および `static` の各修飾子を含めた場合もコンパイル エラーになります。

インターフェイス メンバーの明示的実装には、他のメンバーとは異なるアクセシビリティ特性があります。インターフェイス メンバーの明示的実装は、メソッドの呼び出しやプロパティ アクセスにおいて、完全限定名ではアクセスできません。このため、アクセスはある意味ではプライベートです。ただし、インターフェイス インスタンスからはアクセスできるため、アクセスはある意味でパブリックでもあります。

インターフェイス メンバーの明示的実装には、主に次の 2 つの目的があります。

- インターフェイスの明示的実装はクラスや構造体のインスタンスからアクセスできないため、インターフェイス 実装を、クラスや構造体のパブリック インスタンスから除外できます。クラスや構造体が、そのクラスや構造体のコンシューマーに関係のない内部インターフェイスを実装する場合は、この機能が役立ちます。
- インターフェイス メンバーの明示的実装では、同じシグネチャを持つインターフェイス メンバーを明確に区別できます。インターフェイス メンバーの明示的実装を行わない場合、クラスや構造体が、同じシグネチャと戻り値の型を持つインターフェイス メンバーの異なる実装を持つことはできません。同様に、クラスや構造体が、同じシグネチャと異なる戻り値の型を持つインターフェイス メンバーの実装を持つことはできません。

インターフェイス メンバーの明示的実装を有効にするには、クラスや構造体が、インターフェイス メンバーの明示的実装の完全限定名、型、およびパラメーター型と正確に一致する完全限定名、型、およびパラメーター型を持つメンバーを含む、基底クラスリスト内のインターフェイスを指定する必要があります。次のクラスを例に考えてみます。

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
    int IComparable.CompareTo(object other) {...} // invalid
}
```

`IComparable` は `Shape` の基底クラスリストに含まれておらず、`ICloneable` の基本インターフェイスではないため、`IComparable.CompareTo` を宣言するとコンパイル時のエラーになります。同様に、次の宣言を考えてみます。

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
}
class Ellipse: Shape
{
    object ICloneable.Clone() {...} // invalid
}
```

`ICloneable` が `Ellipse` の基底クラスリストに明示的に含まれていないため、`Ellipse` 内で `ICloneable.Clone` を宣言するとコンパイル時のエラーになります。

インターフェイス メンバーの完全限定名は、そのメンバーを宣言したインターフェイスを参照する必要があります。したがって、次の宣言のようになります。

```

interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

class TextBox: ITextBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
}

```

`Paint` のインターフェイス メンバーの明示的実装は、`IControl.Paint` と記述する必要があります。

### 13.4.2 実装されるインターフェイスの一意性

ジェネリック型宣言によって実装されるインターフェイスは、予想されるすべての構築された型について一意である必要があります。この規則がないと、構築された型に対して呼び出す正しいメソッドを判断できません。たとえば、ジェネリック クラス宣言を次のように記述できると仮定します。

```

interface I<T>
{
    void F();
}

class X<U,V>: I<U>, I<V>           // Error: I<U> and I<V> conflict
{
    void I<U>.F() {...}
    void I<V>.F() {...}
}

```

このような記述が許可されると、次のような場合には実行するコードを判断できなくなります。

```

I<int> x = new X<int,int>();
x.F();

```

ジェネリック型宣言のインターフェイスリストが有効かどうかを判断するために、次の手順が実行されます。

- $L$  はジェネリック クラス、構造体、またはインターフェイス宣言  $C$  に直接指定されているインターフェイスのリストとします。
- $L$  に、既に  $L$  にあるインターフェイスの基本インターフェイスを追加します。
- $L$  から重複を削除します。
- $L$  を型引数で置換した後に、 $C$  から作成される構築された型によって  $L$  内の 2 つのインターフェイスが同じになる場合、 $C$  の宣言は無効です。すべての可能な構築された型を判断するときに、制約宣言は考慮されません。

上記のクラス宣言  $X$  では、インターフェイス リスト  $L$  は  $I<U>$  と  $I<V>$  から構成されます。構築された型の  $U$  と  $V$  が同じ型の場合は、これら 2 つのインターフェイスが同じ型になるため、宣言は無効になります。

異なる継承レベルで指定されたインターフェイスを統一できます。

```

interface I<T>
{
    void F();
}

class Base<U>: I<U>
{
    void I<U>.F() {...}
}

class Derived<U,V>: Base<U>, I<V> // ok
{
    void I<V>.F() {...}
}

```

このコードは、`Derived<U,V>` が `I<U>` と `I<V>` の両方を実装していますが、有効です。次に例を示します。

```

I<int> x = new Derived<int,int>();
x.F();

```

実質的に `Derived<int,int>` は `I<int>` を再実装するため、このコードは `Derived` のメソッドを呼び出します(13.4.6 を参照)。

### 13.4.3 ジェネリック メソッドの実装

ジェネリック メソッドが暗黙的にインターフェイス メソッドを実装するときに、各メソッドの型パラメーターに対して指定される制約は、(インターフェイスの型パラメーターが適切な型引数で置換された後は) 両方の宣言で同等である必要があります。このとき、メソッドの型パラメーターは左から右に向かって何番目であるかによって識別されます。

ただし、ジェネリック メソッドが明示的にインターフェイス メソッドを実装する場合、ジェネリック メソッドには制約が適用されません。代わりに、制約はインターフェイス メソッドから継承されます。

```

interface I<A,B,C>
{
    void F<T>(T t) where T: A;
    void G<T>(T t) where T: B;
    void H<T>(T t) where T: C;
}

class C: I<object,C,string>
{
    public void F<T>(T t) {...} // ok
    public void G<T>(T t) where T: C {...} // ok
    public void H<T>(T t) where T: string {...} // Error
}

```

メソッド `C.F<T>` は `I<object,C,string>.F<T>` を暗黙的に実装します。この場合、`object` はすべての型パラメーターの暗黙の制約であるため、`C.F<T>` は制約 `T: object` を指定する必要はありません(指定できません)。メソッド `C.G<T>` は `I<object,C,string>.G<T>` を暗黙的に実装します。これは、インターフェイスの型パラメーターが対応する型引数に置き換えられた後は、制約がインターフェイスの制約と一致するためです。メソッド `C.H<T>` の制約は、シール型(ここでは `string`) は制約として使用できないため、エラーになります。暗黙のインターフェイス メソッドの実装は一致する必要があるため、制約を省略した場合もエラーになります。したがって、  
`I<object,C,string>.H<T>` を暗黙的に実装することはできません。このインターフェイス メソッドは、次のように明示的なインターフェイス メンバー実装を使用した場合にのみ実装できます。

```

class C: I<object,C,string>
{
    ...
    public void H<U>(U u) where U: class {...}
    void I<object,C,string>.H<T>(T t) {
        string s = t; // Ok
        H<T>(t);
    }
}

```

この例では、明示的なインターフェイス メンバー実装によって、厳密に弱い制約を持つパブリック メソッドが呼び出されます。`T` は `T: string` の制約を継承するため、この制約がソース コードに表現されていなくても、`t` から `s` への代入は有効です。

#### 13.4.4 インターフェイス マップ

クラスや構造体は、そのクラスや構造体の基底クラス リストに含まれるインターフェイスのすべてのメンバーの実装を提供する必要があります。インターフェイス メンバーを実装するクラスや構造体内でインターフェイス メンバーの実装を探すプロセスは、**インターフェイス マップ**と呼ばれます。

クラスまたは構造体 `C` のインターフェイス マップでは、`C` の基底クラス リストで指定された各インターフェイスの各メンバーの実装を検索します。特定のインターフェイス メンバー `I.M`(`I` は、メンバ `M` が宣言されるインターフェイス)の実装は、各クラスまたは構造体 `S` を調べることで決定されます。検索は `C` から始まり、`C` の各基底クラスを順に探して、一致するものが見つかるまで続けます。

- `S` が、`I` および `M` と一致するインターフェイス メンバーの明示的実装の宣言を含んでいる場合は、このメンバーが `I.M` の実装です。
- `S` が、`M` と一致する非静的パブリック メンバーの宣言を含んでいる場合は、このメンバーが `I.M` の実装です。複数のメンバーが一致する場合、どのメンバーが `I.M` の実装であるかは指定されません。この状況が発生するのは、`S` が構築型であり、ジェネリック型で宣言された 2 つのメンバーが別々のシグネチャを持っているが、型引数によってシグネチャが同じになる場合だけです。

`C` の基底クラス リストで指定されたすべてのインターフェイスのすべてのメンバーを調べても実装が見つからない場合は、コンパイル エラーが発生します。インターフェイスのメンバーには、基本インターフェイスから継承されたメンバーが含まれます。

インターフェイス マップの目的では、クラス メンバー `A` は次の場合にインターフェイス メンバー `B` と一致します。

- `A` と `B` がメソッドであり、`A` と `B` の名前、型、および仮パラメーターリストが等しい。
- `A` と `B` がプロパティであり、`A` と `B` の名前および型が等しく、`A` は `B` と同じアクセサーを持つ(`A` がインターフェイス メンバーの明示的実装でなければ、`B` は追加のアクセサーを持つことができます)。
- `A` と `B` がイベントであり、`A` と `B` の名前および型が等しい。
- `A` と `B` がインデクサーであり、`A` と `B` の型および仮パラメーターリストが等しく、`A` は `B` と同じアクセサーを持つ(がインターフェイス メンバーの明示的実装でなければ、`A` は追加のアクセサーを持つことができます)。

インターフェイス マップのアルゴリズムの主な特性は次のとおりです。

- インターフェイス メンバーを実装するクラスや構造体のメンバーを決定するときに、インターフェイス メンバーの明示的実装は、同じクラスや構造体の他のメンバーより高い優先順位を持ちます。
- 非パブリック メンバーや静的メンバーは、インターフェイス マップに関与しません。

次に例を示します。

```
interface ICloneable
{
    object Clone();
}

class C: ICloneable
{
    object ICloneable.Clone() {...}
    public object Clone() {...}
}
```

インターフェイス メンバーの明示的実装は他のメンバーより優先されるため、**C** の **ICloneable.Clone** メンバーは、**ICloneable** 内の **Clone** の実装になります。

クラスや構造体が、同じ名前、型、およびパラメーター型を持つメンバーを含む 2 つ以上のインターフェイスを実装する場合は、これらのインターフェイス メンバーのそれぞれを、1 つのクラスまたは構造体メンバーに割り当てることができます。次に例を示します。

```
interface IControl
{
    void Paint();
}

interface IForm
{
    void Paint();
}

class Page: IControl, IForm
{
    public void Paint() {...}
}
```

この例では、**IControl** と **IForm** の両方の **Paint** メソッドが、**Page** 内の **Paint** メソッドに割り当てられます。また、この 2 つのメソッドのそれぞれに、インターフェイス メンバーの明示的実装を持たせることもできます。

クラスや構造体が、隠ぺいされたメンバーを持つインターフェイスを実装する場合、一部のメンバーはインターフェイス メンバーの明示的実装を使用して実装する必要があります。次に例を示します。

```
interface IBase
{
    int P { get; }

}

interface IDerived: IBase
{
    new int P();
}
```

このインスタンスの実装には、少なくとも 1 つのインスタンス メンバーの明示的実装が必要となります。実装は次の形式のいずれかになります。

```

class C: IDerived
{
    int IBase.P { get {...} }
    int IDerived.P() {...}
}

class C: IDerived
{
    public int P { get {...} }
    int IDerived.P() {...}
}

class C: IDerived
{
    int IBase.P { get {...} }
    public int P() {...}
}

```

クラスが、同じ基本インターフェイスを持つ複数のインターフェイスを実装する場合、その基本インターフェイスの実装は1つだけ存在できます。次に例を示します。

```

interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

class ComboBox: IControl, ITextBox, IListBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
    void IListBox.SetItems(string[] items) {...}
}

```

基底クラスリストで指定された `IControl`、`ITextBox` に継承された `IControl`、および `IListBox` に継承された `IControl` が、それぞれ別の実装を持つことはできません。実際、これらのインターフェイスについて個別の ID という概念はありません。ここでは、`ITextBox` と `IListBox` の実装は同じ `IControl` の実装を共有し、`ComboBox` は単に `IControl`、`ITextBox`、および `IListBox` の3つのインターフェイスを実装すると見なされます。

基底クラスのメンバーは、インターフェイスマップに関与します。次に例を示します。

```

interface Interface1
{
    void F();
}

class Class1
{
    public void F() {}
    public void G() {}
}

```

```
class Class2: Class1, Interface1
{
    new public void G() {}
}
```

Class1 のメソッド F は、Class2 の Interface1 の実装で使用されます。

### 13.4.5 インターフェイス実装の継承

クラスは、その基底クラスによって提供されるすべてのインターフェイス実装を継承します。

インターフェイスを明示的に **再実装**しないと、派生クラスは、その基底クラスから継承したインターフェイスマップを変更できません。たとえば、次の宣言があるとします。

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    public void Paint() {...}

class TextBox: Control
{
    new public void Paint() {...}
}
```

TextBox 内の Paint メソッドは Control 内の Paint メソッドを隠ぺいしますが、**IControl.Paint** に対する **Control.Paint** のマップは変更されません。また、クラスインスタンスとインターフェイスインスタンスを通じた Paint への呼び出しは、次のような結果になります。

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();           // invokes Control.Paint();
t.Paint();           // invokes TextBox.Paint();
ic.Paint();          // invokes Control.Paint();
it.Paint();          // invokes Control.Paint();
```

しかし、インターフェイスメソッドがクラス内の仮想メソッドに割り当てられる場合は、派生クラスが仮想メソッドをオーバーライドし、そのインターフェイスの実装を変更できます。たとえば、上の宣言は次のように記述し直すことができます。

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    public virtual void Paint() {...}

class TextBox: Control
{
    public override void Paint() {...}
}
```

この結果は次のようになります。

```

Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();           // invokes Control.Paint();
t.Paint();           // invokes TextBox.Paint();
ic.Paint();          // invokes Control.Paint();
it.Paint();          // invokes TextBox.Paint();

```

インターフェイス メンバーの明示的実装は仮想で宣言できないため、インターフェイス メンバーの明示的実装はオーバーライドできません。しかし、インターフェイス メンバーの明示的実装で別のメソッドを呼び出し、そのメソッドを仮想として宣言して派生クラスがそれをオーバーライドすることは有効です。次に例を示します。

```

interface IControl
{
    void Paint();
}

class Control: IControl
{
    void IControl.Paint() { PaintControl(); }
    protected virtual void PaintControl() {...}
}

class TextBox: Control
{
    protected override void PaintControl() {...}
}

```

この例で `Control` から派生したクラスは、`PaintControl` メソッドをオーバーライドすることによって、`IControl.Paint` の実装を特化できます。

#### 13.4.6 インターフェイスの再実装

インターフェイスの実装を継承するクラスは、基底クラスリストにそのインターフェイスを含めることによって、インターフェイスを **再実装** できます。

インターフェイスの再実装は、インターフェイスの初期実装とまったく同じインターフェイスマップ規則に従います。このため、継承したインターフェイスマップは、インターフェイスの再実装で確立されるインターフェイスマップに影響を及ぼしません。たとえば、次の宣言があるとします。

```

interface IControl
{
    void Paint();
}

class Control: IControl
{
    void IControl.Paint() {...}
}

class MyControl: Control, IControl
{
    public void Paint() {}
}

```

`Control` が `IControl.Paint` を `Control.IControl.Paint` に割り当てていても、`MyControl` の再実装には影響しません。`MyControl` の再実装では、`IControl.Paint` を `MyControl.Paint` に割り当てます。

継承されたパブリック メンバーの宣言と、継承されたインターフェイス メンバーの明示的宣言は、再実装されるインターフェイスのインターフェイス マップ プロセスに関与します。次に例を示します。

```
interface IMethods
{
    void F();
    void G();
    void H();
    void I();
}

class Base: IMethods
{
    void IMethods.F() {}
    void IMethods.G() {}
    public void H() {}
    public void I() {}
}

class Derived: Base, IMethods
{
    public void F() {}
    void IMethods.H() {}
}
```

この例では、`Derived` 内の `IMethods` の実装によって、各インターフェイス メソッドが `Derived.F`、`Base.IMethods.G`、`Derived.IMethods.H`、および `Base.I` に割り当てられます。

クラスがインターフェイスを実装する場合は、そのインターフェイスのすべての基本インターフェイスも暗黙的に実装します。同様に、インターフェイスの再実装では、そのインターフェイスのすべての基本インターフェイスも暗黙的に再実装します。次に例を示します。

```
interface IBase
{
    void F();
}

interface IDerived: IBase
{
    void G();
}

class C: IDerived
{
    void IBase.F() {...}
    void IDerived.G() {...}
}

class D: C, IDerived
{
    public void F() {...}
    public void G() {...}
}
```

この例では、`IDerived` の再実装によって `IBase` も再実装され、`IBase.F` が `D.F` に割り当てられます。

### 13.4.7 抽象クラスとインターフェイス

非抽象クラスと同様に、抽象クラスは、そのクラスの基底クラスリストに含まれるインターフェイスのすべてのメンバーの実装を提供する必要があります。ただし、抽象クラスは、インターフェイス メソッドを抽象メソッドに割り当てるすることができます。次に例を示します。

```
interface IMETHODS
{
    void F();
    void G();
}

abstract class C: IMETHODS
{
    public abstract void F();
    public abstract void G();
}
```

ここでは、IMETHODS の実装で F と G を抽象メソッドに割り当てています。これらの抽象メソッドは、C から派生する非抽象クラスでオーバーライドする必要があります。

インターフェイス メンバーの明示的実装は抽象にはできませんが、インターフェイス メンバーの明示的実装が抽象メソッドを呼び出すことはできます。次に例を示します。

```
interface IMETHODS
{
    void F();
    void G();
}

abstract class C: IMETHODS
{
    void IMETHODS.F() { FF(); }
    void IMETHODS.G() { GG(); }
    protected abstract void FF();
    protected abstract void GG();
}
```

この場合、C から派生する非抽象クラスは、FF と GG をオーバーライドする必要があります。これにより、IMETHODS の実際の実装が提供されます。



# 14. 列挙型

"**列挙型**" とは、名前付き定数のセットを宣言する固有の値型(4.1を参照)です。

次の例を参照してください。

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

この例では、Red、Green、およびBlue の各メンバーを持つ、Color という名前の列挙型が宣言されます。

## 14.1 列挙型の宣言

列挙型の宣言では、新しい列挙型を宣言します。列挙型の宣言はキーワード enum で始まり、その列挙型の名前、アクセシビリティ、基になる型、およびメンバーを定義します。

```
enum-declaration:
    attributesopt enum-modifiersopt enum identifier enum-baseopt enum-body ;opt
enum-base:
    : integral-type
enum-body:
    { enum-member-declarationsopt }
    { enum-member-declarations , }
```

各列挙型には、列挙型の**基になる型**と呼ばれる、対応する整数型があります。この基になる型は、列挙体で定義されるすべての列挙子値を表すことができる必要があります。列挙型の宣言では、基になる型として byte、sbyte、short、ushort、int、uint、long、または ulong を明示的に宣言できます。char を基になる型として使用することはできません。列挙型の宣言で基になる型を明示的に宣言しない場合、基になる型は int になります。

次の例を参照してください。

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

この例では、基になる型が long である列挙型を宣言します。この例のように、long の範囲内にあっても int の範囲から外れる値を使用できるようにするために、または将来の選択肢として残すために、基になる型として long を使用することもできます。

## 14.2 列挙修飾子

enum-declaration には、一連の列挙修飾子を含めることもできます。

```

enum-modifiers:
  enum-modifier
  enum-modifiers enum-modifier

enum-modifier:
  new
  public
  protected
  internal
  private

```

1 つの列挙型宣言内で同じ修飾子を複数回使用すると、コンパイルエラーになります。

列挙型宣言の修飾子の意味は、クラス宣言 (10.1.1 を参照) の修飾子の意味と同じです。ただし、**abstract** 修飾子と **sealed** 修飾子は、列挙型の宣言では使用できません。列挙型を抽象にすることはできず、派生もできません。

### 14.3 列挙型のメンバー

列挙型の宣言の本体では、0 個以上のメンバーを定義します。これらのメンバーは、列挙型の名前付き定数です。列挙型の 2 つのメンバーに同じ名前を付けることはできません。

```

enum-member-declarations:
  enum-member-declaration
  enum-member-declarations , enum-member-declaration

enum-member-declaration:
  attributesopt identifier
  attributesopt identifier = constant-expression

```

列挙型の各メンバーには、関連付けられた定数値があります。各メンバーに関連付けられた値の型は、列挙型の基になる型です。列挙型の各メンバーの定数値は、その列挙型の基になる型の範囲内にある必要があります。次の例を参照してください。

```

enum Color: uint
{
    Red = -1,
    Green = -2,
    Blue = -3
}

```

この例では、定数値 -1、-2、および -3 が、基になる整数型 **uint** の範囲にないため、コンパイルエラーになります。

列挙型の複数のメンバーで、同じ関連付けられた値を共有できます。次の例を参照してください。

```

enum Color
{
    Red,
    Green,
    Blue,
    Max = Blue
}

```

この例は、同じ値に関連付けられた 2 つのメンバー **Blue** と **Max** を持つ列挙型を示します。

列挙型のメンバーに関連付けられた値は、暗黙的または明示的に代入されます。列挙型のメンバーの宣言に *constant-expression* 初期化子がある場合は、その定数式の値が (列挙型の基になる型に暗黙的に

変換され) 列挙型のメンバーに関連付けられた値となります。列挙型のメンバーの宣言に初期化子がない場合は、関連付けられた値が次のように暗黙的に設定されます。

- そのメンバーが、列挙型で宣言された最初のメンバーである場合、関連付けられた値は 0 になります。
- それ以外の場合、列挙型のメンバーの関連付けられた値は、直前のメンバーの関連付けられた値に 1 を加えることで得られます。この加算後の値は、基になる型で表現できる値の範囲内にある必要があります。それ以外の場合は、コンパイル エラーになります。

次の例を参照してください。

```
using System;
enum Color
{
    Red,
    Green = 10,
    Blue
}
class Test
{
    static void Main() {
        Console.WriteLine(StringFromColor(Color.Red));
        Console.WriteLine(StringFromColor(Color.Green));
        Console.WriteLine(StringFromColor(Color.Blue));
    }
    static string StringFromColor(Color c) {
        switch (c) {
            case Color.Red:
                return String.Format("Red = {0}", (int) c);
            case Color.Green:
                return String.Format("Green = {0}", (int) c);
            case Color.Blue:
                return String.Format("Blue = {0}", (int) c);
            default:
                return "Invalid color";
        }
    }
}
```

この例では、列挙型のメンバーの名前、および列挙型のメンバーに関連付けられた値が出力されます。出力は次のとおりです。

```
Red = 0
Green = 10
Blue = 11
```

理由は次のとおりです。

- 列挙型のメンバー **Red** には、(初期化子がなく、最初のメンバーであるため) 自動的に値 0 が設定されます。
- 列挙型のメンバー **Green** には、明示的に値 10 が設定されます。
- 列挙型のメンバー **Blue** には、直前のメンバーに 1 を加算した値が自動的に設定されます。

列挙型のメンバーに関連付けられた値が、直接または間接的に、その関連付けられたメンバーの値を使用することはできません。この循環制限を除いて、列挙型メンバー初期化子は、その記述位置にか

かわらず、他のメンバー初期化子を自由に参照できます。列挙型メンバー初期化子内では、他のメンバーの値が、基になる型の型を持つものとして扱われます。このため、他のメンバーを参照するときにキャストは不要です。

次の例を参照してください。

```
enum Circular
{
    A = B,
    B
}
```

この例は、**A** と **B** の宣言が循環しているため、コンパイルエラーになります。**A** は明示的に **B** に依存し、**B** は暗黙的に **A** に依存しています。

列挙型のメンバーの名前とスコープは、クラス内のフィールドとほぼ同じ方法で設定されます。列挙型のメンバーのスコープは、列挙型の本体になります。そのスコープ内では、列挙型のメンバーを簡易名で参照できます。他のコードから参照する場合は、列挙型のメンバーの名前をその列挙型の名前で限定する必要があります。列挙型のメンバーは、宣言されたアクセシビリティを持ちません。列挙型にアクセスできる場合は、そのメンバーにもアクセスできます。

## 14.4 System.Enum 型

**System.Enum** 型は、すべての列挙型の抽象基底クラスであり、列挙型の基になる型とはまったく異なります。**System.Enum** から継承されたメンバーは、任意の列挙型で使用できます。任意の列挙型から **System.Enum** へのボックス化変換 (4.3.1 を参照)、および **System.Enum** から任意の列挙型へのボックス化解除変換 (4.3.2 を参照) が存在します。

**System.Enum** 自体は *enum-type* ではありません。**System.Enum** は、すべての *enum-type* の派生元である *class-type* です。**System.Enum** 型は、**System.ValueType** 型 (4.1.1 を参照) を継承し、また、この型は、**object** 型を継承します。実行時に、**System.Enum** 型の値は **null**、または任意の列挙型のボックス化された値への参照になります。

## 14.5 列挙型の値と演算

各列挙型は固有の型を定義します。列挙型と整数型の間、または列挙型どうしで変換するには、明示的な列挙値変換 (6.2.2 を参照) が必要です。列挙型が使用できる値のセットが、その列挙型のメンバーによって制限されることはありません。特に、列挙型の基になる型の値は、その列挙型にキャストでき、その列挙型の固有の有効値となります。

列挙型のメンバーは、列挙型の型を持ちます。ただし、他の列挙型メンバー初期化子内を除きます (14.3 を参照)。列挙型 **E** で宣言され、値 **v** が関連付けられたメンバーの値は、**(E)v** になります。

列挙型の値には、演算子 **==**、**!=**、**<**、**>**、**<=**、**>=** (7.10.5 を参照)、二項演算子 **+** (7.8.4 を参照)、二項演算子 **-** (7.8.5 を参照)、**^**、**&**、**|** (7.11.2 を参照)、**~** (7.7.4 を参照)、**++** と **--** (7.6.9 と 7.7.5 を参照) を使用できます。

すべての列挙型 **enum** は、**System.Enum** クラスから自動的に派生します (また、このクラスは **System.ValueType** と **object** から派生します)。 **\t "See also Enum"** **\t "See also Enum"** **\t "See also Enum"** **\t "See Enum"** **\t "See also Enum"** **\t "See Enum"** 列挙型の値には、このクラスの継承されたメソッドとプロパティを使用できます。

# 15. デリゲート

デリゲートを使用すると、C++、Pascal、Modulaなどの他の言語が関数ポインターで扱うシナリオを処理できるようになります。ただし、C++の関数ポインターとは異なり、デリゲートは完全にオブジェクト指向です。また、C++のメンバー関数へのポインターとは異なり、デリゲートはオブジェクトインスタンスとメソッドの両方をカプセル化します。

デリゲートの宣言では、**System.Delegate** クラスから派生するクラスを定義します。デリゲートインスタンスは、1つ以上のメソッドを列挙した呼び出しリストをカプセル化します。カプセル化された各メソッドは、"呼び出し可能なエンティティ"と呼ばれます。インスタンスマソッドの場合、呼び出し可能なエンティティは、インスタンスと、そのインスタンスのメソッドで構成されています。静的メソッドの場合、呼び出し可能なエンティティはメソッドだけで構成されます。適切な引数を指定してデリゲートインスタンスを呼び出すと、デリゲートの呼び出し可能なすべてのエンティティが、指定した引数を使用して呼び出されます。

デリゲートインスタンスの興味深く有用な性質としては、デリゲートインスタンスがカプセル化しているメソッドのクラスについて何も知らず、関心もないということがあります。重要なことは、これらのメソッドとデリゲートの型の間に互換性がある(15.1を参照)ということだけです。このため、デリゲートは"匿名"の呼び出しに適しています。

## 15.1 デリゲートの宣言

*delegate-declaration* は、新しいデリゲート型を宣言する *type-declaration*(9.6を参照)です。

```

delegate-declaration:
  attributesopt delegate-modifiersopt delegate return-type
    identifier variant-type-parameter-listopt
    ( formal-parameter-listopt ) type-parameter-constraints-clausesopt ;

delegate-modifiers:
  delegate-modifier
  delegate-modifiers delegate-modifier

delegate-modifier:
  new
  public
  protected
  internal
  private

```

1つのデリゲート宣言内で同じ修飾子を複数回使用すると、コンパイルエラーになります。

**new**修飾子を使用できるのは、別の型の中で宣言されたデリゲートだけです。この修飾子は、デリゲートが、継承されたメンバーと同じ名前で隠ぺいするように指定します。詳細については、10.3.4を参照してください。

**public**、**protected**、**internal**、および**private**の各修飾子は、デリゲート型のアクセシビリティを制御します。クラスの宣言が行われるコンテキストによっては、これらの修飾子の一部は使用できません(3.5.1を参照)。

デリゲートの型名は *identifier* です。

省略可能な *formal-parameter-list* は、デリゲートのパラメーターを指定します。*return-type* は、デリゲートの戻り値の型を示します。

省略可能な *variant-type-parameter-list* (13.1.3 を参照) は、そのデリゲート自体に型パラメーターを指定します。

デリゲート型の戻り値の型は、**void** であるか、または出力セーフ (13.1.3.1 を参照) である必要があります。

デリゲート型のすべての仮パラメーター型は、入力セーフである必要があります。また、**out** パラメーターまたは**ref** パラメーターは出力セーフである必要もあります。基になる実行プラットフォームの制限により、**out** パラメーターも入力セーフである必要があります。

C# のデリゲート型は、名前は同じでも、構造的に同じではありません。たとえば、パラメーターリストと戻り値の型が同じである 2 つの異なるデリゲート型は、異なるデリゲート型と見なされます。ただし、2 つの個別のデリゲート型が構造的には同じである場合、それぞれのインスタンスは等しいと解釈されることがあります (7.9.8 を参照)。

次に例を示します。

```
delegate int D1(int i, double d);
class A
{
    public static int M1(int a, double b) {...}
}
class B
{
    delegate int D2(int c, double d);
    public static int M1(int f, double g) {...}
    public static void M2(int k, double l) {...}
    public static int M3(int g) {...}
    public static void M4(int g) {...}
}
```

メソッドの **A.M1** と **B.M1** は、戻り値の型とパラメーターリストが同じであるため、両方ともデリゲート型の **D1** および **D2** と互換性があります。ただし、これらのデリゲート型は 2 つの異なる型であるため、相互交換はできません。メソッドの **B.M2**、**B.M3**、および **B.M4** は、戻り値の型やパラメーターリストが異なるため、デリゲート型の **D1** および **D2** とは互換性がありません。

他のジェネリック型宣言と同様に、構築されたデリゲート型を作成するには型引数を指定する必要があります。構築されたデリゲート型のパラメーター型および戻り値の型は、デリゲードの宣言の各型パラメーターを構築されたデリゲート型の対応する型引数で置き換えて作成します。結果として得られる戻り値の型のパラメーター型を使用して、構築されたデリゲート型と互換性のあるメソッドが判断されます。次に例を示します。

```
delegate bool Predicate<T>(T value);
class X
{
    static bool F(int i) {...}
    static bool G(string s) {...}
}
```

メソッド `X.F` はデリゲート型 `Predicate<int>` と互換性があり、メソッド `X.G` はデリゲート型 `Predicate<string>` と互換性があります。

デリゲート型を宣言する唯一の方法は *delegate-declaration* を使用することです。デリゲート型は、`System.Delegate` から派生するクラス型です。デリゲート型は暗黙的に `sealed` であり、デリゲート型から型を派生させることはできません。また、`System.Delegate` から非デリゲートクラス型を派生させることもできません。`System.Delegate` 自体はデリゲート型ではありません。すべてのデリゲート型の派生元となるクラス型です。

C# には、デリゲートのインスタンス化と呼び出し用に、特別の構文が用意されています。インスタンス化を除き、クラスやクラスインスタンスに適用できる操作は、デリゲートクラスやデリゲートインスタンスにも適用できます。特に、通常のメンバーアクセス構文を使用して、`System.Delegate` 型のメンバーにアクセスできます。

デリゲートインスタンスによってカプセル化されたメソッドのセットは、呼び出しリストと呼ばれます。デリゲートインスタンスが 1 つのメソッドから作成される場合 (15.2 を参照)、デリゲートインスタンスはそのメソッドをカプセル化し、呼び出しリストにはエントリが 1 つだけ格納されます。ただし、`null` 以外の 2 つのデリゲートインスタンスが組み合わされる場合、それぞれの呼び出しリストが左のオペランドから右のオペランドの順に連結されて、新しい呼び出しリストが形成されます。このリストには 2 つ以上のエントリが格納されます。

デリゲートを連結するには、二項演算子 `+` (7.8.4 を参照) と `+=` 演算子 (7.17.2 を参照) を使用します。デリゲートをデリゲートの組み合わせから削除するには、二項演算子 `-` (7.8.5 を参照) と `-=` 演算子 (7.17.2 を参照) を使用します。また、デリゲートが等しいかどうかを比較できます (7.10.8 を参照)。

次の例は、いくつかのデリゲートのインスタンス化と、それぞれに対応する呼び出しリストを示しています。

```
delegate void D(int x);
class C
{
    public static void M1(int i) {...}
    public static void M2(int i) {...}
}
class Test
{
    static void Main()
    {
        D cd1 = new D(C.M1);      // M1
        D cd2 = new D(C.M2);      // M2
        D cd3 = cd1 + cd2;        // M1 + M2
        D cd4 = cd3 + cd1;        // M1 + M2 + M1
        D cd5 = cd4 + cd3;        // M1 + M2 + M1 + M1 + M2
    }
}
```

`cd1` と `cd2` がインスタンス化されると、それぞれが 1 つのメソッドをカプセル化します。`cd3` がインスタンス化されると、`M1` と `M2` の 2 つのメソッドをこの順序で格納した呼び出しリストが設定されます。`cd4` の呼び出しリストには、`M1`、`M2`、および `M1` がこの順序で格納されます。`cd5` の呼び出しリストには、`M1`、`M2`、`M1`、`M1`、および `M2` がこの順序で格納されます。デリゲートの組み合わせと削除のその他の例については、15.4 を参照してください。

## 15.2 デリゲートの互換性

メソッドまたはデリゲート M は、次のすべての条件が満たされる場合に "互換性" を、デリゲート型 D と持ります。

- D と M には同じ数のパラメーターがあり、D の各パラメーターには対応する M のパラメーターと同じ ref または out 修飾子があります。
- 各値パラメーター (ref または out 修飾子を持たないパラメーター) には、D のパラメーター型から対応する M のパラメーター型への恒等変換 (6.1.1 を参照) または暗黙の参照変換 (6.1.6 を参照) が存在します。
- ref パラメーターまたは out パラメーターのそれぞれについて、D のパラメーター型は M のパラメーター型と同じです。
- M の戻り値の型から D の戻り値の型への恒等変換または暗黙の参照変換が存在します。

## 15.3 デリゲートのインスタンス化

デリゲートのインスタンスは、*delegate-creation-expression* (7.6.10.5 を参照) またはデリゲート型への変換によって作成されます。新しく作成されたデリゲートインスタンスは、次のいずれかを参照します。

- *delegate-creation-expression* で参照される静的メソッド
- ターゲットオブジェクト (null 以外) と *delegate-creation-expression* で参照されるインスタンスマソッド
- 他のデリゲート

次に例を示します。

```
delegate void D(int x);
class C
{
    public static void M1(int i) {...}
    public void M2(int i) {...}
}
class Test
{
    static void Main() {
        D cd1 = new D(C.M1);      // static method
        C t = new C();
        D cd2 = new D(t.M2);      // instance method
        D cd3 = new D(cd2);      // another delegate
    }
}
```

デリゲートがインスタンス化されると、デリゲートインスタンスは常に同じターゲットオブジェクトとメソッドを参照します。2つのデリゲートを組み合わせるか、または一方から他方を削除すると、結果として作成される新しいデリゲートには、独自の呼び出しリストが設定されます。組み合わされたり削除されたりしたデリゲートの呼び出しリストは、そのまま残ります。

## 15.4 デリゲートの呼び出し

C# には、デリゲートの呼び出し用に特別の構文が用意されています。呼び出しリストに1つのエントリを持つ null 以外のデリゲートインスタンスが呼び出されると、呼び出されたデリゲートインス

タンスは、指定された同じ引数を使用して 1 つのメソッドを呼び出し、参照先のメソッドと同じ値を返します。デリゲートの呼び出しの詳細については、7.6.5.3 を参照してください。このようなデリゲートの呼び出し中に例外が発生し、呼び出されたメソッド内でその例外がキャッチされない場合は、デリゲートが参照したメソッドをそのメソッドが直接呼び出したかのように、例外キャッチ句の検索がデリゲートを呼び出したメソッド内で継続します。

複数のエントリが格納された呼び出しリストを持つデリゲートインスタンスの呼び出しは、その呼び出しリストの各メソッドを同期的に順に呼び出すことによって進行します。呼び出されたそれぞれのメソッドには、デリゲートインスタンスに指定されたのと同じ引数のセットが渡されます。このようなデリゲートの呼び出しに参照パラメーター (10.6.1.2 を参照) が含まれる場合、各メソッド呼び出しは、同じ変数への参照を使用して発生します。つまり、呼び出しリスト内のあるメソッドがその変数に変更を加えると、呼び出しリストのその後のメソッドにその変更が示されます。デリゲートの呼び出しに出力パラメーターや戻り値が含まれる場合、最終的な値は、リストの最後のデリゲート呼び出しによって返されます。

このようなデリゲートの呼び出し処理中に例外が発生し、呼び出されたメソッド内でその例外がキャッチされない場合、例外キャッチ句の検索はデリゲートを呼び出したメソッド内で継続し、呼び出しリスト内のその後のメソッドは呼び出されません。

値が null のデリゲートインスタンスを呼び出そうとすると、`System.NullReferenceException` の例外が発生します。

デリゲートのインスタンス化、組み合わせ、削除、および呼び出しを行う例を次に示します。

```
using System;
delegate void D(int x);
class C
{
    public static void M1(int i) {
        Console.WriteLine("C.M1: " + i);
    }
    public static void M2(int i) {
        Console.WriteLine("C.M2: " + i);
    }
    public void M3(int i) {
        Console.WriteLine("C.M3: " + i);
    }
}
class Test
{
    static void Main() {
        D cd1 = new D(C.M1);
        cd1(-1);           // call M1
        D cd2 = new D(C.M2);
        cd2(-2);           // call M2
        D cd3 = cd1 + cd2;
        cd3(10);          // call M1 then M2
        cd3 += cd1;
        cd3(20);          // call M1, M2, then M1
        C c = new C();
        D cd4 = new D(c.M3);
        cd3 += cd4;
        cd3(30);          // call M1, M2, M1, then M3
    }
}
```

```

cd3 -= cd1;          // remove last M1
cd3(40);           // call M1, M2, then M3

cd3 -= cd4;          // call M1 then M2
cd3(50);

cd3 -= cd2;          // call M1
cd3(60);

cd3 -= cd2;          // impossible removal is benign
cd3(60);           // call M1

cd3 -= cd1;          // invocation list is empty so cd3 is null
//    cd3(70);           // System.NullReferenceException thrown
cd3 -= cd1;          // impossible removal is benign
}
}

```

ステートメント `cd3 += cd1;` に示されるように、1つのデリゲートが呼び出しリストに複数回存在できます。この場合は、呼び出しリストに出現するたびに、そのデリゲートが1回呼び出されます。例のような呼び出しリストでそのデリゲートが削除される場合、実際に削除されるのは、呼び出しリストで最後に出現するデリゲートです。

最後のステートメント `cd3 -= cd1;` の実行直前に、デリゲート `cd3` は空の呼び出しリストを参照します。空のリストからデリゲートを削除(または、空でないリストから存在しないデリゲートを削除)しようとしても、エラーにはなりません。

生成される出力は次のとおりです。

```

C.M1: -1
C.M2: -2
C.M1: 10
C.M2: 10
C.M1: 20
C.M2: 20
C.M1: 20
C.M1: 30
C.M2: 30
C.M1: 30
C.M3: 30
C.M1: 40
C.M2: 40
C.M3: 40
C.M1: 50
C.M2: 50
C.M1: 60
C.M1: 60

```

# 16. 例外

C# の例外は、システム レベルのエラーとアプリケーション レベルのエラーの両方を処理するために用意されている、構造化された、一定のタイプセーフな手段を利用して処理できます。C# の例外処理機構は C++ とほぼ同じですが、いくつかの重要な違いがあります。

- C# では、すべての例外が、`System.Exception` から派生したクラス型のインスタンスによって表される必要があります。C++ では、任意の型の任意の値を使用して例外を表すことができます。
- C# では、`finally` ブロック (8.10 を参照) を使用して、通常実行と例外状態の両方で実行する終了コードを記述できます。C++ の場合、コードを 2 回記述せずにそのような終了コードを記述するのは困難です。
- C# の場合、オーバーフロー、0 による除算、`null` 逆参照などのシステム レベルの例外は、適切に定義された例外クラスを持ち、アプリケーション レベルのエラー状態と同等になります。

## 16.1 例外の原因

例外は次の 2 つの方法でスローされます。

- `throw` ステートメント (8.9.5 を参照) は、即時かつ無条件に例外をスローします。制御は `throw` 直後のステートメントに到達しません。
- C# のステートメントや式の処理中に発生する特定の状態によって、処理を正常に終了できないときに、例外が発生します。たとえば、整数除算演算 (7.8.2 を参照) で分母が 0 の場合は、`System.DivideByZeroException` がスローされます。この理由で発生する例外のリストについては、16.4 を参照してください。

## 16.2 System.Exception クラス

`System.Exception` クラスは、すべての例外の基本型です。このクラスは、すべての例外が共有する、いくつかの重要なプロパティを持ちます。

- `Message` は、`string` 型の読み取り専用プロパティです。このプロパティには、例外の理由についての、人間が読むことができる形式の説明が格納されます。
- `InnerException` は、`Exception` 型の読み取り専用プロパティです。値が `null` 以外の場合は、現在の例外の原因となった例外を参照します。つまり、現在の例外は、`InnerException` を処理する `catch` ブロックで発生しました。値が `null` の場合は、この例外が別の例外によって起こされたものではないことを示します。この方法でチェインされる例外オブジェクトの数は不定です。

これらのプロパティの値は、`System.Exception` のインスタンス コンストラクターの呼び出しで指定できます。

## 16.3 例外の処理方法

例外は `try` ステートメント (8.10 を参照) で処理されます。

例外が発生すると、システムは、例外を処理できる最も近い `catch` 句を探します。この `catch` 句は、例外の実行時型によって決まります。最初に、現在のメソッドから構文的に外側にある `try` ステート

メントが検索され、その `try` ステートメントに関連付けられた `catch` 句が順に検討されます。これに失敗すると、現在のメソッドを呼び出したメソッドで、構文的に外側にある `try` ステートメントが検索されます。このステートメントは、現在のメソッドが呼び出される部分を囲むステートメントです。この検索は、スローされた例外の実行時型と同じクラスである例外クラス、またはスローされた例外の実行時型の基底クラスである例外クラスを指定することによって、現在の例外を処理できる `catch` 句が見つかるまで続けられます。例外クラスを指定しない `catch` 句は、どのような例外でも処理できます。

一致する `catch` 句が見つかると、その `catch` 句の最初のステートメントに制御が移行するように準備されます。`catch` 句の実行が開始する前に、例外をキャッチした `try` ステートメントよりも内側の入れ子になっている `try` ステートメントに関連付けられている `finally` 句が順に実行されます。

一致する `catch` 句が見つからない場合は、次のいずれかの処理が行われます。

- 一致する `catch` 句の検索が、静的コンストラクター(10.12 を参照)または静的フィールド初期化子に到達した場合は、静的コンストラクターの呼び出しが発生したときに `System.TypeInitializationException` がスローされます。`System.TypeInitializationException` の内部例外には、最初にスローされた例外が含まれます。
- 一致する `catch` 句の検索が、スレッドを最初に起動したコードに到達した場合は、スレッドの実行が終了します。このような終了の影響は、実装で定義されます。

デストラクターの実行中に発生した例外には、特に注意する必要があります。デストラクターの実行中に発生した例外がキャッチされない場合は、そのデストラクターの実行が終了し、基底クラスのデストラクター(存在する場合)が呼び出されます。`object` 型のように基底クラスが存在しない場合、または基底クラスのデストラクターが存在しない場合は、例外が破棄されます。

### 16.4 共通の例外クラス

C# の特定の処理によって、次の例外がスローされます。

<code>System.ArithmetiException</code>	<code>System.DivideByZeroException</code> や <code>System.OverflowException</code> など、算術演算中に発生する例外の基底クラスです。
<code>System.ArrayTypeMismatchException</code>	格納される要素の実際の型が、配列の実際の型と互換性がないために、配列への格納に失敗した場合にスローされます。
<code>System.DivideByZeroException</code>	整数値を 0 で除算しようとしたときにスローされます。
<code>System.IndexOutOfRangeException</code>	0 未満のインデックス、または配列の範囲外のインデックスを使用して、配列のインデックス付けをしようとしたときにスローされます。
<code>System.InvalidCastException</code>	基本型またはインターフェイスから派生型への明示的変換が、実行時に失敗した場合にスローされます。
<code>System.NullReferenceException</code>	参照先オブジェクトを必要とする方法で <code>null</code> 参照が使用された場合にスローされます。
<code>System.OutOfMemoryException</code>	<code>new</code> を使用してメモリを割り当てようとして、割り当てに失敗した場合にスローされます。
<code>System.OverflowException</code>	<code>checked</code> コンテキストで算術演算がオーバーフローしたときにスローされます。
<code>System.StackOverflowException</code>	保留状態のメソッド呼び出しが多すぎて、実行スタックに空きがなくなったときにスローされます。一般的には、再帰が深いか、または無限再帰の場合に、この例外が発生します。
<code>System.TypeInitializationException</code>	静的コンストラクターが例外をスローし、その例外をキャッチする <code>catch</code> 句が存在しない場合にスローされます。



# 17. 属性

C# 言語では、ほとんどの場合、プログラムで定義されるエンティティについての宣言情報を指定できます。たとえば、クラス内のメソッドのアクセシビリティは、*method-modifiers* の **public**、**protected**、**internal**、および **private** でメソッドを修飾することによって指定します。

C# では、"属性" と呼ばれる新しい種類の宣言情報を作成できます。作成した属性は、さまざまなプログラム エンティティに追加できます。また、実行時環境で属性情報を取得できます。たとえば、フレームワークが、特定のプログラム要素(クラスやメソッドなど)に対して設定できる **HelpAttribute** 属性を定義することで、これらのプログラム要素からドキュメントへのマップを提供することができます。

属性は、属性クラス(17.1 を参照)の宣言を通じて定義されます。属性クラスは、位置指定パラメーターと名前付きパラメーター(17.1.2 を参照)を持つことができます。属性は、属性の指定(17.2 を参照)を使用して、C# プログラム内のエンティティに追加されます。また、属性インスタンス(17.3 を参照)として実行時に取得できます。

## 17.1 属性クラス

抽象クラス **System.Attribute** から直接または間接的に派生するクラスは、"属性クラス" です。属性クラスの宣言では、宣言に使用できる新しい種類の属性を定義します。規約により、属性クラスの名前には **Attribute** というサフィックスを付けます。属性を使用するときには、このサフィックスを含めることも省略することもできます。

### 17.1.1 属性の使用法

属性 **AttributeUsage**(17.4.1 を参照)を使用すると、属性クラスの使用法を説明できます。

**AttributeUsage** には、1 つの位置指定パラメーター(17.1.2 を参照)があります。属性クラスは、このパラメーターを使用して、その属性クラスを使用できる宣言の種類を指定できます。次の例を参照してください。

```
using System;
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: Attribute
{
    ...
}
```

この例は、*class-declaration* および *interface-declaration* にのみ使用できる、**SimpleAttribute** という名前の属性クラスを定義しています。次の例を参照してください。

```
[Simple] class Class1 {...}
[Simple] interface Interface1 {...}
```

この例は、**Simple** 属性のいくつかの使用方法を示しています。この属性は **SimpleAttribute** という名前で定義されていますが、使用されるときは、**Attribute** サフィックスを省略して **Simple** という短縮名になっています。したがって、上の例は次と同じ意味になります。

```
[SimpleAttribute] class Class1 {...}
```

```
[SimpleAttribute] interface Interface1 {...}
```

`AttributeUsage` には、`AllowMultiple` という名前付きパラメーター (17.1.2 を参照) があります。このパラメーターは、特定のエンティティでこの属性を複数回指定できるかどうかを示します。属性クラスの `AllowMultiple` が `true` の場合は、"複数回使用できる属性クラス" であり、1つのエンティティに対して複数回指定できます。属性クラスの `AllowMultiple` が `false` であるか、または指定されていない場合は、"1回だけ使用できる属性クラス" であり、1つのエンティティに対して1回だけ指定できます。

次の例を参照してください。

```
using System;
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class AuthorAttribute: Attribute
{
    private string name;
    public AuthorAttribute(string name) {
        this.name = name;
    }
    public string Name {
        get { return name; }
    }
}
```

この例では、`AuthorAttribute` という名前の複数回使用できる属性クラスを定義します。次の例を参照してください。

```
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
class Class1
{
    ...
}
```

この例は、`Author` 属性を2回使用するクラス宣言を示しています。

`AttributeUsage` は、`Inherited` というもう1つの名前付きパラメーターを持ちます。このパラメーターは、属性が基底クラスに対して指定された場合に、その基底クラスから派生するクラスにもその属性が継承されるかどうかを示します。属性クラスの `Inherited` が `true` の場合、その属性は継承されます。属性クラスの `Inherited` が `false` の場合、その属性は継承されません。指定しない場合は、既定で `true` になります。

次のように、`AttributeUsage` 属性が追加されていない属性クラス `X` があるとします。

```
using System;
class X: Attribute {...}
```

これは次と同じです。

```
using System;
[AttributeUsage(
    AttributeTargets.All,
    AllowMultiple = false,
    Inherited = true)
]
class X: Attribute {...}
```

### 17.1.2 位置指定パラメーターと名前付きパラメーター

属性クラスは、**位置指定パラメーター**と**名前付きパラメーター**を持つことができます。属性クラスのそれぞれのパブリックインスタンスコンストラクターは、その属性クラスに有効な一連の位置指定パラメーターを定義します。属性クラスの非静的でパブリックな読み取り/書き込みフィールドとプロパティは、その属性クラスの名前付きパラメーターを定義します。

次の例を参照してください。

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute
{
    public HelpAttribute(string url) {      // Positional parameter
        ...
    }

    public string Topic {                  // Named parameter
        get {...}
        set {...}
    }

    public string Url {
        get {...}
    }
}
```

この例では、1つの位置指定パラメーター `url` と1つの名前付きパラメーター `Topic` を持つ、`HelpAttribute` という名前の属性クラスを定義します。この属性クラスは非静的でパブリックですが、読み取り/書き込みではないため、`url` プロパティは名前付きパラメーターを定義しません。

この属性クラスは、次のように使用できます。

```
[Help("http://www.mycompany.com/.../class1.htm")]
class Class1
{
    ...

[Help("http://www.mycompany.com/.../Misc.htm", Topic = "class2")]
class Class2
{
    ...
}
```

### 17.1.3 属性パラメーター型

属性クラスの位置指定パラメーターと名前付きパラメーターの型は、次に示す**属性パラメーター型**に制限されます。

- `bool`、`byte`、`char`、`double`、`float`、`int`、`long`、`sbyte`、`short`、`string`、`uint`、`ulong`、`ushort` のいずれかの型。
- `object` 型。
- `System.Type` 型。
- 列挙型。ただし、その列挙型が `public` アクセシビリティを持ち、その列挙型が入れ子になった型（存在する場合）も `public` アクセシビリティを持つ場合に限ります（17.2 を参照）。
- 上のいずれかの型の1次元配列。

上記のいずれかの型を持たないコンストラクター引数またはパブリック フィールドは、属性の指定において位置指定パラメーターまたは名前付きパラメーターとして使用できません。

## 17.2 属性の指定

**属性の指定**とは、あらかじめ定義された属性を宣言に適用することです。属性とは、宣言に指定される、追加の宣言情報です。属性は、グローバル スコープで指定することも（この場合、それを含むアセンブリまたはモジュールに属性が指定されます）、*type-declaration* (9.6 を参照)、*class-member-declaration* (10.1.5 を参照)、*interface-member-declaration* (13.2 を参照)、*struct-member-declaration* (11.2 を参照)、*enum-member-declaration* (14.3 を参照)、*accessor-declaration* (10.7.2 を参照)、*event-accessor-declaration* (10.8.1 を参照)、および *formal-parameter-list* (10.6.1 を参照) に対して指定することができます。

属性は**属性セクション**で指定されます。属性セクションは、1つ以上の属性のコンマ区切りリストを囲む、このようなリストに指定される属性の順序、および同じプログラム エンティティに付加されたセクションの順序は、重要ではありません。たとえば、属性の指定 [A][B]、[B][A]、[A,B]、および [B,A] は同じです。

```

global-attributes:
    global-attribute-sections

global-attribute-sections:
    global-attribute-section
    global-attribute-sections global-attribute-section

global-attribute-section:
    [ global-attribute-target-specifier attribute-list ]
    [ global-attribute-target-specifier attribute-list , ]

global-attribute-target-specifier:
    global-attribute-target :

global-attribute-target:
    assembly
    module

attributes:
    attribute-sections

attribute-sections:
    attribute-section
    attribute-sections attribute-section

attribute-section:
    [ attribute-target-specifieropt attribute-list ]
    [ attribute-target-specifieropt attribute-list , ]

attribute-target-specifier:
    attribute-target :

```

```

attribute-target:
  field
  event
  method
  param
  property
  return
  type

attribute-list:
  attribute
  attribute-list , attribute

attribute:
  attribute-name attribute-argumentsopt

attribute-name:
  type-name

attribute-arguments:
  ( positional-argument-listopt )
  ( positional-argument-list , named-argument-list )
  ( named-argument-list )

positional-argument-list:
  positional-argument
  positional-argument-list , positional-argument

positional-argument:
  argument-nameopt attribute-argument-expression

named-argument-list:
  named-argument
  named-argument-list , named-argument

named-argument:
  identifier = attribute-argument-expression

attribute-argument-expression:
  expression

```

属性は、*attribute-name*、および位置指定引数と名前付き引数の省略可能なリストで構成されます。位置指定引数(存在する場合)は、名前付き引数より前に指定します。位置指定引数は、*attribute-argument-expression* で構成されます。名前付き引数は、名前、等号、*attribute-argument-expression* の順序で構成されます。これらの引数は、単純な代入の場合と同じ規則による制約を受けます。名前付き引数の順序は重要ではありません。

*attribute-name* は属性クラスを識別します。*attribute-name* の形式が *type-name* である場合、この名前は属性クラスを参照する必要があります。それ以外の場合は、コンパイル時のエラーが発生します。次の例を参照してください。

```

class class1 {}
[class1] class class2 {} // Error

```

この例は、*class1* が属性クラスではない際に、*class1* を属性クラスとして使用しているため、コンパイルエラーになります。

特定のコンテキストでは、複数のターゲットに属性の指定を行うことができます。プログラムは、*attribute-target-specifier* を含めることによって、ターゲットを明示的に指定できます。属性をグローバル レベルで設定する場合は、*global-attribute-target-specifier* が必要です。他のすべての位置では適切な既定値が適用されますが、*attribute-target-specifier* を使用して、あいまいな状況にある既定値を確認またはオーバーライドしたり、あいまいでない状況にある既定値を単に確認したりできます。このため、グローバル レベルを除いて、通常は *attribute-target-specifier* を省略できます。あいまいになる可能性があるコンテキストは、次のように解決されます。

- グローバル スコープで指定された属性は、ターゲット アセンブリまたはターゲット モジュールに適用できます。このコンテキストには既定値が存在しないため、このコンテキストでは常に *attribute-target-specifier* が必要です。*attribute-target-specifier* が **assembly** の場合は、この属性がターゲット アセンブリに適用されることを示します。*attribute-target-specifier* が **module** の場合は、この属性がターゲット モジュールに適用されることを示します。
- デリゲートの宣言で指定される属性は、宣言されるデリゲート、またはその戻り値に適用できます。*attribute-target-specifier* が指定されない場合、属性はデリゲートに適用されます。*attribute-target-specifier* が **type** の場合は、この属性がデリゲートに適用されることを示します。*attribute-target-specifier* が **return** の場合は、この属性が戻り値に適用されることを示します。
- メソッドの宣言で指定される属性は、宣言されるメソッド、またはその戻り値に適用できます。*attribute-target-specifier* が指定されない場合、属性はメソッドに適用されます。*attribute-target-specifier* が **method** の場合は、この属性がメソッドに適用されることを示します。*attribute-target-specifier* が **return** の場合は、この属性が戻り値に適用されることを示します。
- 演算子の宣言で指定される属性は、宣言される演算子、またはその戻り値に適用できます。*attribute-target-specifier* が指定されない場合、属性は演算子に適用されます。*attribute-target-specifier* が **method** の場合は、この属性が演算子に適用されることを示します。*attribute-target-specifier* が **return** の場合は、この属性が戻り値に適用されることを示します。
- イベント アクセサーを省略するイベントの宣言で指定される属性は、宣言されるイベント、関連付けられたフィールド(イベントが非抽象の場合)、または関連付けられた **add** メソッドと **remove** メソッドに適用できます。*attribute-target-specifier* が指定されない場合、属性はイベントに適用されます。*attribute-target-specifier* が **event** の場合は、この属性がイベントに適用されることを示します。*attribute-target-specifier* が **field** の場合は、この属性がフィールドに適用されることを示します。*attribute-target-specifier* が **method** の場合は、この属性がメソッドに適用されることを示します。
- プロパティまたはインデクサーの宣言の **get** アクセサー宣言で指定される属性は、関連付けられたメソッド、またはその戻り値に適用できます。*attribute-target-specifier* が指定されない場合、属性はメソッドに適用されます。*attribute-target-specifier* が **method** の場合は、この属性がメソッドに適用されることを示します。*attribute-target-specifier* が **return** の場合は、この属性が戻り値に適用されることを示します。
- プロパティまたはインデクサーの宣言の **set** アクセサーで指定される属性は、関連付けられたメソッド、またはその唯一のパラメーターに適用できます。*attribute-target-specifier* が指定されない場合、属性はメソッドに適用されます。*attribute-target-specifier* が **method** の場合は、この属性がメソッドに適用されることを示します。*attribute-target-specifier* が **param** の場合は、この属性がパラメーターに適用されることを示します。*attribute-target-specifier* が **return** の場合は、この属性が戻り値に適用されることを示します。

- イベント宣言の add アクセサーまたは remove アクセサーの宣言で指定される属性は、関連付けられたメソッド、またはその唯一のパラメーターに適用できます。*attribute-target-specifier* が指定されない場合、属性はメソッドに適用されます。*attribute-target-specifier* が *method* の場合は、この属性がメソッドに適用されることを示します。*attribute-target-specifier* が *param* の場合は、この属性がパラメーターに適用されることを示します。*attribute-target-specifier* が *return* の場合は、この属性が戻り値に適用されることを示します。

他のコンテキストでは、*attribute-target-specifier* を含めることはできますが不要です。たとえば、クラス宣言では、型指定子 *type* を含めても省略してもかまいません。

```
[type: Author("Brian Kernighan")]
class Class1 {}

[Author("Dennis Ritchie")]
class Class2 {}
```

無効な *attribute-target-specifier* を指定するとエラーになります。たとえば、*param* 指定子は、クラス宣言では使用できません。

```
[param: Author("Brian Kernighan")] // Error
class Class1 {}
```

規約により、属性クラスの名前には **Attribute** というサフィックスを付けます。*type-name* 形式の *attribute-name* では、このサフィックスを含めることも省略することもできます。サフィックス付きとサフィックスなしの両方の形式の属性クラスが検出された場合は、あいまいさが存在するため、コンパイルエラーになります。*attribute-name* の右端の *identifier* が逐語的識別子(2.4.2を参照)の場合は、サフィックスなしの属性だけが一致するため、あいまいさを解決できます。次の例を参照してください。

```
using System;
[AttributeUsage(AttributeTargets.All)]
public class X: Attribute
{}

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X] // Error: ambiguity
class Class1 {}

[XAttribute] // Refers to XAttribute
class Class2 {}

[@X] // Refers to X
class Class3 {}

[@XAttribute] // Refers to XAttribute
class Class4 {}
```

この例は、*X* および *XAttribute* という名前の 2 つの属性クラスを示しています。*[X]* 属性は、*X* または *XAttribute* のどちらも参照する可能性があるため、あいまいです。逐語的識別子を使用して正確な意図を指定できる場合もあります。*[XAttribute]* 属性は、*XAttributeAttribute!* という属性クラスが存在しない限り、あいまいではありません。*X* クラスの宣言が削除された場合は、次に示すように、両方の属性が *XAttribute* という名前の属性クラスを参照します。

```
using System;
[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}
```

```
[X]           // Refers to XAttribute
class Class1 {}

[XAttribute]   // Refers to XAttribute
class Class2 {}

[@X]          // Error: no attribute named "X"
class Class3 {}
```

1回だけ使用できる属性クラスを同じエンティティで複数回使用すると、コンパイルエラーになります。次の例を参照してください。

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: Attribute
{
    string value;
    public HelpStringAttribute(string value) {
        this.value = value;
    }
    public string Value {
        get {...}
    }
}

[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]
public class Class1 {}
```

この例は、1回だけ使用できる属性クラスである `HelpString` を `Class1` の宣言で複数回使用しているため、コンパイルエラーになります。

次の条件がすべて当てはまる場合、式 `E` は *attribute-argument-expression* です。

- `E` の型が属性パラメータ型 (17.1.3 を参照) である。
- コンパイル時に、`E` の値を次のいずれかに解決できる。
  - 定数値
  - `System.Type` オブジェクト。
  - *attribute-argument-expression* の 1 次元配列

次に例を示します。

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class TestAttribute: Attribute
{
    public int P1 {
        get {...}
        set {...}
    }
    public Type P2 {
        get {...}
        set {...}
    }
}
```

```

public object P3 {
    get {...}
    set {...}
}
[Test(P1 = 1234, P3 = new int[] {1, 3, 5}, P2 = typeof(float))]
class MyClass {}

```

属性引数の式として使用される *typeof-expression* (7.6.11 を参照) は、非ジェネリック型、クローズ構築型、または非バインド ジェネリック型を参照できますが、オープン型は参照できません。これは、コンパイル時に式が確実に解決されるようにするためにです。

```

class A: Attribute
{
    public A(Type t) {...}

class G<T>
{
    [A(typeof(T))] T t;           // Error, open type in attribute

class X
{
    [A(typeof(List<int>))] int x;   // Ok, closed constructed type
    [A(typeof(List<>))] int y;     // Ok, unbound generic type
}

```

## 17.3 属性インスタンス

属性インスタンスは、実行時に属性を表すインスタンスです。属性は、属性クラス、位置指定引数、および名前付き引数で定義されます。属性インスタンスは、位置指定引数と名前付き引数で初期化される、属性クラスのインスタンスです。

属性インスタンスを取得するには、以降の各トピックで説明するように、コンパイル時の処理と実行時の処理の両方が必要です。

### 17.3.1 属性のコンパイル

属性クラス *T*、*positional-argument-list P*、および *named-argument-list N* を持つ *attribute* のコンパイルは、次の手順で行われます。

- *new T(P)* 形式の *object-creation-expression* をコンパイルするための、コンパイル時の処理手順に従います。これらの手順によって、コンパイルエラーが発生するか、または実行時に呼び出すことができる *T* のインスタンス コンストラクター *C* が決定されます。
- *C* が *public* アクセシビリティを持たない場合は、コンパイルエラーが発生します。
- *N* の *named-argument* である *Arg* ごとに、次の処理を実行します。
  - *Name* を *identifier (named-argument Arg.)* にします。
  - *Name* は、*T* の非静的な読み取り/書き込みパブリック フィールドまたはプロパティを識別する必要があります。*T* がそのようなフィールドまたはプロパティを持たない場合は、コンパイルエラーが発生します。
- 属性の実行時インスタンス化のために、属性クラス *T*、*T* のインスタンス コンストラクター *C*、*positional-argument-list P*、*named-argument-list N* の各情報を保持します。

### 17.3.2 属性インスタンスの実行時取得

*attribute* のコンパイルによって、属性クラス *T*、*T* のインスタンス コンストラクター *C*、*positional-argument-list P*、および *named-argument-list N* が作成されます。この情報があると、次の手順を使用して、属性インスタンスを実行時に取得できます。

- コンパイル時に決定されるインスタンス コンストラクター *C* を使用して、`new T(P)` 形式の *object-creation-expression* を実行するための、実行時の処理手順に従います。これらの手順によって、例外が発生するか、または *T* のインスタンス *o* が生成されます。
- *N* の *named-argument* である *Arg* ごとに、次の処理を順番に実行します。
  - *Name* を *identifier (named-argument Arg.)* にします。*Name* が、*o* の非静的でパブリックな読み取り/書き込みフィールドまたはプロパティを識別しない場合は、例外がスローされます。
  - *value* を *attribute-argument-expression* を評価した結果にします (*Arg*)。
  - *Name* が *o* のフィールドを識別する場合は、このフィールドを値 *value* に設定します。
  - それ以外の場合、*Name* は *o* のプロパティを識別します。このプロパティを *value* に設定します。
  - 結果は、*positional-argument-list P* と *named-argument-list N* で初期化された属性クラス *T* のインスタンスである *o* になります。

## 17.4 予約済み属性

いくつかの属性は、何らかの形で言語に影響を与えます。このような属性を次に示します。

- `System.AttributeUsageAttribute` (17.4.1 を参照)。属性クラスの使用方法を説明する属性です。
- `System.Diagnostics.ConditionalAttribute` (17.4.2 を参照)。条件付きメソッドを定義する属性です。
- `System.ObsoleteAttribute` (17.4.3 を参照)。メンバーを旧式のメンバーとしてマークする属性です。
- `System.Runtime.CompilerServices.CallerLineNumberAttribute`、  
`System.Runtime.CompilerServices.CallerFilePathAttribute`、および  
`System.Runtime.CompilerServices.CallerMemberNameAttribute` (17.4.4 を参照)。呼び出し元コンテキストに関する情報を省略可能なパラメータに提供する属性です。

### 17.4.1 AttributeUsage 属性

`AttributeUsage` 属性は、属性クラスの使用方法を説明するために使用します。

`AttributeUsage` 属性で修飾されるクラスは、`System.Attribute` から直接または間接的に派生する必要があります。それ以外の場合は、コンパイル時のエラーが発生します。

```
namespace System
{
    [AttributeUsage(AttributeTargets.Class)]
    public class AttributeUsageAttribute : Attribute
    {
        public AttributeUsageAttribute(AttributeTargets validon) {...}
        public virtual bool AllowMultiple { get {...} set {...} }
        public virtual bool Inherited { get {...} set {...} }
```

```

        public virtual AttributeTargets ValidOn { get {...} }

    public enum AttributeTargets
    {
        Assembly      = 0x0001,
        Module        = 0x0002,
        Class         = 0x0004,
        Struct         = 0x0008,
        Enum           = 0x0010,
        Constructor   = 0x0020,
        Method         = 0x0040,
        Property       = 0x0080,
        Field          = 0x0100,
        Event          = 0x0200,
        Interface     = 0x0400,
        Parameter      = 0x0800,
        Delegate       = 0x1000,
        ReturnValue    = 0x2000,
        All = Assembly | Module | Class | Struct | Enum | Constructor |
              Method | Property | Field | Event | Interface | Parameter |
              Delegate | ReturnValue
    }
}

```

## 17.4.2 Conditional 属性

`Conditional`\b 属性により、**条件付きメソッド**および**条件付き属性クラス**を定義できるようになります。

```

namespace System.Diagnostics
{
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
                    AllowMultiple = true)]
    public class ConditionalAttribute: Attribute
    {
        public ConditionalAttribute(string conditionString) {...}
        public string ConditionString { get {...} }
    }
}

```

### 17.4.2.1 条件付きメソッド

`Conditional` 属性を使って修飾されたメソッドは条件付きメソッドです。`Conditional` 属性は、条件付きコンパイルシンボルをテストすることによって条件を示します。条件付きメソッドの呼び出しは、呼び出しが行われた時点でこのシンボルが定義されているかどうかによって、実際に含まれるか省略されるかが決まります。シンボルが定義されている場合は呼び出しが行われ、定義されていない場合は呼び出し(呼び出しのレシーバーおよびパラメーターの評価も含む)が行われません。

条件付きメソッドには、次の制限事項があります。

- 条件付きメソッドは、*class-declaration* または *struct-declaration* のメソッドである必要があります。インターフェイス宣言のメソッドで `Conditional` 属性が指定されていると、コンパイルエラーになります。
- 条件付きメソッドは、`void` 型の戻り値を持つ必要があります。
- 条件付きメソッドには、`override` 修飾子を設定できません。ただし、条件付きメソッドに `virtual` 修飾子を設定することはできます。この修飾子を持つメソッドのオーバーライドは暗黙的に条件付きであり、明示的に `Conditional` 属性を設定することはできません。

- 条件付きメソッドには、インターフェイス メソッドの実装は使用できません。それ以外の場合は、コンパイル時のエラーが発生します。

また、条件付きメソッドを *delegate-creation-expression* で使用すると、コンパイル エラーが発生します。次の例を参照してください。

```
#define DEBUG
using System;
using System.Diagnostics;
class Class1
{
    [Conditional("DEBUG")]
    public static void M()
    {
        Console.WriteLine("Executed Class1.M");
    }
}
class Class2
{
    public static void Test()
    {
        Class1.M();
    }
}
```

この例では、`Class1.M` を条件付きメソッドとして宣言します。`Class2` の `Test` メソッドがこのメソッドを呼び出します。条件付きコンパイルシンボル `DEBUG` が定義されているため、`Class2.Test` が呼び出されると、`M` が呼び出されます。シンボル `DEBUG` が定義されていなかった場合、`Class2.Test` は `Class1.M` を呼び出しません。

条件付きメソッドの呼び出しが含まれるか除外されるかは、呼び出しの時点での条件付きコンパイルシンボルによって制御されます。次に例を示します。

```
ファイル class1.cs
using System.Diagnostics;
class Class1
{
    [Conditional("DEBUG")]
    public static void F()
    {
        Console.WriteLine("Executed Class1.F");
    }
}
ファイル class2.cs
#define DEBUG
class Class2
{
    public static void G()
    {
        Class1.F();           // F is called
    }
}
ファイル class3.cs
#undef DEBUG
```

```
class Class3
{
    public static void H()
    {
        Class1.F(); // F is not called
    }
}
```

この例のクラス `Class2` と `Class3` には、それぞれ条件付きメソッド `Class1.F` の呼び出しが設定されています。このメソッドの呼び出しは、`DEBUG` が定義されているかどうかによって決まります。このシンボルは、`Class2` のコンテキストでは定義されていますが `Class3` では定義されていません。このため、`Class2` での `F` の呼び出しが行われますが、`Class3` での `F` の呼び出しが行われません。

継承チェーン内で条件付きメソッドを使用すると、混乱する可能性があります。`base.M` の形式の条件付きメソッドを `base` を使用して呼び出す場合は、通常の条件付きメソッドの呼び出し規約に従います。次に例を示します。

ファイル `Class1.cs`

```
using System;
using System.Diagnostics;
class Class1
{
    [Conditional("DEBUG")]
    public virtual void M()
    {
        Console.WriteLine("Class1.M executed");
    }
}
```

ファイル `Class2.cs`

```
using System;
class Class2: Class1
{
    public override void M()
    {
        Console.WriteLine("Class2.M executed");
        base.M(); // base.M is not called!
    }
}
```

ファイル `Class3.cs`

```
#define DEBUG
using System;
class Class3
{
    public static void Test()
    {
        Class2 c = new Class2();
        c.M(); // M is called
    }
}
```

`Class2` には、基底クラスで定義されている `M` の呼び出しが含まれています。基本メソッドはシンボル `DEBUG` の存在に基づく条件付きメソッドですが、このシンボルは未定義であるため、この呼び出しが省略されます。このため、メソッドはコンソールに `"Class2.M executed"` と書き込むだけです。*pp-declaration* を効果的に使用することで、この問題を解決できます。

### 17.4.2.2 条件付き属性クラス

1つ以上の `Conditional` 属性を使って修飾された属性クラス (17.1 を参照) は "条件付き属性クラス" です。条件付き属性クラスは、その `Conditional` 属性に宣言されている条件付きコンパイルシンボルに関連付けられています。次の例を参照してください。

```
using System;
using System.Diagnostics;
[Conditional("ALPHA")]
[Conditional("BETA")]
public class TestAttribute : Attribute {}
```

この例は、`TestAttribute` を、条件付きコンパイルシンボル `ALPHA` および `BETA` に関連付けられた条件付き属性クラスとして宣言します。

指定の時点で、関連付けられた条件付きコンパイルシンボルが 1つ以上定義されている場合、条件付き属性の属性の指定 (17.2 を参照) が含まれます。定義されていない場合、属性の指定は省略されます。

条件付き属性クラスの属性の指定が含まれるか除外されるかは、指定の時点での条件付きコンパイルシンボルによって制御されます。次に例を示します。

```
ファイル test.cs
using System;
using System.Diagnostics;
[Conditional("DEBUG")]
public class TestAttribute : Attribute {}

ファイル class1.cs
#define DEBUG
[Test]           // TestAttribute is specified
class class1 {}

ファイル class2.cs
#undef DEBUG
[Test]           // TestAttribute is not specified
class class2 {}
```

この例で、クラス `Class1` および `Class2` はそれぞれ属性 `Test` で修飾されます。この属性は `DEBUG` が定義されているかどうかに基づく条件付き属性です。このシンボルは、`Class1` のコンテキストでは定義されていますが `Class2` では定義されていません。このため、`Class1` での `Test` 属性の指定は行われますが、`Class2` での `Test` 属性の指定は行われません。

### 17.4.3 Obsolete 属性

`Obsolete` 属性は、使用されなくなった型および型のメンバーをマークします。

```

namespace System
{
    [AttributeUsage(
        AttributeTargets.Class | 
        AttributeTargets.Struct | 
        AttributeTargets.Enum | 
        AttributeTargets.Interface | 
        AttributeTargets.Delegate | 
        AttributeTargets.Method | 
        AttributeTargets.Constructor | 
        AttributeTargets.Property | 
        AttributeTargets.Field | 
        AttributeTargets.Event,
        Inherited = false)
    ]
    public class ObsoleteAttribute: Attribute
    {
        public ObsoleteAttribute() {...}
        public ObsoleteAttribute(string message) {...}
        public ObsoleteAttribute(string message, bool error) {...}
        public string Message { get {...} }
        public bool IsError { get {...} }
    }
}

```

プログラムが **Obsolete** 属性で修飾された型やメンバーを使用している場合、コンパイラは警告またはエラーを通知します。たとえば、エラーパラメーターが提供されない場合や、エラーパラメーターが提供されてもその値が **false** である場合、コンパイラは警告を通知します。エラーパラメーターが指定され、その値が **true** である場合、コンパイラはエラーを通知します。

次に例を示します。

```

[Obsolete("This class is obsolete; use class B instead")]
class A
{
    public void F() {}
}

class B
{
    public void F() {}
}

class Test
{
    static void Main()
    {
        A a = new A();           // warning
        a.F();
    }
}

```

この例では、クラス **A** が **Obsolete** 属性で修飾されています。**Main** で **A** を使用するたびに、指定されたメッセージ "This class is obsolete; use class B instead" を含む警告が表示されます。

#### 17.4.4 呼び出し元情報属性

ログやレポートなどの目的では、関数メンバーが呼び出し元のコードに関する特定のコンパイル時の情報を取得すると便利な場合があります。呼び出し元情報属性は、そのような情報を透過的に渡す方法を提供します。

省略可能なパラメーターにいずれかの呼び出し元情報属性で注釈が付けられている場合、呼び出しで対応する引数を省略しても、既定のパラメーター値が置き換えられるとは限りません。代わりに、呼び出し元コンテキストの指定した情報が使用できる場合は、その情報が引数値として渡されます。

次に例を示します。

```
using System.Runtime.CompilerServices

...
public void Log(
    [CallerLineNumber] int line = -1,
    [CallerFilePath] string path = null,
    [CallerMemberName] string name = null
)
{
    Console.WriteLine((line < 0) ? "No line" : "Line " + line);
    Console.WriteLine((path == null) ? "No file path" : path);
    Console.WriteLine((name == null) ? "No member name" : name);
}
```

引数のない `Log()` の呼び出しでは、呼び出しの行番号とファイルパスだけでなく、呼び出しが発生したメンバーの名前も印刷されます。

呼び出し元情報属性は、デリゲート宣言などの任意の場所の省略可能なパラメーターで使用できます。ただし、特定の呼び出し元情報属性には、起因するパラメーターの型に制限があるため、置換された値からパラメーター型への暗黙の変換が常に存在します。

部分メソッド定義宣言の定義部分と実装部分の両方のパラメーターに同じ呼び出し元情報属性が存在するとエラーになります。定義部分の呼び出し元情報属性のみが適用され、実装部分のみで発生した呼び出し元情報属性は無視されます。

呼び出し元情報はオーバーロードの解決には影響しません。属性付きの省略可能なパラメーターは、依然として呼び出し元のソース コードから省略されるため、オーバーロードの解決では、省略されたその他の省略可能なパラメーターを無視するのと同じ方法でそれらのパラメーターを無視します(7.5.3 を参照)。

呼び出し元情報は、関数がソース コードで明示的に呼び出された場合にのみ置き換えられます。暗黙の親コンストラクター呼び出しなどの暗黙的な呼び出しには、ソースの場所がないため、呼び出し元情報は置き換えられません。また、動的にバインドされる呼び出しでは、呼び出し元情報は置き換えられません。呼び出し元情報属性付きパラメーターがこのような場合に省略されると、パラメーターの指定した既定値が代わりに使用されます。

クエリ式だけが異なります。これらは構文の展開と見なされ、展開される呼び出しが呼び出し元情報属性を持つ省略可能なパラメーターを省略すると、呼び出し元情報が置き換えられます。使用される場所は、呼び出しが生成されたクエリ句の場所です。

特定のパラメーターに複数の呼び出し元情報属性が指定されている場合は、`CallerLineNumber`、`CallerFilePath`、`CallerMemberName` の順序で優先されます。

#### 17.4.4.1 CallerLineNumber 属性

`System.Runtime.CompilerServices.CallerLineNumberAttribute` は、定数値 `int.MaxValue` からパラメーターの型への標準暗黙変換(6.3.1 を参照)がある場合に、省略可能なパラメーターで使用できます。これにより、その値までの負でない行番号をエラーなしで渡すことができます。

ソース コード内の場所からの関数の呼び出しで `CallerLineNumberAttribute` を持つ省略可能なパラメーターが省略されると、その場所の行番号を表す数値リテラルが、既定のパラメーター値の代わりに呼び出しの引数として使用されます。

呼び出しが複数の行にまたがる場合、選択された行は実装に依存します。

行番号は `#line` ディレクティブ (2.5.7 を参照) の影響を受ける場合があります。

#### 17.4.4.2 CallerFilePath 属性

`System.Runtime.CompilerServices.CallerFilePathAttribute` は、`string` からパラメーターの型への標準暗黙変換 (6.3.1 を参照) がある場合に、省略可能なパラメーターで使用できます。

ソース コード内の場所からの関数の呼び出しで `CallerFilePathAttribute` を持つ省略可能なパラメーターが省略されると、その場所のファイルパスを表すリテラル文字列が、既定のパラメーター値の代わりに呼び出しの引数として使用されます。

ファイルパスの形式は実装に依存します。

ファイルパスは `#line` ディレクティブ (2.5.7 を参照) の影響を受ける場合があります。

#### 17.4.4.3 CallerMemberName 属性

`System.Runtime.CompilerServices.CallerMemberNameAttribute` は、`string` からパラメーターの型への標準暗黙変換 (6.3.1 を参照) がある場合に、省略可能なパラメーターで使用できます。

関数メンバーの本体内または属性内の場所からの関数の呼び出しが関数メンバー自体またはその戻り値の型に適用されると、ソース コード内のパラメーターまたは型パラメーターは

`CallerMemberNameAttribute` を持つ省略可能なパラメーターを省略し、そのメンバーの名前を表すリテラル文字列が既定のパラメーター値の代わりに呼び出しの引数として使用されます。

ジェネリック メソッド内で発生する呼び出しの場合、型パラメーター リストなしでメソッド名のみが使用されます。

明示的なインターフェイス メンバー実装内で発生する呼び出しの場合、先行するインターフェイス 修飾なしでメソッド名のみが使用されます。

プロパティまたはイベント アクセサー内で発生する呼び出しの場合、使用されるメンバーネームはプロパティまたはイベント自体の名前です。

インデクサー アクセサー内で発生する呼び出しの場合、使用されるメンバーネームは、インデクサー メンバー (存在する場合) の `IndexerNameAttribute` (17.5.2.1 を参照) が提供する名前か、そうでなければ既定値 `Item` です。

インスタンス コンストラクター、静的コンストラクター、デストラクター、および演算子の宣言内で発生する呼び出しの場合、使用されるメンバーネームは実装に依存します。

### 17.5 相互運用の属性

メモ：このセクションの内容は、Microsoft .NET の C# の実装だけに適用されます。

#### 17.5.1 COM コンポーネントと Win32 コンポーネントとの相互運用

.NET ランタイムには、C# プログラムが COM と Win32 の DLL を使用して記述されたコンポーネントと相互運用できるように、多数の属性が用意されています。たとえば、`static extern` メソッドで `DllImport` 属性を使用すると、メソッドの実装が Win32 DLL に存在することを示すことができます。

これらの属性は、`System.Runtime.InteropServices` 名前空間にあり、.NET ランタイム ドキュメントで詳しく説明しています。

## 17.5.2 他の .NET 言語との相互運用

### 17.5.2.1 IndexerName 属性

インデクサーは、インデックス付きプロパティを使用して .NET に実装されます。インデクサーの名前は、.NET メタデータに格納されます。インデクサーに `IndexerName` 属性がない場合、既定では名前 `Item` が使用されます。`IndexerName` 属性を使用すると、開発者がこの既定値をオーバーライドして、別の名前を指定できます。

```
namespace System.Runtime.CompilerServices.CSharp
{
    [AttributeUsage(AttributeTargets.Property)]
    public class IndexerNameAttribute: Attribute
    {
        public IndexerNameAttribute(string indexerName) {...}
        public string Value { get {...} }
    }
}
```

# 18. 安全でないコード

前の章で定義されたように、コアの C# 言語は、データ型としてのポインターを省略した点で、C および C++ とは大きく異なります。代わりに、C# には、参照、およびガベージコレクターによって管理されるオブジェクトを作成する機能があります。このデザインと他の機能を組み合わせることによって、C# は、C や C++ よりも高い安全性を持つ言語になっています。コアの C# 言語では、初期化されていない変数、参照先がないポインター、または配列の境界を越えてインデックスを作成する式を使用できません。このため、C および C++ のプログラムに発生しがちな、すべての種類のバグが排除されます。

C または C++ で構築されるほぼすべてのポインター型については、対応する参照型が C# に存在します。しかし、C# でもポインター型へのアクセスが必要になる状況があります。たとえば、基になるオペレーティングシステムとのインターフェイス、メモリ マップ デバイスへのアクセス、またはタイムクリティカルなアルゴリズムの実装を行う場合は、ポインターへのアクセスを使用しなければ不可能であるか、または実用的ではありません。この必要性に対処するために、C# には **安全でないコード** を記述する機能が用意されています。

安全でないコードを使用すると、ポインターの宣言および操作、ポインターと整数型との間の変換、変数のアドレスの取得などが可能です。安全でないコードを記述することは、ある意味で、C# プログラム内に C コードを記述するのと似ています。

実際、安全でないコードは、開発者とユーザーの両方の立場から見て "安全" な機能です。安全でないコードは、**unsafe** 修飾子で明確にマークする必要があります。このため、開発者が安全でない機能を誤って使用する可能性はありません。また、実行エンジンによって、安全でないコードを信頼性のない環境で実行することはできません。

## 18.1 Unsafe コンテキスト

C# の unsafe 機能は、"unsafe コンテキスト" だけで利用できます。unsafe コンテキストは、型またはメンバーの宣言に unsafe 修飾子を含めるか、または unsafe-statement を使用することで導入されます。

- **unsafe** 修飾子は、クラス、構造体、インターフェイス、またはデリゲートの宣言に含めることができます。この場合、その型宣言の範囲全体(クラス、構造体、またはインターフェイスの本体を含む)が、unsafe コンテキストと見なされます。
- **unsafe** 修飾子は、フィールド、メソッド、プロパティ、イベント、インデクサー、演算子、インスタンス コンストラクター、デストラクター、または静的コンストラクターの宣言に含めることができます。この場合、そのメンバー宣言の範囲全体が、unsafe コンテキストと見なされます。
- **unsafe-statement** を使用すると、block 内で unsafe コンテキストを使用できます。関連する block の範囲全体が、unsafe コンテキストと見なされます。

関連する文法拡張を次に示します。表記を簡略にするために、省略記号 (...) を使用して、これまでの章で示された要素を表しています。

*class-modifier:*

...  
**unsafe**

*struct-modifier:*

...  
**unsafe**

*interface-modifier:*

...  
**unsafe**

*delegate-modifier:*

...  
**unsafe**

*field-modifier:*

...  
**unsafe**

*method-modifier:*

...  
**unsafe**

*property-modifier:*

...  
**unsafe**

*event-modifier:*

...  
**unsafe**

*indexer-modifier:*

...  
**unsafe**

*operator-modifier:*

...  
**unsafe**

*constructor-modifier:*

...  
**unsafe**

*destructor-declaration:*

$\text{attributes}_{\text{opt}} \text{ extern}_{\text{opt}} \text{ unsafe}_{\text{opt}} \sim \text{identifier} (\ ) \text{ destructor-body}$   
 $\text{attributes}_{\text{opt}} \text{ unsafe}_{\text{opt}} \text{ extern}_{\text{opt}} \sim \text{identifier} (\ ) \text{ destructor-body}$

*static-constructor-modifiers:*

**extern**<sub>opt</sub> **unsafe**<sub>opt</sub> **static**  
**unsafe**<sub>opt</sub> **extern**<sub>opt</sub> **static**  
**extern**<sub>opt</sub> **static** **unsafe**<sub>opt</sub>  
**unsafe**<sub>opt</sub> **static** **extern**<sub>opt</sub>  
**static** **extern**<sub>opt</sub> **unsafe**<sub>opt</sub>  
**static** **unsafe**<sub>opt</sub> **extern**<sub>opt</sub>

*embedded-statement:*

...

*unsafe-statement*

*unsafe-statement:*  
    **unsafe** *block*

次に例を示します。

```
public unsafe struct Node
{
    public int value;
    public Node* Left;
    public Node* Right;
}
```

構造体宣言で **unsafe** 修飾子を指定すると、構造体宣言の範囲全体が **unsafe** コンテキストになります。このため、**Left** フィールドと **Right** フィールドをポインター型として宣言できます。上の例は次のようにも記述できます。

```
public struct Node
{
    public int value;
    public unsafe Node* Left;
    public unsafe Node* Right;
}
```

この例では、フィールドの宣言に **unsafe** 修飾子が指定されているため、これらの宣言が **unsafe** コンテキストと見なされます。

**unsafe** コンテキストを設定する以外にも、この方法でポインター型の使用を許可すると、**unsafe** 修飾子は型やメンバーに影響を及ぼしません。次に例を示します。

```
public class A
{
    public unsafe virtual void F() {
        char* p;
        ...
    }
}
public class B: A
{
    public override void F() {
        base.F();
        ...
    }
}
```

**A** の **F** メソッドで **unsafe** 修飾子を指定すると、**F** の範囲が **unsafe** コンテキストになります。この **unsafe** コンテキストでは、言語の **unsafe** 機能を使用できます。**B** の **F** メソッド自体が **unsafe** 機能へのアクセスを必要としない限り、**B** の **F** のオーバーライドで **unsafe** 修飾子を再指定する必要はありません。

ポインター型がメソッドのシグネチャに含まれる場合は、状況が少し異なります。

```
public unsafe class A
{
    public virtual void F(char* p) {...}
```

```
public class B: A
{
    public unsafe override void F(char* p) {...}
```

この例では、`F` のシグネチャにポインター型が含まれるため、`unsafe` コンテキストでだけ記述できます。ただし、`A` の場合のようにクラス全体を `unsafe` にするか、`B` の場合のようにメソッド宣言に `B` 修飾子を含めることによって、`unsafe` コンテキストを導入できます。

## 18.2 ポインター型

`unsafe` コンテキストでは、`type` (4 を参照) は *value-type* や *reference-type* だけでなく *pointer-type* することもできます。ただし、*pointer-type* は `unsafe` コンテキストの外側の `typeof` 式 (7.6.10.6 を参照) に使用される場合もあります。このような使用は `unsafe` ではありません。

*type*:

```
...
pointer-type
```

*pointer-type* は、*unmanaged-type* またはキーワード `void` の後に `*` トークンを続ける形式で記述します。

*pointer-type*:

```
unmanaged-type *
void *
```

*unmanaged-type*:

```
type
```

ポインター型の `*` の前に指定された型は、ポインター型の *参照先の型* と呼ばれます。これは、ポインター型の値が指す変数の型を表します。

参照型の値である参照とは異なり、ポインターはガベージコレクターによって追跡されません。ガベージコレクターは、ポインターが指すデータについては認識しません。このため、ポインターは、参照または参照を含む構造体を指すことができません。また、ポインターの参照先の型は *unmanaged-type* である必要があります。

*unmanaged-type* は、*reference-type* または構築された型ではない型であり、どの入れ子レベルにも *reference-type* フィールドまたは構築された型フィールドを含みません。つまり、*unmanaged-type* は次のいずれかになります。

- `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、`decimal`、または `bool`。
- 任意の *enum-type* 型。
- 任意の *pointer-type*
- 構築された型ではなく、*unmanaged-type* のフィールドだけを格納する、ユーザー定義の任意の *struct-type*。

ポインターと参照を混在させる場合に明白な規則は、参照の参照先 (オブジェクト) ではポインターを含めることができても、ポインターの参照先では参照を含めることができないことです。

ポインター型の例を次の表に示します。

例	説明
<code>byte*</code>	<code>byte</code> へのポインター
<code>char*</code>	<code>char</code> へのポインター
<code>int**</code>	<code>int</code> へのポインターのポインター
<code>int*[]</code>	<code>int</code> へのポインターの 1 次元配列
<code>void*</code>	未知の型へのポインター

所定の実装では、すべてのポインター型が同じサイズおよび表現を持つ必要があります。

C および C++ とは異なり、複数のポインターが同じ宣言内で宣言された場合、C# での \* は、各ポインター名のプレフィックス区切り記号としてではなく、基になる型と共にだけ記述されます。次に例を示します。

```
int* pi, pj; // NOT as int *pi, *pj;
```

`T*` 型を持つポインターの値は、`T` 型の変数の *address* を表します。ポインター間接演算子 \* (18.5.1 を参照) を使用すると、この変数にアクセスできます。たとえば、`int*` 型の変数 `P` の場合、式 `*P` は、`P` に格納されたアドレスにある `int` 変数を表しています。

ポインターには、オブジェクト参照と同様に、`null` を設定できます。`null` ポインターに間接演算子を適用すると、実装で定義されている動作が発生します。`null` 値を持つポインターは、すべてのビットが 0 の値で表されます。

`void*` 型は、未知の型へのポインターを表します。参照型は未知であるため、`void*` 型のポインターに対しては間接演算子を適用できません。また、どのような算術も実行できません。ただし、`void*` 型のポインターは他のポインター型にキャストできます(その逆も可能です)。

ポインター型は、型の独立したカテゴリです。参照型や値型とは異なり、ポインター型は `object` からは継承せず、ポインター型と `object` との間の変換は存在しません。特に、ボックス化とボックス化解除 (4.3 を参照) はポインターではサポートされていません。ただし、異なるポインター型の間の変換、およびポインター型と整数型との間の変換は可能です。これは 18.4 で説明します。

`pointer-type` は型引数 (4.4 を参照) として使用できず、型引数がポインター型であると推論されたジェネリック メソッド呼び出しでは型推論 (7.5.2 を参照) が失敗します。

`pointer-type` は、動的にバインドされる操作 (7.2.2 を参照) の構成する式の型として使用できません。

`pointer-type` は `volatile` フィールド (10.5.3 を参照) の型として使用されることがあります。

ポインターを `ref` パラメーターや `out` パラメーターとして渡すことはできますが、これを行うと、不定の動作が発生することがあります。これは、呼び出されたメソッドから制御が戻る時点で存在していないローカル変数を指すようにポインターが設定されているか、またはポインターが指していた固定オブジェクトが既に固定されていないことが原因です。次に例を示します。

```
using System;
class Test
{
    static int value = 20;
```

```

unsafe static void F(out int* pi1, ref int* pi2) {
    int i = 10;
    pi1 = &i;
    fixed (int* pj = &value) {
        // ...
        pi2 = pj;
    }
}

static void Main() {
    int i = 10;
    unsafe {
        int* px1;
        int* px2 = &i;
        F(out px1, ref px2);
        Console.WriteLine("*px1 = {0}, *px2 = {1}",
            *px1, *px2); // undefined behavior
    }
}
}

```

メソッドはいくつかの型の値を返すことができますが、その型がポインター型である場合もあります。たとえば、連続した `int` 値へのポインター、その連続した値の要素数、および他の `int` 値が指定されたとします。この場合、次のメソッドは、(一致する値があると) そのシーケンス内で条件に一致した値のアドレスを返します。それ以外の場合は `null` を返します。

```

unsafe static int* Find(int* pi, int size, int value) {
    for (int i = 0; i < size; ++i) {
        if (*pi == value)
            return pi;
        ++pi;
    }
    return null;
}

```

`unsafe` コンテキストでは、ポインターの操作で以下を利用できます。

- `*` 演算子を使用すると、ポインターの間接参照を実行できます (18.5.1 を参照)。
- `->` 演算子を使用すると、ポインターを通じて構造体のメンバーにアクセスできます (18.5.2 を参照)。
- `[]` 演算子を使用すると、ポインターのインデックス設定を実行できます (18.5.3 を参照)。
- `&` 演算子を使用すると、変数のアドレスを取得できます (18.5.4 を参照)。
- `++` 演算子および `--` 演算子を使用すると、ポインターをインクリメントおよびデクリメントできます (18.5.5 を参照)。
- `+` 演算子および `-` 演算子を使用すると、ポインターの算術を実行できます (18.5.6 を参照)。
- `==`、`!=`、`<`、`>`、`<=`、`=>` の各演算子を使用すると、ポインターを比較できます (18.5.7 を参照)。
- `stackalloc` 演算子を使用すると、コール スタックからメモリを割り当てることができます (18.7 を参照)。
- `fixed` ステートメントを使用すると、変数を一時的に固定して、そのアドレスを取得できます (18.6 を参照)。

### 18.3 固定変数と移動可能変数

アドレス演算子 (18.5.4 を参照) および **fixed** ステートメント (18.6 を参照) は、変数を "固定変数" と "移動可能変数" の 2 つのカテゴリに分割します。

固定変数は、ガベージコレクターの処理による影響を受けない格納場所に存在します。固定変数の例としては、ローカル変数、値パラメーター、逆参照ポインターによって作成された変数などがあります。これに対して、移動可能変数は、ガベージコレクターによって再配置されたり破棄されたりする格納場所に存在します。移動可能変数の例としては、オブジェクト内のフィールドや配列の要素などがあります。

& 演算子 (18.5.4 を参照) を使用すると、固定変数のアドレスを制限なしに取得できます。しかし、移動可能変数はガベージコレクターによる再配置や破棄の対象となるため、移動可能変数のアドレスを取得するには、**fixed** ステートメント (18.6 を参照) を使用する必要があります。この場合、取得したアドレスは、その **fixed** ステートメントの存続期間だけ有効です。

正確には、固定変数は次のいずれかになります。

- ローカル変数または値パラメーターを参照する *simple-name* (7.6.2 を参照) の結果である変数。ただし、匿名関数によってキャプチャされた変数は除きます。
- **v.I** 形式の *member-access* (7.6.4 を参照) の結果である変数。 **v** は *struct-type* の固定変数です。
- **\*P** 形式の *pointer-indirection-expression* (18.5.1 を参照)、**P->I** 形式の *pointer-member-access* (18.5.2 を参照)、**P[E]** 形式の *pointer-element-access* (18.5.3 を参照) のいずれかの結果である変数。

その他すべての変数は、移動可能変数に分類されます。

静的フィールドは、移動可能変数に分類されます。また、**ref** パラメーターや **out** パラメーターは、パラメーターに指定される引数が固定変数である場合でも、移動可能変数に分類されます。ポインターの逆参照によって生成される変数は、常に固定変数に分類されます。

### 18.4 ポインター変換

unsafe コンテキストでは、利用できる暗黙の変換 (0 を参照) に、次の暗黙のポインター変換が含まれます。

- 任意の *pointer-type* 型から **void\*** 型への変換
- **null** リテラルから任意の *pointer-type* への変換

unsafe コンテキストでは、利用できる明示的な変換 (6.2 を参照) に、次の明示的なポインター変換が含まれます。

- 任意の *pointer-type* から他の任意の *pointer-type* への変換
- **sbyte**、**byte**、**short**、**ushort**、**int**、**uint**、**long**、または **ulong** から任意の *pointer-type* への変換
- 任意の *pointer-type* から **sbyte**、**byte**、**short**、**ushort**、**int**、**uint**、**long**、または **ulong** への変換

さらに、unsafe コンテキストでは、標準の暗黙の変換 (6.3.1 を参照) に、次のポインター変換が含まれます。

- 任意の *pointer-type* 型から **void\*** 型への変換

- `null` リテラルから任意の *pointer-type* への変換

2 つのポインター型間の変換によって、実際のポインター値が変更されることはありません。つまり、あるポインター型から別のポインター型に変換しても、ポインターによって指定された基になるアドレスに影響はありません。

あるポインター型を別のポインター型に変換したときに、結果のポインターがポイントされる型に正しく位置合わせされない場合は、結果のポインターが逆参照されると動作が不定になります。“正しく位置合わせする”という概念は、通常は推移的です。型 **A** へのポインターが型 **B** へのポインターに正しく位置合わせされ、型 **B** へのポインターが型 **C** へのポインターに正しく位置合わせされる場合、型 **A** へのポインターは型 **C** へのポインターに正しく位置合わせされます。

ある型の変数が、別の型へのポインターを使用してアクセスされる場合について考えてみます。

```
char c = 'A';
char* pc = &c;
void* pv = pc;
int* pi = (int*)pv;
int i = *pi;      // undefined
*pi = 123456;    // undefined
```

あるポインター型が、バイトを指すポインターに変換された場合、結果は変数のアドレスの最下位バイトを指します。結果を変数のサイズだけ連続してインクリメントすると、その変数の残りのバイトへのポインターがそれぞれ生成されます。たとえば、次のメソッドは、`double` 型の 8 バイトをそれぞれ 16 進数の値として表示します。

```
using System;
class Test
{
    unsafe static void Main() {
        double d = 123.456e23;
        unsafe {
            byte* pb = (byte*)&d;
            for (int i = 0; i < sizeof(double); ++i)
                Console.Write("{0:X2} ", *pb++);
            Console.WriteLine();
        }
    }
}
```

生成される出力は、エンディアンに依存します。

ポインターと整数との間のマップは、実装で定義されます。ただし、リニアアドレス空間を持つ 32 ビット CPU アーキテクチャと 64 ビット CPU アーキテクチャでは、ポインターと整数型との間の変換は、`uint` 値または `ulong` 値と整数型との間の変換まったく同じように処理されます。

#### 18.4.1 ポインター配列

`unsafe` コンテキストでは、ポインターの配列を作成できます。ポインター配列では、他の配列型に適用される一部の変換のみが使用できます。

- 任意の *array-type* から、`System.Array` およびそれによって実装されるインターフェイスへの暗黙の参照変換 (6.1.6 を参照) も、ポインター配列に適用されます。ただし、`System.Array` またはそれによって実装されるインターフェイスを使用して配列要素にアクセスしようとすると、実行時に必ず例外が発生します。これは、ポインター型が `object` に変換できないためです。

- 1 次元配列型 `s[]` から、`System.Collections.Generic.IList<T>` およびそのジェネリックな基本インターフェイスへの暗黙的および明示的な参照変換 (§ 6.1.6、§ 6.2.4 を参照) は、ポインター配列には適用されません。これは、ポインター型が型引数として使用できず、さらにポインター型から非ポインター型への変換が存在しないためです。
- `System.Array` およびそれによって実装されるインターフェイスから、任意の `array-type` への明示的な参照変換 (6.2.4 を参照) は、ポインター配列に適用されます。
- `System.Collections.Generic.IList<S>` およびその基本インターフェイスから、1 次元配列型 `T[]` への明示的な参照変換 (6.2.4 を参照) は、ポインター配列には適用されません。これは、ポインター型が型引数として使用できず、さらにポインター型から非ポインター型への変換が存在しないためです。

このような制限から、8.8.4 で解説した `foreach` ステートメントの展開はポインター配列に適用されません。代わりに、`foreach` ステートメントは次の形式になります。

`foreach (v v in x) embedded-statement`

ここで、`x` の型は `T[, , … , ]` という形式の配列型、`n` は次元数から 1 を減じた数、`T` または `v` はポインター型です。これは、次のように入れ子になった `for` ループに展開されます。

```
{
    T[, , … , ] a = x;
    v v;
    for (int i0 = a.GetLowerBound(0); i0 <= a.GetUpperBound(0); i0++)
    for (int i1 = a.GetLowerBound(1); i1 <= a.GetUpperBound(1); i1++)
    ...
    for (int in = a.GetLowerBound(n); in <= a.GetUpperBound(n); in++) {
        v = (V)a.GetValue(i0, i1, … , in);
        embedded-statement
    }
}
```

変数 `a`、`i0`、`i1`、… `in` は、`x`、`embedded-statement`、またはプログラムのその他のソース コードからは、参照もアクセスもできません。変数 `v` は、埋め込みステートメントでは読み取り専用です。`T` (要素型) から `v` への明示的な変換 (18.4 を参照) がない場合、エラーが発生し、以降の手順は行われません。`x` が `null` 値を持つ場合、実行時に `System.NullReferenceException` がスローされます。

## 18.5 式のポインター

`unsafe` コンテキストでは、式がポインター型の結果を生成することがあります。しかし、`unsafe` コンテキストの外では、式がポインター型を生成するとコンパイル エラーになります。具体的には、`unsafe` コンテキストの外では、`simple-name` (7.6.2 を参照)、`member-access` (7.6.4 を参照)、`invocation-expression` (7.6.5 を参照)、または `element-access` (7.6.6 を参照) のいずれかがポインター型であるとコンパイル エラーになります。

`unsafe` コンテキスト内では、`primary-no-array-creation-expression` (7.6 を参照) と `unary-expression` (7.7 を参照) により、次の構成要素を追加できます。

`primary-no-array-creation-expression:`

```
...
pointer-member-access
pointer-element-access
sizeof-expression
```

*unary-expression*:

...

*pointer-indirection-expression*

*addressof-expression*

これらの構成要素について、次の各セクションで解説します。安全でない演算子の優先順位と結合規則は、文法により暗黙的に設定されます。

### 18.5.1 ポインターの間接参照

*pointer-indirection-expression* は、アスタリスク (\*) と *unary-expression* で構成されます。

*pointer-indirection-expression*:

\* *unary-expression*

単項演算子 \* は、*pointer indirection* を表し、ポインターが指す変数を取得するために使用されます。*P* がポインター型 *T\** の式である場合、\**P* の評価の結果は *T* 型の変数になります。単項演算子 \* を *void\** 型の式、またはポインター型以外の式に適用すると、コンパイルエラーになります。

単項演算子 \* を *null* ポインターに適用した場合の結果は、実装で定義されます。この演算によって **System.NullReferenceException** がスローされるとは限りません。

ポインターに無効な値が設定されている場合、単項演算子 \* の動作は不定になります。単項演算子 \* でポインターを逆参照する場合の無効な値としては、参照先の型と正しく位置合わせされていないアドレス (18.4 の例を参照) や、有効期間が終了した変数のアドレスなどがあります。

代入分析を明確にするため、\**P* 形式の式の評価によって生成された変数は、初期状態で代入されたと見なされます (5.3.1 を参照)。

### 18.5.2 ポインター メンバー アクセス

*pointer-member-access* は、順に、*primary-expression*、"->" トークン、*identifier* および省略可能な *type-argument-list* で構成されます。

*pointer-member-access*:

*primary-expression* -> *identifier* *type-argument-list*<sub>opt</sub>

*P->I* 形式のポインター メンバー アクセスでは、*P* は *void\** 以外のポインター型の式で、*I* は *P* が指す型のアクセス可能なメンバーを表す必要があります。

*P->I* 形式のポインター メンバー アクセスは、(\**P*).*I* として評価されます。ポインター間接演算子 (\*) については、18.5.1 を参照してください。メンバー間接演算子 (.) については、7.6.4 を参照してください。

次に例を示します。

```
using System;
struct Point
{
    public int x;
    public int y;
    public override string ToString() {
        return "(" + x + "," + y + ")";
    }
}
```

```
class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            p->x = 10;
            p->y = 20;
            Console.WriteLine(p->ToString());
        }
    }
}
```

この例では、`->` 演算子を使用して、ポインターによるフィールドへのアクセスや構造体のメソッドの呼び出しが行われます。`P->I` の演算は `(*P).I` とまったく同じであるため、`Main` メソッドは次のように記述することもできます。

```
class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            (*p).x = 10;
            (*p).y = 20;
            Console.WriteLine((*p).ToString());
        }
    }
}
```

### 18.5.3 ポインター要素アクセス

*pointer-element-access* は、*primary-no-array-creation-expression*、および "[" と "]" で囲まれた式で構成されます。

*pointer-element-access*:  
*primary-no-array-creation-expression* [ *expression* ]

`P[E]` 形式のポインター要素アクセスでは、`P` は `void*` 以外のポインター型の式で、`E` は `int`、`uint`、`long`、または `ulong` に暗黙に変換できる式である必要があります。

`P[E]` 形式のポインター要素アクセスは、`*(P + E)` として評価されます。ポインター間接演算子 (\*) については、18.5.1 を参照してください。ポインター加算演算子 (+) については、18.5.6 を参照してください。

次に例を示します。

```
class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) p[i] = (char)i;
        }
    }
}
```

この例では、ポインター要素アクセスを使用して、`for` ループ内で文字バッファーを初期化します。`P[E]` の演算は `*(P + E)` とまったく同じであるため、上の例は次のように記述することもできます。

```

class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) *(p + i) = (char)i;
        }
    }
}

```

ポインター要素アクセス演算子は、範囲外のエラーをチェックしません。また、範囲外の要素にアクセスした場合の動作は未定義です。これは、C と C++ でも同じです。

#### 18.5.4 アドレス演算子

*addressof-expression* は、アンパサンド (&) と *unary-expression* で構成されます。

*addressof-expression:*  
 & *unary-expression*

T型の固定変数(18.3 を参照)として分類される式 E があるとき、&E は E によって指定される変数のアドレスを計算します。結果の型は T\* であり、値として分類されます。E が変数として分類されない場合、E が読み取り専用のローカル変数として分類される場合、または E が移動可能変数を表す場合は、コンパイルエラーが発生します。この最後の場合は、fixed ステートメント(18.6 を参照)を使用して、変数のアドレスを取得する前に変数を一時的に "固定" できます。7.6.4 に記述されているとおり、readonly フィールドを定義する構造体またはクラスのインスタンス コンストラクターまたは静的コンストラクターの外部では、そのフィールドは変数ではなく値と見なされます。この場合、値のアドレスは取得できません。同様に、定数のアドレスも取得できません。

& 演算子では、その引数を明示的に代入する必要はありません。しかし、& 演算で & 演算子が適用される変数は、演算が発生する実行パスで明示的に代入されると見なされます。プログラマは、この状況で変数の適切な初期化が必ず行われるようにする必要があります。

次に例を示します。

```

using System;
class Test
{
    static void Main() {
        int i;
        unsafe {
            int* p = &i;
            *p = 123;
        }
        Console.WriteLine(i);
    }
}

```

この例の i は、p を初期化するための &i 演算に続いて明示的に代入されると見なされます。\*p への代入により、実質的に i が初期化されます。ただし、この初期化はプログラマの責任で含める必要があります、この代入が削除されてもコンパイルエラーは発生しません。

& 演算子で明示的な代入を行うという規則が存在することで、ローカル変数の冗長な初期化を避けることができます。たとえば、外部 API の多くは、その API によって設定される構造体へのポインターを使用します。こうした API への呼び出しでは、通常、ローカル構造体変数のアドレスが渡され、前述の規則がない場合は、構造体変数の冗長な初期化が必要となります。

## 18.5.5 ポインターのインクリメントとデクリメント

unsafe コンテキストでは、`++` 演算子 (7.6.9 を参照) および `--` 演算子 (7.7.5 を参照) は、`void*` を除くすべての型のポインター変数に適用できます。このため、ポインター型 `T*` ごとに、次の演算子が暗黙に定義されます。

```
T* operator ++(T* x);
T* operator --(T* x);
```

演算子の結果はそれぞれ `x + 1` と `x - 1` になります (18.5.6 を参照)。つまり、型 `T*` のポインター変数については、`++` 演算子はその変数に格納されたアドレスに `sizeof(T)` を足し、`--` 演算子はその変数に格納されたアドレスから `sizeof(T)` を引きます。

ポインターのインクリメントまたはデクリメント演算がポインター型の領域をオーバーフローした場合、結果は実装で定義されますが、例外は生成されません。

## 18.5.6 ポインターの算術演算

unsafe コンテキストでは、`+` 演算子 (7.8.4 を参照) および `-` 演算子 (7.8.5 を参照) は、`void*` を除くすべてのポインター型の値に適用できます。このため、ポインター型 `T*` ごとに、次の演算子が暗黙に定義されます。

```
T* operator +(T* x, int y);
T* operator +(T* x, uint y);
T* operator +(T* x, long y);
T* operator +(T* x, ulong y);

T* operator +(int x, T* y);
T* operator +(uint x, T* y);
T* operator +(long x, T* y);
T* operator +(ulong x, T* y);

T* operator -(T* x, int y);
T* operator -(T* x, uint y);
T* operator -(T* x, long y);
T* operator -(T* x, ulong y);

long operator -(T* x, T* y);
```

ポインター型 `T*` の式 `P` と、`int`、`uint`、`long`、`ulong` の各型の式 `N` がある場合、式 `P+N` および式 `N+P` は、`P` で指定されるアドレスに `N * sizeof(T)` を加えて得られる、`T*` 型のポインター値を計算します。同様に、式 `P-N` は、`P` で指定されるアドレスから `N * sizeof(T)` を引いて得られる、`T*` 型のポインター値を計算します。

ポインター型 `T*` の 2 つの式 `P` と `Q` がある場合、式 `P - Q` は、`P` and `Q` で指定されるアドレスの差を計算し、その差を `sizeof(T)` で除算します。結果の型は常に `long` になります。実際には、`P - Q` は `((long)(P) - (long)(Q)) / sizeof(T)` として計算されます。

次に例を示します。

```
using System;
class Test
{
```

```

static void Main() {
    unsafe {
        int* values = stackalloc int[20];
        int* p = &values[1];
        int* q = &values[15];
        Console.WriteLine("p - q = {0}", p - q);
        Console.WriteLine("q - p = {0}", q - p);
    }
}

```

この例では、次のように出力されます。

```

p - q = -14
q - p = 14

```

ポインターの算術演算がポインター型の領域をオーバーフローした場合、結果は実装で定義される方法で切り捨てられますが、例外は生成されません。

### 18.5.7 ポインターの比較

`unsafe` コンテキストでは、`==`、`!=`、`<`、`>`、`<=`、`>=` の各演算子 (7.10 を参照) は、すべてのポインター型の値に適用できます。ポインター比較演算子には次の種類があります。

```

bool operator ==(void* x, void* y);
bool operator !=(void* x, void* y);
bool operator <(void* x, void* y);
bool operator >(void* x, void* y);
bool operator <=(void* x, void* y);
bool operator >=(void* x, void* y);

```

ポインター型から `void*` 型への暗黙の変換が存在するため、ポインター型のオペランドは、これらの演算子を使用して比較できます。比較演算子は、2つのオペランドで指定されるアドレスを符号なし整数として比較します。

### 18.5.8 `sizeof` 演算子

`sizeof` 演算子は、指定された型の変数のバイト数を返します。`sizeof` のオペランドとして指定される型は、*unmanaged-type* (18.2 を参照) である必要があります。

```

sizeof-expression:
    sizeof ( unmanaged-type )

```

`sizeof` 演算子の結果は、`int` 型の値になります。特定の定義済みの型の場合、`sizeof` 演算子は次の表に示す定数値を生成します。

式	結果
<code>sizeof(sbyte)</code>	1
<code>sizeof(byte)</code>	1
<code>sizeof(short)</code>	2
<code>sizeof(ushort)</code>	2
<code>sizeof(int)</code>	4
<code>sizeof(uint)</code>	4
<code>sizeof(long)</code>	8
<code>sizeof(ulong)</code>	8
<code>sizeof(char)</code>	2
<code>sizeof(float)</code>	4
<code>sizeof(double)</code>	8
<code>sizeof(bool)</code>	1

その他のすべての型の場合、`sizeof` 演算子の結果は実装で定義され、定数ではなく値として分類されます。

メンバーが構造体にパックされる順序は不定です。

配置のために、構造体の先頭、内部、および末尾に、無名の埋め込みが行われることがあります。埋め込みとして使用されるビットの内容は不定です。

`struct` 型を持つオペランドに適用される場合、結果は、その型の変数内の(埋め込みを含めた)合計バイト数です。

## 18.6 fixed ステートメント

`unsafe` コンテキストでは、*embedded-statement* (8を参照)によって、追加の構成要素である `fixed` ステートメントを使用できます。このステートメントは、移動可能変数を "固定" して、このステートメントが存続する間、移動可能変数のアドレスを一定に保ちます。

*embedded-statement*:

```
...
fixed-statement
```

*fixed-statement*:

```
fixed ( pointer-type fixed-pointer-declarators ) embedded-statement
```

*fixed-pointer-declarators*:

```
fixed-pointer-declarator
fixed-pointer-declarators , fixed-pointer-declarator
```

*fixed-pointer-declarator*:

```
identifier = fixed-pointer-initializer
```

*fixed-pointer-initializer*:

```
& variable-reference
expression
```

各 *fixed-pointer-declarator* は、指定された *pointer-type* のローカル変数を宣言し、対応する *fixed-pointer-initializer* によって計算されたアドレスでそのローカル変数を初期化します。 `fixed` ステート

メントで宣言されたローカル変数は、その変数宣言の右側にある *fixed-pointer-initializer* 内、および **fixed** ステートメントの *embedded-statement* 内でアクセスできます。**fixed** ステートメントで宣言されたローカル変数は、読み取り専用と見なされます。埋め込みステートメントが、代入または **++** 演算子と **--** 演算子でローカル変数を変更しようとしたり、ローカル変数を **ref** パラメーターまたは **out** パラメーターとして渡そうとしたりすると、コンパイルエラーが発生します。

*fixed-pointer-initializer* には次のいずれかを指定できます。

- $T^*$  型を **fixed** ステートメントで指定されたポインター型に暗黙的に変換できる場合は、トークン "&" の後に、アンマネージ型  $T$  の移動可能変数 (5.3.3 を参照) への *variable-reference* (18.3 を参照) を指定できます。この場合、初期化子は指定された変数のアドレスを計算し、変数は **fixed** ステートメントが存続する間は固定されたアドレスにとどまります。
- $T^*$  型を **fixed** ステートメントで指定されたポインター型に暗黙的に変換できる場合は、アンマネージ型  $T$  の要素を持つ *array-type* の式を指定できます。この場合、初期化子は配列内の先頭要素のアドレスを計算し、配列全体は **fixed** ステートメントが存続する間は固定されたアドレスにとどまります。配列式が **null** の場合、または配列の要素が 0 の場合、**fixed** ステートメントの動作は実装で定義されます。
- **char\*** 型を **fixed** ステートメントで指定されたポインター型に暗黙的に変換できる場合は、**string** 型の式を指定できます。この場合、初期化子は文字列内の先頭文字のアドレスを計算し、文字列全体は **fixed** ステートメントが存続する間は固定されたアドレスにとどまります。文字列式が **null** の場合、**fixed** ステートメントの動作は実装で定義されます。
- 固定サイズバッファー メンバーの型を **fixed** ステートメントで指定されたポインター型に暗黙的に変換できる場合は、移動可能変数の固定サイズバッファー メンバーを参照する *simple-name* または *member-access* を指定できます。この場合、初期化子は固定サイズバッファー (18.7.2 を参照) の最初の要素へのポインターを計算し、固定サイズバッファーは **fixed** ステートメントが存続する間は固定されたアドレスにとどまります。

*fixed-pointer-initializer* で計算された各アドレスについては、**fixed** ステートメントによって、そのアドレスが参照する変数が、**fixed** ステートメントが存続する間はガベージコレクターで再配置または破棄されないことが保証されます。たとえば、*fixed-pointer-initializer* で計算されたアドレスがオブジェクトのフィールドまたは配列インスタンスの要素を参照する場合、**fixed** ステートメントによって、ステートメントが存続する間はオブジェクトインスタンスが再配置または破棄されないことが保証されます。

プログラマは、**fixed** ステートメントにより作成されたポインターがそのステートメントの実行後も残存しないようにする必要があります。たとえば、**fixed** ステートメントで作成されたポインターが外部 API に渡される場合は、その API がこれらのポインターのメモリを保持しないようにする必要があります。

固定されたオブジェクトによってヒープの断片化が発生することがあります。これは、オブジェクトを移動できないためです。このため、絶対に必要な場合にだけ、できるだけ短い時間に限って、オブジェクトを固定する必要があります。

次の例を参照してください。

```
class Test
{
    static int x;
    int y;
```

```

unsafe static void F(int* p) {
    *p = 1;
}

static void Main() {
    Test t = new Test();
    int[] a = new int[10];
    unsafe {
        fixed (int* p = &x) F(p);
        fixed (int* p = &t.y) F(p);
        fixed (int* p = &a[0]) F(p);
        fixed (int* p = a) F(p);
    }
}
}

```

この例は、**fixed** ステートメントのいくつかの使用方法を示しています。最初のステートメントは、静的フィールドのアドレスを固定して取得します。2番目のステートメントは、インスタンスフィールドのアドレスを固定して取得します。3番目のステートメントは、配列要素のアドレスを固定して取得します。いずれの場合でも、通常の **&** 演算子を使用するとエラーになります。これは、変数がすべて移動可能変数として分類されるためです。

上の例の4番目の **fixed** ステートメントは、3番目と同様の結果になります。

この **fixed** ステートメントの例では **string** を使用します。

```

class Test
{
    static string name = "xx";
    unsafe static void F(char* p) {
        for (int i = 0; p[i] != '\0'; ++i)
            Console.WriteLine(p[i]);
    }

    static void Main() {
        unsafe {
            fixed (char* p = name) F(p);
            fixed (char* p = "xx") F(p);
        }
    }
}

```

**unsafe** コンテキストでは、1次元配列の配列要素は、インデックス 0 から始まってインデックス **Length - 1** で終わるインデックスの昇順に格納されます。多次元配列の場合の配列要素は、右端の次元のインデックスが最初に増加し、次にその左側の次元のインデックスが増加し、さらにその左側が増加する、という順序で格納されます。配列インスタンス **a** へのポインター **p** を取得する **fixed** ステートメント内で、**p** から **p + a.Length - 1** までの範囲のポインター値は、配列内の要素のアドレスを表します。同様に、**p[0]** から **p[a.Length - 1]** までの範囲の変数は、実際の配列要素を表します。配列の格納方法に従って、任意の次元の配列を線形として処理できます。

次に例を示します。

```
using System;
```

```

class Test
{
    static void Main() {
        int[,] a = new int[2,3,4];
        unsafe {
            fixed (int* p = a) {
                for (int i = 0; i < a.Length; ++i) // treat as linear
                    p[i] = i;
            }
        }

        for (int i = 0; i < 2; ++i)
            for (int j = 0; j < 3; ++j) {
                for (int k = 0; k < 4; ++k)
                    Console.WriteLine("[{0},{1},{2}] = {3} {i}, {j}, {k}, {a[i,j,k]}");
                Console.WriteLine();
            }
        }
    }
}

```

この例では、次のように出力されます。

[0,0,0]	=	0	[0,0,1]	=	1	[0,0,2]	=	2	[0,0,3]	=	3
[0,1,0]	=	4	[0,1,1]	=	5	[0,1,2]	=	6	[0,1,3]	=	7
[0,2,0]	=	8	[0,2,1]	=	9	[0,2,2]	=	10	[0,2,3]	=	11
[1,0,0]	=	12	[1,0,1]	=	13	[1,0,2]	=	14	[1,0,3]	=	15
[1,1,0]	=	16	[1,1,1]	=	17	[1,1,2]	=	18	[1,1,3]	=	19
[1,2,0]	=	20	[1,2,1]	=	21	[1,2,2]	=	22	[1,2,3]	=	23

次に例を示します。

```

class Test
{
    unsafe static void Fill(int* p, int count, int value) {
        for (; count != 0; count--) *p++ = value;
    }

    static void Main() {
        int[] a = new int[100];
        unsafe {
            fixed (int* p = a) Fill(p, 100, -1);
        }
    }
}

```

この例では、**fixed** ステートメントを使用して、配列を固定します。これによって、ポインターを持つメソッドに配列のアドレスを渡すことができます。

次に例を示します。

```

unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufsize) {
        int len = s.Length;
        if (len > bufsize) len = bufsize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufsize; i++) buffer[i] = (char)0;
    }

    Font f;
}

```

```

unsafe static void Main()
{
    Test test = new Test();
    test.f.size = 10;
    fixed (char* p = test.f.name) {
        PutString("Times New Roman", p, 32);
    }
}

```

この例では、`fixed` ステートメントを使用して、構造体の固定サイズバッファーを固定します。これによって、そのアドレスをポインターとして使用できます。

文字列インスタンスを固定することによって生成された `char*` 値は、`null` で終わる文字列を指します。文字列インスタンス `s` へのポインター `p` を取得する `fixed` ステートメント内で、`p` から `p + s.Length - 1` までの範囲にあるポインター値は、文字列内の文字のアドレスを表します。また、ポインター値 `p + s.Length` は、常に `null` 文字(値が '`\0`' の文字)を指します。

固定されたポインターを通じてマネージ型のオブジェクトを変更すると、未定義の動作が発生する可能性があります。文字列は変更できないため、プログラマはたとえば、固定された文字列へのポインターによって参照される文字が変更されないようにする必要があります。

文字列を自動的に `null` で終わらせるか、"C スタイル" の文字列を期待する外部 API を呼び出す場合に便利です。ただし、文字列インスタンスは `null` 文字を含むことができます。`null` 文字が存在する場合、文字列が `null` で終わる `char*` として処理されると、切り捨てられたように見えます。

## 18.7 固定サイズバッファー

固定サイズバッファーを使用して、"C スタイル" のインライン配列を構造体のメンバーとして宣言できます。固定サイズバッファーは主に、アンマネージ型の API とのインターフェイスとして使用するのに役立ちます。

### 18.7.1 固定サイズバッファーの宣言

固定サイズバッファーは、特定の型の変数に対応する固定長バッファーのストレージを示すメンバーです。固定サイズバッファーを宣言すると、特定の要素型の 1 つ以上の固定サイズバッファーが導入されます。固定サイズバッファーは構造体宣言のみで許可され、`unsafe` コンテキスト(18.1 を参照)でのみ使用できます。

```

struct-member-declaration:
...
fixed-size-buffer-declaration

fixed-size-buffer-declaration:
    attributesopt fixed-size-buffer-modifiersopt fixed buffer-element-type
    fixed-size-buffer-declarators ;

fixed-size-buffer-modifiers:
    fixed-size-buffer-modifier
    fixed-size-buffer-modifier fixed-size-buffer-modifiers

```

```

fixed-size-buffer-modifier:
  new
  public
  protected
  internal
  private
  unsafe

buffer-element-type:
  type

fixed-size-buffer-declarators:
  fixed-size-buffer-declarator
  fixed-size-buffer-declarator , fixed-size-buffer-declarators

fixed-size-buffer-declarator:
  identifier [ constant-expression ]

```

固定サイズ バッファーの宣言では、属性(17を参照)の集合、`new`修飾子(10.2.2を参照)、4つのアクセス修飾子の有効な組み合わせ(10.2.3を参照)、および`unsafe`修飾子(18.1を参照)を指定できます。属性および修飾子は、固定サイズ バッファーの宣言で宣言されるすべてのメンバーに適用されます。1つの固定サイズ バッファーの宣言内で同じ修飾子を複数回使用すると、エラーになります。

固定サイズ バッファーの宣言に、`static`修飾子を指定することはできません。

固定サイズ バッファーの宣言のバッファーの要素型は、宣言によって導入されるバッファーの要素型を指定します。バッファーの要素型は、定義済みの型 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`char`、`float`、`double`、または`bool`である必要があります。

このバッファーの要素の型の後には、固定サイズ バッファーの宣言のリストが続き、各宣言子によって新規のメンバーが導入されます。固定サイズ バッファー宣言子は、メンバーに名前を付ける識別子と、それに続く[トークンおよび]トークンで囲まれた定数式で構成されます。定数式は、その固定サイズ バッファーの宣言子によって導入されるメンバーの要素の数を示します。定数式の型は暗黙的に型 `int` に変換できる必要があり、値はゼロ以外の正の整数である必要があります。

固定サイズ バッファーの要素は、メモリ内に順番に配置されることが保証されます。

複数の固定サイズ バッファーを宣言する固定サイズ バッファー宣言は、属性と要素型が同じ1つの固定サイズ バッファー宣言を複数回宣言することと等価です。次に例を示します。

```

unsafe struct A
{
    public fixed int x[5], y[10], z[100];
}

```

上記のコードは、次のコードと同じです。

```

unsafe struct A
{
    public fixed int x[5];
    public fixed int y[10];
    public fixed int z[100];
}

```

## 18.7.2 式の固定サイズ バッファー

固定サイズ バッファーメンバーのメンバー検索(7.3を参照)は、フィールドのメンバー検索と完全に同じように処理されます。

式で固定サイズバッファーを参照するには、*simple-name* (7.5.2 を参照) または *member-access* (7.5.4 を参照) を使用します。

固定サイズバッファー メンバーを簡易名として参照することは、固定サイズバッファー メンバーを *this.I* としたときの *I* 形式のメンバー アクセスと同じです。

形式 *E.I* のメンバー アクセスで、*E* が構造体型であり、その構造体型で *I* のメンバー検索を実行することで固定サイズのメンバーが識別される場合、*E.I* は次のように分類および評価されます。

- 式 *E.I* が unsafe コンテキストで使用されない場合は、コンパイル エラーが発生します。
- E* が 値に分類される場合は、コンパイル エラーが発生します。
- それ以外の場合で、*E* が移動可能変数 (18.3 を参照) であり、式 *E.I* が *fixed-pointer-initializer* (18.6 を参照) でない場合は、コンパイル エラーが発生します。
- 上記のいずれでもない場合、*E* は固定変数を参照し、式の結果は *E* の固定サイズバッファー メンバー *I* の最初の要素のポインターになります。*S* を *I* の要素型とする型 *S\** の結果は、値に分類されます。

固定サイズバッファーのその後の要素には、最初の要素からのポインター操作を使用してアクセスできます。配列へのアクセスとは異なり、固定サイズバッファーの要素へのアクセスは安全でない操作であり、範囲のチェックは行われません。

次の例は、固定サイズバッファー メンバーを持つ構造体を宣言して使用します。

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufsize) {
        int len = s.Length;
        if (len > bufsize) len = bufsize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufsize; i++) buffer[i] = (char)0;
    }

    unsafe static void Main()
    {
        Font f;
        f.size = 10;
        PutString("Times New Roman", f.name, 32);
    }
}
```

### 18.7.3 確実な代入のチェック

固定サイズバッファーには確実な代入のチェック (5.3 を参照) の制限事項はなく、構造体型変数の確実な代入のチェックにあたって固定サイズバッファーのメンバーは無視されます。

固定サイズバッファー メンバーの最も外側の構造体変数が静的変数、クラス インスタンスのインスタンス変数、または配列要素の場合は、固定サイズバッファーの要素は自動的に既定値に初期化されます (5.2 を参照)。これ以外のすべての場合、固定サイズバッファーの初期内容は未定義です。

## 18.8 スタック割り当て

`unsafe` コンテキストでは、ローカル変数宣言 (8.5.1 を参照) に、コールスタックからメモリを割り当てるスタック割り当て初期化子を含めることができます。

```

local-variable-initializer:
...
stackalloc-initializer
stackalloc-initializer:
    stackalloc unmanaged-type [ expression ]

```

`unmanaged-type` は新しく割り当てられた場所に格納される項目の型を示し、`expression` はこれらの項目の数を示します。この 2 つを使用して、必要な割り当てサイズを指定します。スタック割り当てのサイズを負の値にすることはできないため、負の値に評価される `constant-expression` として項目数を指定すると、コンパイルエラーになります。

`stackalloc T[E]` 形式のスタック割り当て初期化子では、`T` をアンマネージ型 (18.2 を参照) に、`E` を `int` 型の式にする必要があります。この構造は、コールスタックから `E * sizeof(T)` バイトを割り当てる、新しく割り当てられたブロックを指す `T*` 型のポインターを返します。`E` が負の値である場合の動作は不定です。`E` が 0 の場合、割り当ては行われず、返されるポインターは実装で定義されます。メモリが足りないために指定されたサイズのブロックを割り当てられない場合は、`System.StackOverflowException` がスローされます。

新しく割り当てられたメモリの内容は不定です。

スタック割り当て初期化子は、`catch` ブロックや `finally` ブロックでは使用できません (8.10 を参照)。

`stackalloc` を使用して割り当てられたメモリを明示的に解放する方法はありません。関数メンバーの実行中に作成された、スタック割り当てのすべてのメモリブロックは、関数メンバーから制御が戻るときに自動的に破棄されます。これは、C および C++ の実装で一般的な拡張機能である `alloca` 関数に対応しています。

次に例を示します。

```

using System;
class Test
{
    static string IntToString(int value) {
        int n = value >= 0? value: -value;
        unsafe {
            char* buffer = stackalloc char[16];
            char* p = buffer + 16;
            do {
                *--p = (char)(n % 10 + '0');
                n /= 10;
            } while (n != 0);
            if (value < 0) *--p = '-';
            return new string(p, 0, (int)(buffer + 16 - p));
        }
    }
    static void Main() {
        Console.WriteLine(IntToString(12345));
        Console.WriteLine(IntToString(-999));
    }
}

```

この例では、`stackalloc` 初期化子が `IntToString` メソッドで使用されて、スタックに 16 文字のバッファーを割り当てます。メソッドから制御が戻ると、バッファーは自動的に破棄されます。

## 18.9 動的メモリ割り当て

`stackalloc` 演算子を除いて、C# には、ガベージコレクションの対象とならないメモリを管理するための、定義済みの構成要素がありません。このようなメモリを管理するサービスは、通常、クラスライブラリをサポートすることで提供されるか、または基になるオペレーティングシステムから直接インポートされます。たとえば、次の例の `Memory` クラスは、基になるオペレーティングシステムのヒープ関数に C# からアクセスする方法を示しています。

```
using System;
using System.Runtime.InteropServices;

public unsafe class Memory
{
    // Handle for the process heap. This handle is used in all calls to the
    // HeapXXX APIs in the methods below.
    static int ph = GetProcessHeap();

    // Private instance constructor to prevent instantiation.
    private Memory() {}

    // Allocates a memory block of the given size. The allocated memory is
    // automatically initialized to zero.
    public static void* Alloc(int size) {
        void* result = HeapAlloc(ph, HEAP_ZERO_MEMORY, size);
        if (result == null) throw new OutOfMemoryException();
        return result;
    }

    // Copies count bytes from src to dst. The source and destination
    // blocks are permitted to overlap.
    public static void Copy(void* src, void* dst, int count) {
        byte* ps = (byte*)src;
        byte* pd = (byte*)dst;
        if (ps > pd) {
            for (; count != 0; count--) *pd++ = *ps++;
        }
        else if (ps < pd) {
            for (ps += count, pd += count; count != 0; count--) *--pd = *--ps;
        }
    }

    // Frees a memory block.
    public static void Free(void* block) {
        if (!HeapFree(ph, 0, block)) throw new InvalidOperationException();
    }

    // Re-allocates a memory block. If the reallocation request is for a
    // larger size, the additional region of memory is automatically
    // initialized to zero.
    public static void* ReAlloc(void* block, int size) {
        void* result = HeapReAlloc(ph, HEAP_ZERO_MEMORY, block, size);
        if (result == null) throw new OutOfMemoryException();
        return result;
    }

    // Returns the size of a memory block.
}
```

```

public static int Sizeof(void* block) {
    int result = HeapSize(ph, 0, block);
    if (result == -1) throw new InvalidOperationException();
    return result;
}
// Heap API flags
const int HEAP_ZERO_MEMORY = 0x00000008;
// Heap API functions
[DllImport("kernel32")]
static extern int GetProcessHeap();
[DllImport("kernel32")]
static extern void* HeapAlloc(int hHeap, int flags, int size);
[DllImport("kernel32")]
static extern bool HeapFree(int hHeap, int flags, void* block);
[DllImport("kernel32")]
static extern void* HeapReAlloc(int hHeap, int flags,
    void* block, int size);
[DllImport("kernel32")]
static extern int HeapSize(int hHeap, int flags, void* block);
}

```

この `Memory` クラスを使用する例を次に示します。

```

class Test
{
    static void Main() {
        unsafe {
            byte* buffer = (byte*)Memory.Alloc(256);
            try {
                for (int i = 0; i < 256; i++) buffer[i] = (byte)i;
                byte[] array = new byte[256];
                fixed (byte* p = array) Memory.Copy(buffer, p, 256);
            }
            finally {
                Memory.Free(buffer);
            }
            for (int i = 0; i < 256; i++) Console.WriteLine(array[i]);
        }
    }
}

```

この例では、`Memory.Alloc` を使用して 256 バイトのメモリを割り当て、0 から 255 までの値でメモリ ブロックを初期化します。次に、256 の要素を持つバイト配列を割り当て、`Memory.Copy` を使用してメモリ ブロックの内容をバイト配列にコピーします。最後に、`Memory.Free` を使用してメモリ ブロックを解放し、バイト配列の内容をコンソールに出力します。

# A. ドキュメントのコメント

C# には、XML テキストを含む特殊なコメント構文を使用してコードをドキュメント化するためのプログラマ向けの機構が用意されています。ソースコードファイルで特定の形式のコメントを使用すると、コメントとその直後にあるソースコード要素から XML を生成するようにツールに指示できます。この特殊な構文を使用したコメントは、**ドキュメントコメント**と呼ばれます。ドキュメントコメントは、ユーザー定義型(クラス、デリゲート、インターフェイスなど)またはメンバー(フィールド、イベント、プロパティ、メソッドなど)の直前に指定する必要があります。XML 生成ツールは、**ドキュメントジェネレーター**\t "See documentation comment" と呼ばれます(C# コンパイラ自体がこのジェネレーターになる場合もあります)。ドキュメントジェネレーターによって生成される出力は、**ドキュメントファイル**\t "See documentation comment" \b と呼ばれます。ドキュメントファイルは、型情報とそれに関連したドキュメントを視覚的に表示することを目的としたツールである**ドキュメントビューアー**\t "See documentation comment" \b \b の入力ファイルとして使用します。

この仕様では、ドキュメントコメントで使用する一連のタグについて説明しますが、タグの使用は必須ではありません。整形式の XML 規則に従う限り、必要に応じて他のタグも使用できます。

## A.1 概要

特殊な形式のコメントを使用すると、コメントとその直後にあるソースコード要素から XML を生成するようにツールに指示できます。コメントには、3つのスラッシュ(/ / /)で始まる單一行のコメントと、スラッシュと2つのアスタリスク(/ \*\*/)で始まる区切られたコメントがあります。ドキュメントコメントは、注釈を付けるユーザー定義型(クラス、デリゲート、インターフェイスなど)またはメンバー(フィールド、イベント、プロパティ、メソッドなど)の直前に指定する必要があります。属性セクション(17.2を参照)は宣言の一部と見なされるため、ドキュメントコメントは型またはメンバーに適用される属性よりも前に指定する必要があります。

構文：

```

single-line-doc-comment:
  /// input-charactersopt

delimited-doc-comment:
  /** delimited-comment-textopt */

```

*single-line-doc-comment* では、現在の *single-line-doc-comment* に隣接する各 *single-line-doc-comment* の /// 文字の直後に空白文字がある場合、その空白文字は XML 出力に含まれません。

*delimited-doc-comment* では、2行目の最初の空白以外の文字がアスタリスクで、*delimited-doc-comment* 内の各行の先頭にあるオプションの空白文字とアスタリスク文字が同じパターンで繰り返される場合、繰り返しのパターンの文字は XML 出力に含まれません。パターンのアスタリスク文字の前後には、空白文字を挿入できます。

例：

```

    ///<summary>Class <c>Point</c> models a point in a two-dimensional
    ///<summary>plane.</summary>
    ///
    public class Point
    {
        ///<summary>method <c>draw</c> renders the point.</summary>
        void draw() {…}
    }

```

ドキュメントコメント内のテキストは、整形式の XML の規則に従っている必要があります (<http://www.w3.org/TR/REC-xml>)。適切な XML でない場合は警告が表示され、ドキュメントファイルにはエラーが発生したことを表すコメントが記録されます。

開発者は独自のタグを自由に作成できますが、推奨される一連のタグは付録 A.2 で定義されています。推奨されるタグには、次のような特殊な意味を持つタグがあります。

- `<param>` タグは、パラメーターの説明に使用します。このタグが使用されている場合、ドキュメントジェネレーターは、指定されたパラメーターが存在することと、すべてのパラメーターがドキュメントコメント内に記述されていることを検査する必要があります。検査が失敗すると、ドキュメントジェネレーターは警告を通知します。
- `cref \b` 属性をタグに追加すると、コード要素を参照できます。ドキュメントジェネレーターは、該当するコード要素が存在することを検査する必要があります。検査が失敗すると、ドキュメントジェネレーターは警告を通知します。`cref` 属性で記述された名前を検索する場合、ドキュメントジェネレーターは、ソースコード内に出現する `using` ステートメントに応じて名前空間の可視性を考慮する必要があります。ジェネリックのコード要素では、通常のジェネリック構文 ("`List<T>`") は無効な XML が生成されるので使用できません。山かつこの代わりに、中かつこ ("`List{T}`") または XML のエスケープ構文 ("`List&lt;T&gt;`") を使用できます。
- `<summary>` タグは、型やメンバーの追加情報を表示するためにドキュメントビューアーで使用されます。
- `<include>` タグには、外部 XML ファイルの情報が含まれます。

ドキュメントファイルでは、型やメンバーに関する完全な情報は提供されません。たとえば、ドキュメントファイルには型情報が含まれていません。型やメンバーの完全な情報を得るには、ドキュメントファイルを実際の型やメンバーのリフレクションと共に使用する必要があります。

## A.2 推奨されるタグ

ドキュメントジェネレーターは、XML の規則に従って、有効なタグをすべて受け入れて処理する必要があります。ユーザー ドキュメントで一般的に使用される機能を提供するタグを次の表に示します。この表に示した以外のタグも使用できます。

タグ	セクション	目的
<c>	A.2.1	テキストをコード形式のフォントに設定します。
<code>	A.2.2	1行以上のソース コードまたはプログラム出力を設定します。
<example>	A.2.3	例を示します。
<exception>	A.2.4	メソッドがスローする例外を示します。
<include>	A.2.5	外部ファイルから XML をインクルードします。
<list>	A.2.6	リストまたは表を作成します。
<para>	A.2.7	テキストを構造化できます。
<param>	A.2.8	メソッドまたはコンストラクターのパラメーターについて説明します。
<paramref>	A.2.9	単語がパラメータ一名であることを示します。
<permission>	A.2.10	メンバーのセキュリティ アクセシビリティを示します。
<remark>	A.2.11	型に関する追加情報について説明します。
<returns>	A.2.12	メソッドの戻り値について説明します。
<see>	A.2.13	リンクを指定します。
<seealso>	A.2.14	「参照」のエントリを生成します。
<summary>	A.2.15	型または型のメンバーについて説明します。
<value>	A.2.16	プロパティについて説明します。
<typeparam>		ジェネリック型パラメーターについて説明します。
<typeparamref>		単語が型パラメータ一名であることを示します。

### A.2.1 <c> \t "See <c>" \b

このタグは、記述内のテキストの一部を、コードブロックで使用される特殊なフォントに設定する必要があることを示します。実際のコード行には、<code> (A.2.2 を参照) を使用します。

構文：

```
<c> text</c>
```

例：

```
///<summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>
public class Point
{
    // ...
}
```

**A.2.2 <code> \t "See <code>" \b**

このタグは、1行以上のソース コードまたはプログラム出力を特殊フォントに設定する場合に使用します。説明文の中に出現するコード片には、`<c>` (A.2.1 を参照) を使用します。

構文 :

```
<code>source code or program output</code>
```

例 :

```
///<summary>This method changes the point's location by
///the given x- and y-offsets.
///<example>For example:
///<code>
///    Point p = new Point(3,5);
///    p.Translate(-1,3);
///</code>
/// results in <c>p</c>'s having the value (2,8).
///</example>
///</summary>

public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}
```

**A.2.3 <example> \t "See <example>" \b**

このタグを使用すると、コメント内にプログラム例を挿入して、メソッドや他のライブラリ メンバーの使用方法を指定できます。通常は、タグ `<code>` (A.2.2 を参照) と一緒に使用します。

構文 :

```
<example>description</example>
```

例 :

例については、「`<code>`」 (A.2.2 を参照) を参照してください。

**A.2.4 <exception> \t "See <exception>" \b**

このタグを使用すると、メソッドがスローできる例外を示すことができます。

構文 :

```
<exception cref="member">description</exception>
```

指定項目

`cref="member"`

メンバーの名前。ドキュメント ジェネレーターは、指定されたメンバーが存在するかどうかをチェックし、`member` をドキュメント ファイル内の標準要素名に変換します。

*description*

例外がスローされる状況についての説明です。

例 :

```

public class DataBaseOperations
{
    ///<exception cref="MasterFileFormatCorruptException"></exception>
    ///<exception cref="MasterFileLockedOpenException"></exception>
    public static void ReadRecord(int flag) {
        if (flag == 1)
            throw new MasterFileFormatCorruptException();
        else if (flag == 2)
            throw new MasterFileLockedOpenException();
        // ...
    }
}

```

### A.2.5 <include>

このタグを使用すると、ソースコードファイルの外部にある XML ドキュメントの情報をインクルードできます。外部ファイルは整形式の XML ドキュメントであることが必要です。XPath 式を XML ドキュメントに適用して、ドキュメントのどの XML をインクルードするかを指定します。次に、<include> タグは、外部ドキュメントで選択されている XML で置き換えられます。

構文：

```
<include file="filename" path="xpath" />
```

指定項目

**file="filename"**

外部 XML ファイルの名前です。ファイル名は、include タグが含まれているファイルに関連のあるファイルとして解釈されます。

**path="xpath"**

外部 XML ファイルにある一部の XML を選択する XPath 式です。

例：

ソースコードで、次のように宣言します。

```
///<include file="docs.xml" path='extradoc/class[@name="IntList"]/*' />
public class IntList { ... }
```

外部ファイル "docs.xml" の内容は、次のとおりです。

```

<?xml version="1.0"?>
<extradoc>
    <class name="IntList">
        <summary>
            Contains a list of integers.
        </summary>
    </class>
    <class name="StringList">
        <summary>
            Contains a list of integers.
        </summary>
    </class>
</extradoc>

```

出力されるドキュメントは、ソースコードに次の内容が記述されていた場合と同じです。

```

///<summary>
///    Contains a list of integers.
///</summary>
public class IntList { ... }

```

**A.2.6 <list> \t "See <list>" \b**

このタグを使用して、項目のリストや表を作成します。<listheader> ブロックを含めると、表または定義リストの見出し行を定義できます(表を定義する場合は、見出しの *term* のエントリだけを指定します)。

リストの各項目は、<item> ブロックで指定します。定義リストを作成する場合は、*term* と *description* の両方を指定する必要があります。ただし、表、箇条書きリスト、または番号付きリストの場合は、*description*だけを指定します。

構文：

```
<list type="bullet" | "number" | "table">
  <listheader>
    <term>term</term>
    <description>description</description>
  </listheader>
  <item>
    <term>term</term>
    <description>description</description>
  </item>
  ...
  <item>
    <term>term</term>
    <description>description</description>
  </item>
</list>
```

指定項目

*term*

定義する用語です。定義は、*description* に記述します。

*description*

箇条書きリストまたは番号付きリストの項目、あるいは *term* の定義のいずれか。

例：

```
public class MyClass
{
  /// <summary>Here is an example of a bulleted list:
  /// <list type="bullet">
  ///   <item>
  ///     <description>Item 1.</description>
  ///   </item>
  ///   <item>
  ///     <description>Item 2.</description>
  ///   </item>
  /// </list>
  /// </summary>
  public static void Main () {
    // ...
  }
}
```

**A.2.7 <para> \t "See <para>" \b**

このタグを <summary> (A.2.11 を参照) や <returns> (A.2.12 を参照) などの他のタグの内部で使用すると、テキストを構造化できます。

構文：

<para>*content*</para>

指定項目

*content*

段落のテキスト。

例：

```
/// <summary>This is the entry point of the Point class testing program.
/// <para>This program tests each method and operator, and
/// is intended to be run after any non-trivial maintenance has
/// been performed on the Point class.</para></summary>
public static void Main() {
    // ...
}
```

#### A.2.8 <param> \t "See <param>" \b

このタグを使用して、メソッド、コンストラクター、またはインデクサーのパラメーターについて説明します。

構文：

<param name="*name*">*description*</param>

指定項目

*name*

パラメーターの名前です。

*description*

パラメーターの説明です。

例：

```
/// <summary>This method changes the point's location to
/// the given coordinates.</summary>
/// <param name="xor">the new x-coordinate.</param>
/// <param name="yor">the new y-coordinate.</param>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}
```

#### A.2.9 <paramref> \t "See <paramref>" \b

このタグは、単語がパラメーターであることを示すために使用します。ドキュメントファイルを処理するときに、このパラメーターに別の書式を設定できます。

構文：

<paramref name="*name*" />

指定項目

*name*

パラメーターの名前です。

例：

```

///<summary>This constructor initializes the new Point to
///(<paramref name="xor"/>,<paramref name="yor"/>).</summary>
///<param name="xor">the new Point's x-coordinate.</param>
///<param name="yor">the new Point's y-coordinate.</param>
public Point(int xor, int yor) {
    X = xor;
    Y = yor;
}

```

**A.2.10 <permission> It "See <permission>" \b**

このタグを使用すると、メンバーのセキュリティ アクセシビリティを示すことができます。

構文：

```
<permission cref="member">description</permission>
```

指定項目

*cref="member"*

メンバーの名前。ドキュメント ジェネレーターは、指定されたコード要素が存在するかどうかを確認し、*member* をドキュメント ファイル内の標準要素名に変換します。

*description*

メンバーへのアクセスの説明。

例：

```

///<permission cref="System.Security.PermissionSet">Everyone can
///access this method.</permission>
public static void Test() {
    // ...
}

```

**A.2.11 <remark> It "See <remarks>" \b**

このタグを使用して、型の追加情報を指定します。型自体および型のメンバーの説明には、  
<summary> (A.2.15 を参照) を使用します。

構文：

```
<remark>description</remark>
```

指定項目

*description*

解説のテキストです。

例：

```

///<summary>Class <c>Point</c> models a point in a
///two-dimensional plane.</summary>
///<remark>Uses polar coordinates</remark>
public class Point
{
    // ...
}

```

**A.2.12 <returns> It "See <returns>" \b**

このタグを使用して、メソッドの戻り値について説明します。

構文：

```
<returns>description</returns>
```

指定項目

*description*

戻り値の説明。

例：

```
/// <summary>Report a point's location as a string.</summary>
/// <returns>A string representing a point's location, in the form (x,y),
/// without any leading, trailing, or embedded whitespace.</returns>
public override string ToString() {
    return "(" + X + "," + Y + ")";
}
```

### A.2.13 <see> \t "See <see>" \b

このタグを使用すると、テキスト内にリンクを指定できます。「参照」セクションに示すテキストを指定するには、<seealso> (A.2.14 を参照) を使用します。

構文：

```
<see cref="member"/>
```

指定項目

*cref="member"*

メンバーの名前。ドキュメントジェネレーターは、指定されたコード要素が存在するかどうかを確認し、*member* を生成されたドキュメントファイル内の要素名に変更します。

例：

```
/// <summary>This method changes the point's location to
/// the given coordinates.</summary>
/// <see cref="Translate"/>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}
/// <summary>This method changes the point's location by
/// the given x- and y-offsets.
/// </summary>
/// <see cref="Move"/>
public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}
```

### A.2.14 <seealso> \t "See <seealso>" \b

このタグを使用すると、「参照」セクションのエントリを生成できます。(テキスト内でリンクを指定するには、<see> (A.2.13 を参照) タグを使用します。)

構文：

```
<seealso cref="member"/>
```

指定項目

*cref="member"*

メンバーの名前。ドキュメントジェネレーターは、指定されたコード要素が存在するかどうかを確認し、*member* を生成されたドキュメントファイル内の要素名に変更します。

例：

```
/// <summary>This method determines whether two Points have the same
/// location.</summary>
/// <seealso cref="operator=="/>
/// <seealso cref="operator!="/>
public override bool Equals(object o) {
    // ...
}
```

#### A.2.15 <summary> \t "See <summary>" \b

このタグを使用すると、型または型のメンバーについて説明できます。型自体の説明には、<remark> (A.2.11 を参照) を使用します。

構文：

<summary>*description*</summary>

指定項目

*description*

型またはメンバーの概要です。

例：

```
/// <summary>This constructor initializes the new Point to (0,0).</summary>
public Point() : this(0,0) { }
```

#### A.2.16 <value> \t "See <value>" \b

このタグを使用すると、プロパティについて説明できます。

構文：

<value>*property description*</value>

指定項目

*property description*

プロパティの説明。

例：

```
/// <value>Property <c>X</c> represents the point's x-coordinate.</value>
public int X
{
    get { return x; }
    set { x = value; }
}
```

#### A.2.17 <typeparam>

このタグを使用して、クラス、構造体、インターフェイス、デリゲート、またはメソッドのジェネリック型パラメーターについて説明します。

構文：

<typeparam name="*name*">*description*</typeparam>

## 指定項目

*name*

型パラメーターの名前。

*description*

型パラメーターの説明。

## 例：

```
/// <summary>A generic list class.</summary>
/// <typeparam name="T">The type stored by the list.</typeparam>
public class myList<T> {
    ...
}
```

**A.2.18 <typeparamref>**

このタグは、単語が型パラメーターであることを示すために使用します。ドキュメントファイルを処理するときに、この型パラメーターに別の書式を設定できます。

## 構文：

```
<typeparamref name="name"/>
```

## 指定項目

*name*

型パラメーターの名前。

## 例：

```
/// <summary>This method fetches data and returns a list of <typeparamref
name="T"> "/> .</summary>
/// <param name="string">query to execute</param>

public List<T> FetchData<T>(string query) {
    ...
}
```

**A.3 ドキュメントファイルの処理**

ドキュメントジェネレーターは、ドキュメントコメントでタグ付けされたソースコード内の要素ごとに、ID 文字列を生成します。この ID 文字列は、ソース要素を一意に識別します。ドキュメントビューアーは、ID 文字列を使用して、ドキュメントの適用対象となるメタデータやリフレクション項目を識別できます。

ドキュメントファイルは、ソースコードの階層的表現ではなく、要素ごとに生成された ID 文字列をフラットなリストで表します。

**A.3.1 ID 文字列形式**

ドキュメントジェネレーターは、次の規則に基づいて ID 文字列を生成します。

- 生成される文字列に空白は含まれません。
- 文字列の先頭部分は、記述されるメンバーの種類を、1 文字とコロンを使用して識別します。定義されるメンバーの種類は、次のとおりです。

文字	説明
E	イベント
F	フィールド
M	メソッド(コンストラクター、デストラクター、演算子を含む)
N	名前空間
P	プロパティ(インデクサーを含む)。
T	型(クラス、デリゲート、列挙型、インターフェイス、構造体など)
!	エラー文字列。エラーに続く文字列で、エラーの内容を示します。たとえば、ドキュメントジェネレーターは、解決できないリンクについてのエラー情報を生成します。

- 文字列の2番目の部分は、要素の完全限定名です。名前は、名前空間のルートから始まります。要素の名前、要素が含まれている型、および名前空間は、ピリオドで区切られます。名前自体にピリオドがある場合、名前のピリオドは#(U+0023) 文字に置き換えられます。要素名にはこの文字が含まれていないことが前提になっています。
- メソッドやプロパティの引数がある場合は、かつて囲まれた引数リストが続きます。引数がない場合は、かつてが省略されます。引数の区切り文字には、コンマを使用します。各引数のエンコーディングは、次のように CLI シグネチャと同じです。
  - 引数は、完全修飾名に基づくドキュメント名で表され、次のように変更されます。  
ジェネリック型を表す引数には、"" 文字の後に型パラメーターの数が続きます。
  - out** 修飾子または **ref** 修飾子を持つ引数は、型名の後に @ が付きます。値渡しの引数または **params** を通じて渡す引数には特別な表記がありません。

引数が配列の場合は、[ *lowerbound* : *size* , … , *lowerbound* : *size* ] の形式で表されます。ここで、コンマの個数はランクから 1 を引いた数であり、各次元の下限とサイズは、明らかになっている場合は、10進数で表されます。下限またはサイズが指定されていない場合は、省略されます。特定の次元で下限およびサイズが省略されている場合は、その次元の ":" も省略されます。ジャグ配列は、1 レベルにつき 1 つの "[]" で表されます。

**void** 以外のポインター型を持つ引数は、型名の後に \* が付きます。**void** ポインターは、**System.Void** の型名を使用して表されます。

型で定義されているジェネリック型パラメーターを参照する引数は、"" 文字の後に型パラメーターのゼロから始まるインデックスを続けてエンコードします。

メソッドで定義されているジェネリック型パラメーターを使用する引数は、型で使用する "" の代わりに "``" を使用します。

構築されたジェネリック型を参照する引数は、ジェネリック型、 "{}"、コンマ区切りの型引数リスト、

### A.3.2 ID 文字列の例

次のコード例は、C# のコードの一部と、ドキュメントコメントを持つことができる各ソース要素から生成された ID 文字列を示します。

- 型は、完全修飾名にジェネリック情報を加えて表されます。

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

namespace Acme
{
    interface IProcess { ... }
    struct valueType { ... }
    class widget: IProcess
    {
        public class NestedClass { ... }
        public interface IMenuItem { ... }
        public delegate void Del(int i);
        public enum Direction { North, South, East, West }
    }
    class myList<T>
    {
        class Helper<U,V> { ... }
    }
}

"T:Color"
"T:Acme.IProcess"
"T:Acme.ValueType"
"T:Acme.Widget"
"T:Acme.Widget.NestedClass"
"T:Acme.Widget.IMenuItem"
"T:Acme.Widget.Del"
"T:Acme.Widget.Direction"
"T:Acme.myList`1"
"T:Acme.myList`1.Helper`2"
```

- フィールドは、完全修飾名で表されます。

```
namespace Acme
{
    struct valueType
    {
        private int total;
    }
    class widget: IProcess
    {
        public class NestedClass
        {
            private int value;
        }
    }
}
```

## C# LANGUAGE SPECIFICATION

```
    private string message;
    private static Color defaultColor;
    private const double PI = 3.14159;
    protected readonly double monthlyAverage;
    private long[] array1;
    private Widget[,] array2;
    private unsafe int *pCount;
    private unsafe float **ppValues;
}
}

"F:Acme.ValueType.total"
"F:Acme.Widget.NestedClass.value"
"F:Acme.Widget.message"
"F:Acme.Widget.defaultColor"
"F:Acme.Widget.PI"
"F:Acme.Widget.monthlyAverage"
"F:Acme.Widget.array1"
"F:Acme.Widget.array2"
"F:Acme.Widget.pCount"
"F:Acme.Widget.ppValues"
```

- コンストラクターです。

```
namespace Acme
{
    class Widget: IProcess
    {
        static Widget() {...}
        public Widget() {...}
        public Widget(string s) {...}
    }
}

"M:Acme.Widget.#cctor"
"M:Acme.Widget.#ctor"
"M:Acme.Widget.#ctor(System.String)"
```

- デストラクターです。

```
namespace Acme
{
    class Widget: IProcess
    {
        ~Widget() {...}
    }
}

"M:Acme.Widget.Finalize"
```

- メソッドです。

```
namespace Acme
{
    struct ValueType
    {
        public void M(int i) {...}
    }

    class Widget: IProcess
    {
        public class NestedClass
        {
            public void M(int i) {...}
        }
    }
}
```

```

public static void M0() {...}
public void M1(char c, out float f, ref valueType v) {...}
public void M2(short[] x1, int[,] x2, long[][] x3) {...}
public void M3(long[][] x3, widget[, ,] x4) {...}
public unsafe void M4(char *pc, Color **pf) {...}
public unsafe void M5(void *pv, double *[], pd) {...}
public void M6(int i, params object[] args) {...}
}

class myList<T>
{
    public void Test(T t) { }
}

class UseList
{
    public void Process(myList<int> list) { }
    public myList<T> GetValues<T>(T inputValue) { return null; }
}
}

"M:Acme.ValueType.M(System.Int32)"
"M:Acme.Widget.NestedClass.M(System.Int32)"
"M:Acme.Widget.M0"
"M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"
"M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][][])"
"M:Acme.Widget.M3(System.Int64[],Acme.Widget[0:,0:,0:][])"
"M:Acme.Widget.M4(System.Char*,Color**)"
"M:Acme.Widget.M5(System.Void*,System.Double*[0:,0:][])"
"M:Acme.Widget.M6(System.Int32,System.Object[])"
"M:Acme.MyList`1.Test(`0)"
"M:Acme.UseList.Process(Acme.MyList{System.Int32})"
"M:Acme.UseList.GetValues(`0)"

```

- プロパティとインデクサーです。

```

namespace Acme
{
    class Widget: IProcess
    {
        public int width { get {...} set {...} }
        public int this[int i] { get {...} set {...} }
        public int this[string s, int i] { get {...} set {...} }
    }
}

"P:Acme.Widget.Width"
"P:Acme.Widget.Item(System.Int32)"
"P:Acme.Widget.Item(System.String, System.Int32)"

```

- イベントです。

```

namespace Acme
{
    class Widget: IProcess
    {
        public event Del AnEvent;
    }
}

"E:Acme.Widget.AnEvent"

```

- 単項演算子。

```

namespace Acme
{
    class Widget: IProcess
    {
        public static Widget operator+(Widget x) {...}
    }
}

"M:Acme.Widget.op_UnaryPlus(Acme.Widget)"

```

使用される単項演算子関数の名前は、`op_UnaryPlus`、`op_UnaryNegation`、`op_LogicalNot`、`op_OnesComplement`、`op_Increment`、`op_Decrement`、`op_True`、および`op_False`です。

- 二項演算子。

```

namespace Acme
{
    class Widget: IProcess
    {
        public static Widget operator+(Widget x1, Widget x2) {...}
    }
}

"M:Acme.Widget.op_Addition(Acme.Widget,Acme.Widget)"

```

使用される二項演算子関数の名前は、`op>Addition`、`op_Subtraction`、`op_Multiply`、`op_Division`、`op_Modulus`、`op_BitwiseAnd`、`op_BitwiseOr`、`op_ExclusiveOr`、`op_LeftShift`、`op_RightShift`、`op_Equality`、`op_Inequality`、`op_LessThan`、`op_LessThanOrEqual`、`op_GreaterThan`、および`op_GreaterThanOrEqual`です。

- 変換演算子の末尾には、"~"と戻り値の型が付きます。

```

namespace Acme
{
    class Widget: IProcess
    {
        public static explicit operator int(Widget x) {...}
        public static implicit operator long(Widget x) {...}
    }
}

"M:Acme.Widget.op_Explicit(Acme.Widget)~System.Int32"
"M:Acme.Widget.op_Implicit(Acme.Widget)~System.Int64"

```

## A.4 例

### A.4.1 C# のソース コード

次のコード例は、`Point` クラスのソース コードを示しています。

```

namespace Graphics
{
    /// <summary>Class <c>Point</c> models a point in a two-dimensional plane.
    /// </summary>
    public class Point
    {
        /// <summary>Instance variable <c>x</c> represents the point's
        /// x-coordinate.</summary>
        private int x;

        /// <summary>Instance variable <c>y</c> represents the point's
        /// y-coordinate.</summary>
        private int y;
    }
}

```

```

/// <value>Property <c>X</c> represents the point's x-coordinate.</value>
public int X
{
    get { return x; }
    set { x = value; }
}

/// <value>Property <c>Y</c> represents the point's y-coordinate.</value>
public int Y
{
    get { return y; }
    set { y = value; }
}

/// <summary>This constructor initializes the new Point to
/// (0,0).</summary>
public Point() : this(0,0) {}

/// <summary>This constructor initializes the new Point to
/// (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
/// <param><c>xor</c></param> is the new Point's x-coordinate.</param>
/// <param><c>yor</c></param> is the new Point's y-coordinate.</param>
public Point(int xor, int yor) {
    X = xor;
    Y = yor;
}

/// <summary>This method changes the point's location to
/// the given coordinates.</summary>
/// <param><c>xor</c></param> is the new x-coordinate.</param>
/// <param><c>yor</c></param> is the new y-coordinate.</param>
/// <see cref="Translate"/>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}

/// <summary>This method changes the point's location by
/// the given x- and y-offsets.
/// <example>For example:
/// <code>
///     Point p = new Point(3,5);
///     p.Translate(-1,3);
/// </code>
/// results in <c>p</c>'s having the value (2,8).
/// </example>
/// <summary>
/// <param><c>xor</c></param> is the relative x-offset.</param>
/// <param><c>yor</c></param> is the relative y-offset.</param>
/// <see cref="Move"/>
public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}

/// <summary>This method determines whether two Points have the same
/// location.</summary>
/// <param><c>o</c></param> is the object to be compared to the current object.
/// </param>
/// <returns>True if the Points have the same location and they have
/// the exact same type; otherwise, false.</returns>
/// <seealso cref="operator=="/>
/// <seealso cref="operator!="/>
public override bool Equals(object o) {
    if (o == null) {
        return false;
    }
}

```

```

        if (this == o) {
            return true;
        }
        if (GetType() == o.GetType()) {
            Point p = (Point)o;
            return (X == p.X) && (Y == p.Y);
        }
        return false;
    }

    /// <summary>Report a point's location as a string.</summary>
    /// <returns>A string representing a point's location, in the form (x,y),
    /// without any leading, trailing, or embedded whitespace.</returns>
    public override string ToString() {
        return "(" + X + "," + Y + ")";
    }

    /// <summary>This operator determines whether two Points have the same
    /// location.</summary>
    /// <param><c>p1</c></param> is the first Point to be compared.</param>
    /// <param><c>p2</c></param> is the second Point to be compared.</param>
    /// <returns>True if the Points have the same location and they have
    /// the exact same type; otherwise, false.</returns>
    /// <seealso cref="Equals"/>
    /// <seealso cref="operator!="/>
    public static bool operator==(Point p1, Point p2) {
        if ((object)p1 == null || (object)p2 == null) {
            return false;
        }

        if (p1.GetType() == p2.GetType()) {
            return (p1.X == p2.X) && (p1.Y == p2.Y);
        }

        return false;
    }

    /// <summary>This operator determines whether two Points have the same
    /// location.</summary>
    /// <param><c>p1</c></param> is the first Point to be compared.</param>
    /// <param><c>p2</c></param> is the second Point to be compared.</param>
    /// <returns>True if the Points do not have the same location and the
    /// exact same type; otherwise, false.</returns>
    /// <seealso cref="Equals"/>
    /// <seealso cref="operator=="/>
    public static bool operator!=(Point p1, Point p2) {
        return !(p1 == p2);
    }

    /// <summary>This is the entry point of the Point class testing
    /// program.
    /// <para>This program tests each method and operator, and
    /// is intended to be run after any non-trivial maintenance has
    /// been performed on the Point class.</para></summary>
    public static void Main() {
        // class test code goes here
    }
}
}

```

#### A.4.2 生成される XML

上に示した Point クラスのソース コードに対してドキュメント ジェネレーターが生成する出力は、次のとおりです。

```

<?xml version="1.0"?>
<doc>
  <assembly>
    <name>Point</name>
  </assembly>
  <members>
    <member name="T:Graphics.Point">
      <summary>Class <c>Point</c> models a point in a two-dimensional
      plane.
      </summary>
    </member>

    <member name="F:Graphics.Point.x">
      <summary>Instance variable <c>x</c> represents the point's
      x-coordinate.</summary>
    </member>

    <member name="F:Graphics.Point.y">
      <summary>Instance variable <c>y</c> represents the point's
      y-coordinate.</summary>
    </member>

    <member name="M:Graphics.Point.#ctor">
      <summary>This constructor initializes the new Point to
      (0,0).</summary>
    </member>

    <member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">
      <summary>This constructor initializes the new Point to
      (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
      <param><c>xor</c> is the new Point's x-coordinate.</param>
      <param><c>yor</c> is the new Point's y-coordinate.</param>
    </member>

    <member name="M:Graphics.Point.Move(System.Int32,System.Int32)">
      <summary>This method changes the point's location to
      the given coordinates.</summary>
      <param><c>xor</c> is the new x-coordinate.</param>
      <param><c>yor</c> is the new y-coordinate.</param>
      <see
      cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/>
    </member>

    <member
      name="M:Graphics.Point.Translate(System.Int32,System.Int32)">
      <summary>This method changes the point's location by
      the given x- and y-offsets.
      <example>For example:
      <code>
        Point p = new Point(3,5);
        p.Translate(-1,3);
      </code>
      results in <c>p</c>'s having the value (2,8).
      </example>
      </summary>
      <param><c>xor</c> is the relative x-offset.</param>
      <param><c>yor</c> is the relative y-offset.</param>
      <see cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>
    </member>
  </members>
</doc>

```

```

<member name="M:Graphics.Point.Equals(System.Object)">
    <summary>This method determines whether two Points have the same
    location.</summary>
    <param><c>o</c> is the object to be compared to the current
    object.
    </param>
    <returns>True if the Points have the same location and they have
    the exact same type; otherwise, false.</returns>
    <seealso
        cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
    <seealso
        cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
</member>

<member name="M:Graphics.Point.ToString">
    <summary>Report a point's location as a string.</summary>
    <returns>A string representing a point's location, in the form
    (x,y),
    without any leading, trailing, or embedded whitespace.</returns>
</member>

<member
name="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)">
    <summary>This operator determines whether two Points have the
    same
    location.</summary>
    <param><c>p1</c> is the first Point to be compared.</param>
    <param><c>p2</c> is the second Point to be compared.</param>
    <returns>True if the Points have the same location and they have
    the exact same type; otherwise, false.</returns>
    <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
    <seealso
        cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
</member>

<member
name="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)">
    <summary>This operator determines whether two Points have the
    same
    location.</summary>
    <param><c>p1</c> is the first Point to be compared.</param>
    <param><c>p2</c> is the second Point to be compared.</param>
    <returns>True if the Points do not have the same location and
    the
    exact same type; otherwise, false.</returns>
    <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
    <seealso
        cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
</member>

<member name="M:Graphics.Point.Main">
    <summary>This is the entry point of the Point class testing
    program.
    <para>This program tests each method and operator, and
    is intended to be run after any non-trivial maintenance has
    been performed on the Point class.</para></summary>
</member>

<member name="P:Graphics.Point.X">
    <value>Property <c>X</c> represents the point's
    x-coordinate.</value>
</member>

```

```
<member name="P:Graphics.Point.Y">
    <value>Property <c>Y</c> represents the point's
    y-coordinate.</value>
</member>
</members>
</doc>
```





# B. 文法

ここでは、本文中に見られる字句文法と構文文法、および安全でないコードのための文法の概要を示します。文法は、本文中に出現するのと同じ順序で紹介します。

## B.1 字句文法

```

input:
  input-sectionopt

input-section:
  input-section-part
  input-section input-section-part

input-section-part:
  input-elementsopt new-line
  pp-directive

input-elements:
  input-element
  input-elements input-element

input-element:
  whitespace
  comment
  token

```

### B.1.1 行末記号

```

new-line:
  Carriage return character (U+000D)
  Line feed character (U+000A)
  Carriage return character (U+000D) followed by line feed character (U+000A)
  Next line character (U+0085)
  Line separator character (U+2028)
  Paragraph separator character (U+2029)

```

### B.1.2 コメント

```

comment:
  single-line-comment
  delimited-comment

single-line-comment:
  // input-charactersopt

input-characters:
  input-character
  input-characters input-character

input-character:
  Any Unicode character except a new-line-character

```

*new-line-character:*

- Carriage return character (U+000D)
- Line feed character (U+000A)
- Next line character (U+0085)
- Line separator character (U+2028)
- Paragraph separator character (U+2029)

*delimited-comment:*

```
/* delimited-comment-textopt asterisks */
```

*delimited-comment-text:*

```
delimited-comment-section
delimited-comment-text delimited-comment-section
```

*delimited-comment-section:*

```
/ 
asterisksopt not-slash-or-asterisk
```

*asterisks:*

```
*
```

*not-slash-or-asterisk:*

```
Any Unicode character except / or *
```

### B.1.3 空白

*whitespace:*

- Any character with Unicode class Zs
- Horizontal tab character (U+0009)
- Vertical tab character (U+000B)
- Form feed character (U+000C)

### B.1.4 トークン

*token:*

- identifier*
- keyword*
- integer-literal*
- real-literal*
- character-literal*
- string-literal*
- operator-or-punctuator*

### B.1.5 Unicode 文字のエスケープ シーケンス

*unicode-escape-sequence:*

```
\u hex-digit hex-digit hex-digit hex-digit
\u hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit hex-digit
```

### B.1.6 識別子

*identifier:*

- available-identifier*
- @ *identifier-or-keyword*

*available-identifier:*

An *identifier-or-keyword* that is not a *keyword*

*identifier-or-keyword:*

*identifier-start-character* *identifier-part-characters<sub>opt</sub>*

*identifier-start-character:*

*letter-character*

\_ (the underscore character U+005F)

*identifier-part-characters:*

*identifier-part-character*

*identifier-part-characters* *identifier-part-character*

*identifier-part-character:*

*letter-character*

*decimal-digit-character*

*connecting-character*

*combining-character*

*formatting-character*

*letter-character:*

A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl

A *unicode-escape-sequence* representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl

*combining-character:*

A Unicode character of classes Mn or Mc

A *unicode-escape-sequence* representing a character of classes Mn or Mc

*decimal-digit-character:*

A Unicode character of the class Nd

A *unicode-escape-sequence* representing a character of the class Nd

*connecting-character:*

A Unicode character of the class Pc

A *unicode-escape-sequence* representing a character of the class Pc

*formatting-character:*

A Unicode character of the class Cf

A *unicode-escape-sequence* representing a character of the class Cf

### B.1.7 キーワード

<i>keyword:</i>	one of				
abstract	as	base	bool	break	
byte	case	catch	char	checked	
class	const	continue	decimal	default	
delegate	do	double	else	enum	
event	explicit	extern	false	finally	
fixed	float	for	foreach	goto	
if	implicit	in	int	interface	
internal	is	lock	long	namespace	
new	null	object	operator	out	
override	params	private	protected	public	
readonly	ref	return	sbyte	sealed	
short	sizeof	stackalloc	static	string	
struct	switch	this	throw	true	
try	typeof	uint	ulong	unchecked	
unsafe	ushort	using	virtual	void	
volatile	while				

### B.1.8 リテラル

*literal:*

- boolean-literal*
- integer-literal*
- real-literal*
- character-literal*
- string-literal*
- null-literal*

*boolean-literal:*

- true*
- false*

*integer-literal:*

- decimal-integer-literal*
- hexadecimal-integer-literal*

*decimal-integer-literal:*

- decimal-digits integer-type-suffix<sub>opt</sub>*

*decimal-digits:*

- decimal-digit*
- decimal-digits decimal-digit*

*decimal-digit:* one of

- 0 1 2 3 4 5 6 7 8 9

*integer-type-suffix:* one of

- U u L l UL U1 uL u1 LU Lu lU lu

*hexadecimal-integer-literal:*

- 0x *hex-digits integer-type-suffix<sub>opt</sub>*
- 0X *hex-digits integer-type-suffix<sub>opt</sub>*

*hex-digits:*

*hex-digit*

*hex-digits hex-digit*

*hex-digit:* one of

  0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

*real-literal:*

*decimal-digits . decimal-digits exponent-part<sub>opt</sub> real-type-suffix<sub>opt</sub>*

*. decimal-digits exponent-part<sub>opt</sub> real-type-suffix<sub>opt</sub>*

*decimal-digits exponent-part real-type-suffix<sub>opt</sub>*

*decimal-digits real-type-suffix*

*exponent-part:*

*e sign<sub>opt</sub> decimal-digits*

*E sign<sub>opt</sub> decimal-digits*

*sign:* one of

  + -

*real-type-suffix:* one of

  F f D d M m

*character-literal:*

  ' character '

*character:*

*single-character*

*simple-escape-sequence*

*hexadecimal-escape-sequence*

*unicode-escape-sequence*

*single-character:*

  Any character except ' (U+0027), \ (U+005C), and new-line-character

*simple-escape-sequence:* one of

  \' \" \\ \0 \a \b \f \n \r \t \v

*hexadecimal-escape-sequence:*

  \x *hex-digit hex-digit<sub>opt</sub> hex-digit<sub>opt</sub> hex-digit<sub>opt</sub>*

*string-literal:*

*regular-string-literal*

*verbatim-string-literal*

*regular-string-literal:*

  " *regular-string-literal-characters<sub>opt</sub>* "

*regular-string-literal-characters:*

*regular-string-literal-character*

*regular-string-literal-characters regular-string-literal-character*

*regular-string-literal-character:*

*single-regular-string-literal-character*

*simple-escape-sequence*

*hexadecimal-escape-sequence*

*unicode-escape-sequence*

*single-regular-string-literal-character:*

Any character except " (U+0022), \ (U+005C), and new-line-character

*verbatim-string-literal:*

@" verbatim-string-literal-characters<sub>opt</sub> "

*verbatim-string-literal-characters:*

verbatim-string-literal-character

verbatim-string-literal-characters verbatim-string-literal-character

*verbatim-string-literal-character:*

single-verbatim-string-literal-character

quote-escape-sequence

*single-verbatim-string-literal-character:*

any character except "

*quote-escape-sequence:*

""

*null-literal:*

null

### B.1.9 演算子と区切り記号

*operator-or-punctuator:* one of

{	}	[	]	(	)	.	,	:	;
+	-	*	/	%	&		^	!	~
=	<	>	?	??	::	++	--	&&	
->	==	!=	<=	>=	+=	-=	*=	/=	%=
&=	=	^=	<<	<<=	=>				

*right-shift:*

>/>

*right-shift-assignment:*

>/>=

### B.1.10 プリプロセッサ ディレクティブ

*pp-directive:*

pp-declaration

pp-conditional

pp-line

pp-diagnostic

pp-region

pp-pragma

*conditional-symbol:*

Any identifier-or-keyword except true or false

*pp-expression:*

whitespace<sub>opt</sub> pp-or-expression whitespace<sub>opt</sub>

*pp-or-expression:*

pp-and-expression

pp-or-expression whitespace<sub>opt</sub> || whitespace<sub>opt</sub> pp-and-expression

*pp-and-expression:*  
  *pp-equality-expression*  
  *pp-and-expression whitespace<sub>opt</sub> && whitespace<sub>opt</sub> pp-equality-expression*

*pp-equality-expression:*  
  *pp-unary-expression*  
  *pp-equality-expression whitespace<sub>opt</sub> == whitespace<sub>opt</sub> pp-unary-expression*  
  *pp-equality-expression whitespace<sub>opt</sub> != whitespace<sub>opt</sub> pp-unary-expression*

*pp-unary-expression:*  
  *pp-primary-expression*  
  *! whitespace<sub>opt</sub> pp-unary-expression*

*pp-primary-expression:*  
  *true*  
  *false*  
  *conditional-symbol*  
  *( whitespace<sub>opt</sub> pp-expression whitespace<sub>opt</sub> )*

*pp-declaration:*  
  *whitespace<sub>opt</sub> # whitespace<sub>opt</sub> define whitespace conditional-symbol pp-new-line*  
  *whitespace<sub>opt</sub> # whitespace<sub>opt</sub> undef whitespace conditional-symbol pp-new-line*

*pp-new-line:*  
  *whitespace<sub>opt</sub> single-line-comment<sub>opt</sub> new-line*

*pp-conditional:*  
  *pp-if-section pp-elif-sections<sub>opt</sub> pp-else-section<sub>opt</sub> pp-endif*

*pp-if-section:*  
  *whitespace<sub>opt</sub> # whitespace<sub>opt</sub> if whitespace pp-expression pp-new-line conditional-section<sub>opt</sub>*

*pp-elif-sections:*  
  *pp-elif-section*  
  *pp-elif-sections pp-elif-section*

*pp-elif-section:*  
  *whitespace<sub>opt</sub> # whitespace<sub>opt</sub> elif whitespace pp-expression pp-new-line conditional-section<sub>opt</sub>*

*pp-else-section:*  
  *whitespace<sub>opt</sub> # whitespace<sub>opt</sub> else pp-new-line conditional-section<sub>opt</sub>*

*pp-endif:*  
  *whitespace<sub>opt</sub> # whitespace<sub>opt</sub> endif pp-new-line*

*conditional-section:*  
  *input-section*  
  *skipped-section*

*skipped-section:*  
  *skipped-section-part*  
  *skipped-section skipped-section-part*

*skipped-section-part:*  
  *skipped-characters<sub>opt</sub> new-line*  
  *pp-directive*

*skipped-characters:*

*whitespace<sub>opt</sub>* *not-number-sign* *input-characters<sub>opt</sub>*

*not-number-sign:*

Any *input-character* except #

*pp-diagnostic:*

*whitespace<sub>opt</sub>* # *whitespace<sub>opt</sub>* **error** *pp-message*

*whitespace<sub>opt</sub>* # *whitespace<sub>opt</sub>* **warning** *pp-message*

*pp-message:*

**new-line**

*whitespace* *input-characters<sub>opt</sub>* **new-line**

*pp-region:*

**pp-start-region** *conditional-section<sub>opt</sub>* **pp-end-region**

*pp-start-region:*

*whitespace<sub>opt</sub>* # *whitespace<sub>opt</sub>* **region** *pp-message*

*pp-end-region:*

*whitespace<sub>opt</sub>* # *whitespace<sub>opt</sub>* **endregion** *pp-message*

*pp-line:*

*whitespace<sub>opt</sub>* # *whitespace<sub>opt</sub>* **line** *whitespace* *line-indicator* *pp-new-line*

*line-indicator:*

*decimal-digits* *whitespace* *file-name*

*decimal-digits*

**default**

**hidden**

*file-name:*

" *file-name-characters* "

*file-name-characters:*

*file-name-character*

*file-name-characters* *file-name-character*

*file-name-character:*

Any *input-character* except "

*pp-pragma:*

*whitespace<sub>opt</sub>* # *whitespace<sub>opt</sub>* **pragma** *whitespace* *pragma-body* *pp-new-line*

*pragma-body:*

*pragma-warning-body*

*pragma-warning-body:*

**warning** *whitespace* *warning-action*

**warning** *whitespace* *warning-action* *whitespace* *warning-list*

*warning-action:*

**disable**

**restore**

*warning-list:*

*decimal-digits*

*warning-list* *whitespace<sub>opt</sub>* , *whitespace<sub>opt</sub>* *decimal-digits*

## B.2 構文文法

### B.2.1 基本概念

```

namespace-name:
    namespace-or-type-name

type-name:
    namespace-or-type-name

namespace-or-type-name:
    identifier type-argument-listopt
    namespace-or-type-name . identifier type-argument-listopt
    qualified-alias-member

```

### B.2.2 型

```

type:
    value-type
    reference-type
    type-parameter

```

```

value-type:
    struct-type
    enum-type

```

```

struct-type:
    type-name
    simple-type
    nullable-type

```

```

simple-type:
    numeric-type
    bool

```

```

numeric-type:
    integral-type
    floating-point-type
    decimal

```

```

integral-type:
    sbyte
    byte
    short
    ushort
    int
    uint
    long
    ulong
    char

```

```

floating-point-type:
    float
    double

```

```

nullable-type:
    non-nullable-value-type ?

```

```

non-nullable-value-type:
  type

enum-type:
  type-name

reference-type:
  class-type
  interface-type
  array-type
  delegate-type

class-type:
  type-name
  object
  dynamic
  string

interface-type:
  type-name

rank-specifiers:
  rank-specifier
  rank-specifiers rank-specifier

rank-specifier:
  [ dim-separatorsopt ]

dim-separators:
  ,
  dim-separators ,

delegate-type:
  type-name

type-argument-list:
  < type-arguments >

type-arguments:
  type-argument
  type-arguments , type-argument

type-argument:
  type

type-parameter:
  identifier

```

### B.2.3 変数

*variable-reference*:

*expression*

### B.2.4 式

*argument-list*:

*argument*

*argument-list* , *argument*

## C# LANGUAGE SPECIFICATION

*argument:*  
    *argument-name<sub>opt</sub>* *argument-value*

*argument-name:*  
    *identifier* :

*argument-value:*  
    *expression*  
    **ref** *variable-reference*  
    **out** *variable-reference*

*primary-expression:*  
    *primary-no-array-creation-expression*  
    *array-creation-expression*

*primary-no-array-creation-expression:*  
    *literal*  
    *simple-name*  
    *parenthesized-expression*  
    *member-access*  
    *invocation-expression*  
    *element-access*  
    *this-access*  
    *base-access*  
    *post-increment-expression*  
    *post-decrement-expression*  
    *object-creation-expression*  
    *delegate-creation-expression*  
    *anonymous-object-creation-expression*  
    *typeof-expression*  
    *checked-expression*  
    *unchecked-expression*  
    *default-value-expression*  
    *anonymous-method-expression*

*simple-name:*  
    *identifier* *type-argument-list<sub>opt</sub>*

*parenthesized-expression:*  
    (*expression*)

*member-access:*  
    *primary-expression* . *identifier* *type-argument-list<sub>opt</sub>*  
    *predefined-type* . *identifier* *type-argument-list<sub>opt</sub>*  
    *qualified-alias-member* . *identifier*

*predefined-type:* one of  
    bool       byte       char       decimal    double     float       int       long  
    object     sbyte     short     string      uint       ulong     ushort

*invocation-expression:*  
    *primary-expression* (*argument-list<sub>opt</sub>*)

*element-access:*  
    *primary-no-array-creation-expression* [ *argument-list* ]

*this-access:*  
  *this*

*base-access:*  
  *base . identifier*  
  *base [ argument-list ]*

*post-increment-expression:*  
  *primary-expression ++*

*post-decrement-expression:*  
  *primary-expression --*

*object-creation-expression:*  
  *new type ( argument-list<sub>opt</sub> ) object-or-collection-initializer<sub>opt</sub>*  
  *new type object-or-collection-initializer*

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

*object-initializer:*  
  { *member-initializer-list<sub>opt</sub>* }  
  { *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*

*collection-initializer:*  
  { *element-initializer-list* }  
  { *element-initializer-list* , }

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

*element-initializer:*  
  *non-assignment-expression*  
  { *expression-list* }

*expression-list:*  
  *expression*  
  *expression-list , expression*

*array-creation-expression:*  
  *new non-array-type [ expression-list ] rank-specifiers<sub>opt</sub> array-initializer<sub>opt</sub>*  
  *new array-type array-initializer*  
  *new rank-specifier array-initializer*

*delegate-creation-expression:*  
  *new delegate-type ( expression )*

## C# LANGUAGE SPECIFICATION

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

*anonymous-object-initializer:*  
    { *member-declarator-list<sub>opt</sub>* }  
    { *member-declarator-list* , *member-declarator-list* }

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

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

*typeof-expression:*  
    typeof ( *type* )  
    typeof ( *unbound-type-name* )  
    typeof ( void )

*unbound-type-name:*  
    *identifier generic-dimension-specifier<sub>opt</sub>*  
    *identifier* :: *identifier generic-dimension-specifier<sub>opt</sub>*  
    *unbound-type-name* . *identifier generic-dimension-specifier<sub>opt</sub>*

*generic-dimension-specifier:*  
    < *commas<sub>opt</sub>* >

*commas:*  
    ,  
    *commas* ,

*checked-expression:*  
    checked ( *expression* )

*unchecked-expression:*  
    unchecked ( *expression* )

*default-value-expression:*  
    default ( *type* )

*unary-expression:*  
    *primary-expression*  
    + *unary-expression*  
    - *unary-expression*  
    ! *unary-expression*  
    ~ *unary-expression*  
    *pre-increment-expression*  
    *pre-decrement-expression*  
    *cast-expression*

*pre-increment-expression:*  
    ++ *unary-expression*

*pre-decrement-expression:*  
    -- *unary-expression*

*cast-expression:*  
 ( type ) unary-expression

*multiplicative-expression:*  
 unary-expression  
 multiplicative-expression \* unary-expression  
 multiplicative-expression / unary-expression  
 multiplicative-expression % unary-expression

*additive-expression:*  
 multiplicative-expression  
 additive-expression + multiplicative-expression  
 additive-expression - multiplicative-expression

*shift-expression:*  
 additive-expression  
 shift-expression << additive-expression  
 shift-expression right-shift additive-expression

*relational-expression:*  
 shift-expression  
 relational-expression < shift-expression  
 relational-expression > shift-expression  
 relational-expression <= shift-expression  
 relational-expression >= shift-expression  
 relational-expression is type  
 relational-expression as type

*equality-expression:*  
 relational-expression  
 equality-expression == relational-expression  
 equality-expression != relational-expression

*and-expression:*  
 equality-expression  
 and-expression & equality-expression

*exclusive-or-expression:*  
 and-expression  
 exclusive-or-expression ^ and-expression

*inclusive-or-expression:*  
 exclusive-or-expression  
 inclusive-or-expression | exclusive-or-expression

*conditional-and-expression:*  
 inclusive-or-expression  
 conditional-and-expression && inclusive-or-expression

*conditional-or-expression:*  
 conditional-and-expression  
 conditional-or-expression || conditional-and-expression

*null-coalescing-expression:*  
 conditional-or-expression  
 conditional-or-expression ?? null-coalescing-expression

## C# LANGUAGE SPECIFICATION

*conditional-expression:*  
    *null-coalescing-expression*  
    *null-coalescing-expression* ? *expression* : *expression*

*lambda-expression:*  
    *anonymous-function-signature* => *anonymous-function-body*

*anonymous-method-expression:*  
    **delegate** *explicit-anonymous-function-signature<sub>opt</sub>* *block*

*anonymous-function-signature:*  
    *explicit-anonymous-function-signature*  
    *implicit-anonymous-function-signature*

*explicit-anonymous-function-signature:*  
    ( *explicit-anonymous-function-parameter-list<sub>opt</sub>* )

*explicit-anonymous-function-parameter-list:*  
    *explicit-anonymous-function-parameter*  
    *explicit-anonymous-function-parameter-list* , *explicit-anonymous-function-parameter*

*explicit-anonymous-function-parameter:*  
    *anonymous-function-parameter-modifier<sub>opt</sub>* *type identifier*

*anonymous-function-parameter-modifier:*  
    **ref**  
    **out**

*implicit-anonymous-function-signature:*  
    ( *implicit-anonymous-function-parameter-list<sub>opt</sub>* )  
    *implicit-anonymous-function-parameter*

*implicit-anonymous-function-parameter-list:*  
    *implicit-anonymous-function-parameter*  
    *implicit-anonymous-function-parameter-list* , *implicit-anonymous-function-parameter*

*implicit-anonymous-function-parameter:*  
    *identifier*

*anonymous-function-body:*  
    *expression*  
    *block*

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

*from-clause:*  
    **from** *type<sub>opt</sub>* *identifier* **in** *expression*

*query-body:*  
    *query-body-clauses<sub>opt</sub>* *select-or-group-clause* *query-continuation<sub>opt</sub>*

*query-body-clauses:*  
    *query-body-clause*  
    *query-body-clauses* *query-body-clause*

*query-body-clause:*

- from-clause*
- let-clause*
- where-clause*
- join-clause*
- join-into-clause*
- orderby-clause*

*let-clause:*

- let** *identifier* = *expression*

*where-clause:*

- where** *boolean-expression*

*join-clause:*

- join** *type<sub>opt</sub>* *identifier* **in** *expression* **on** *expression* **equals** *expression*

*join-into-clause:*

- join** *type<sub>opt</sub>* *identifier* **in** *expression* **on** *expression* **equals** *expression* **into** *identifier*

*orderby-clause:*

- orderby** *orderings*

*orderings:*

- ordering*
- orderings* , *ordering*

*ordering:*

- expression* *ordering-direction<sub>opt</sub>*

*ordering-direction:*

- ascending**
- descending**

*select-or-group-clause:*

- select-clause*
- group-clause*

*select-clause:*

- select** *expression*

*group-clause:*

- group** *expression* **by** *expression*

*query-continuation:*

- into** *identifier* *query-body*

*assignment:*

- unary-expression* *assignment-operator* *expression*

*assignment-operator:*  
=  
+=  
-=  
\*=  
/=  
%=  
&=  
|=  
^=  
<<=  
*right-shift-assignment*

*expression:*  
non-assignment-expression  
assignment

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

constant-expression:  
expression

boolean-expression:  
expression

### B.2.5 ステートメント

*statement:*  
labeled-statement  
declaration-statement  
embedded-statement

*embedded-statement:*  
block  
empty-statement  
expression-statement  
selection-statement  
iteration-statement  
jump-statement  
try-statement  
checked-statement  
unchecked-statement  
lock-statement  
using-statement  
yield-statement

*block:*  
{ statement-list<sub>opt</sub> }

*statement-list:*  
statement  
statement-list statement

```

empty-statement:
    ;

labeled-statement:
    identifier : statement

declaration-statement:
    local-variable-declaration ;
    local-constant-declaration ;

local-variable-declaration:
    local-variable-type local-variable-declarators

local-variable-type:
    type
    var

local-variable-declarators:
    local-variable-declarator
    local-variable-declarators , local-variable-declarator

local-variable-declarator:
    identifier
    identifier = local-variable-initializer

local-variable-initializer:
    expression
    array-initializer

local-constant-declaration:
    const type constant-declarators

constant-declarators:
    constant-declarator
    constant-declarators , constant-declarator

constant-declarator:
    identifier = constant-expression

expression-statement:
    statement-expression ;

statement-expression:
    invocation-expression
    object-creation-expression
    assignment
    post-increment-expression
    post-decrement-expression
    pre-increment-expression
    pre-decrement-expression

selection-statement:
    if-statement
    switch-statement

if-statement:
    if ( boolean-expression ) embedded-statement
    if ( boolean-expression ) embedded-statement else embedded-statement

```

## C# LANGUAGE SPECIFICATION

```
switch-statement:  
    switch ( expression ) switch-block  
  
switch-block:  
    { switch-sectionsopt }  
  
switch-sections:  
    switch-section  
    switch-sections switch-section  
  
switch-section:  
    switch-labels statement-list  
  
switch-labels:  
    switch-label  
    switch-labels switch-label  
  
switch-label:  
    case constant-expression :  
    default :  
  
iteration-statement:  
    while-statement  
    do-statement  
    for-statement  
    foreach-statement  
  
while-statement:  
    while ( boolean-expression ) embedded-statement  
  
do-statement:  
    do embedded-statement while ( boolean-expression ) ;  
  
for-statement:  
    for ( for-initializeropt ; for-conditionopt ; for-iteratoropt ) embedded-statement  
  
for-initializer:  
    local-variable-declaration  
    statement-expression-list  
  
for-condition:  
    boolean-expression  
  
for-iterator:  
    statement-expression-list  
  
statement-expression-list:  
    statement-expression  
    statement-expression-list , statement-expression  
  
foreach-statement:  
    foreach ( local-variable-type identifier in expression ) embedded-statement  
  
jump-statement:  
    break-statement  
    continue-statement  
    goto-statement  
    return-statement  
    throw-statement
```

```

break-statement:
    break ;

continue-statement:
    continue ;

goto-statement:
    goto identifier ;
    goto case constant-expression ;
    goto default ;

return-statement:
    return expressionopt ;

throw-statement:
    throw expressionopt ;

try-statement:
    try block catch-clauses
    try block finally-clause
    try block catch-clauses finally-clause

catch-clauses:
    specific-catch-clauses general-catch-clauseopt
    specific-catch-clausesopt general-catch-clause

specific-catch-clauses:
    specific-catch-clause
    specific-catch-clauses specific-catch-clause

specific-catch-clause:
    catch ( class-type identifieropt ) block

general-catch-clause:
    catch block

finally-clause:
    finally block

checked-statement:
    checked block

unchecked-statement:
    unchecked block

lock-statement:
    lock ( expression ) embedded-statement

using-statement:
    using ( resource-acquisition ) embedded-statement

resource-acquisition:
    local-variable-declaration
    expression

yield-statement:
    yield return expression ;
    yield break ;

```

**B.2.6 名前空間**

```

compilation-unit:
  extern-alias-directivesopt using-directivesopt global-attributesopt
    namespace-member-declarationsopt

namespace-declaration:
  namespace qualified-identifier namespace-body ;opt

qualified-identifier:
  identifier
  qualified-identifier . identifier

namespace-body:
  { extern-alias-directivesopt using-directivesopt namespace-member-declarationsopt }

extern-alias-directives:
  extern-alias-directive
  extern-alias-directives extern-alias-directive

extern-alias-directive:
  extern alias identifier ;

using-directives:
  using-directive
  using-directives using-directive

using-directive:
  using-alias-directive
  using-namespace-directive

using-alias-directive:
  using identifier = namespace-or-type-name ;

using-namespace-directive:
  using namespace-name ;

namespace-member-declarations:
  namespace-member-declaration
  namespace-member-declarations namespace-member-declaration

namespace-member-declaration:
  namespace-declaration
  type-declaration

type-declaration:
  class-declaration
  struct-declaration
  interface-declaration
  enum-declaration
  delegate-declaration

qualified-alias-member:
  identifier :: identifier type-argument-listopt

```

**B.2.7 クラス**

```

class-declaration:
  attributesopt class-modifiersopt partialopt class identifier type-parameter-listopt
    class-baseopt type-parameter-constraints-clausesopt class-body ;opt

```

```

class-modifiers:
  class-modifier
  class-modifiers class-modifier

class-modifier:
  new
  public
  protected
  internal
  private
  abstract
  sealed
  static

type-parameter-list:
  < type-parameters >

type-parameters:
  attributesopt type-parameter
  type-parameters , attributesopt type-parameter

type-parameter:
  identifier

class-base:
  : class-type
  : interface-type-list
  : class-type , interface-type-list

interface-type-list:
  interface-type
  interface-type-list , interface-type

type-parameter-constraints-clauses:
  type-parameter-constraints-clause
  type-parameter-constraints-clauses type-parameter-constraints-clause

type-parameter-constraints-clause:
  where type-parameter : type-parameter-constraints

type-parameter-constraints:
  primary-constraint
  secondary-constraints
  constructor-constraint
  primary-constraint , secondary-constraints
  primary-constraint , constructor-constraint
  secondary-constraints , constructor-constraint
  primary-constraint , secondary-constraints , constructor-constraint

primary-constraint:
  class-type
  class
  struct

```

*secondary-constraints:*  
    *interface-type*  
    *type-parameter*  
    *secondary-constraints* , *interface-type*  
    *secondary-constraints* , *type-parameter*

*constructor-constraint:*  
    *new* ( )

*class-body:*  
    { *class-member-declarations<sub>opt</sub>* }

*class-member-declarations:*  
    *class-member-declaration*  
    *class-member-declarations* *class-member-declaration*

*class-member-declaration:*  
    *constant-declaration*  
    *field-declaration*  
    *method-declaration*  
    *property-declaration*  
    *event-declaration*  
    *indexer-declaration*  
    *operator-declaration*  
    *constructor-declaration*  
    *destructor-declaration*  
    *static-constructor-declaration*  
    *type-declaration*

*constant-declaration:*  
    *attributes<sub>opt</sub>* *constant-modifiers<sub>opt</sub>* **const** *type* *constant-declarators* ;

*constant-modifiers:*  
    *constant-modifier*  
    *constant-modifiers* *constant-modifier*

*constant-modifier:*  
    *new*  
    *public*  
    *protected*  
    *internal*  
    *private*

*constant-declarators:*  
    *constant-declarator*  
    *constant-declarators* , *constant-declarator*

*constant-declarator:*  
    *identifier* = *constant-expression*

*field-declaration:*  
    *attributes<sub>opt</sub>* *field-modifiers<sub>opt</sub>* *type* *variable-declarators* ;

*field-modifiers:*  
    *field-modifier*  
    *field-modifiers* *field-modifier*

*field-modifier:*

new  
public  
protected  
internal  
private  
static  
readonly  
volatile

*variable-declarators:*

*variable-declarator*  
*variable-declarators , variable-declarator*

*variable-declarator:*

*identifier*  
*identifier = variable-initializer*

*variable-initializer:*

*expression*  
*array-initializer*

*method-declaration:*

*method-header method-body*

*method-header:*

*attributes<sub>opt</sub> method-modifiers<sub>opt</sub> partial<sub>opt</sub> return-type member-name type-parameter-list<sub>opt</sub>*  
*( formal-parameter-list<sub>opt</sub> ) type-parameter-constraints-clauses<sub>opt</sub>*

*method-modifiers:*

*method-modifier*  
*method-modifiers method-modifier*

*method-modifier:*

new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern

*return-type:*

*type*  
*void*

*member-name:*

*identifier*  
*interface-type . identifier*

*method-body:*

*block*  
;

*formal-parameter-list:*  
    *fixed-parameters*  
    *fixed-parameters* , *parameter-array*  
    *parameter-array*

*fixed-parameters:*  
    *fixed-parameter*  
    *fixed-parameters* , *fixed-parameter*

*fixed-parameter:*  
    *attributes<sub>opt</sub>* *parameter-modifier<sub>opt</sub>* *type* *identifier* *default-argument<sub>opt</sub>*

*default-argument:*  
    = *expression*

*parameter-modifier:*  
    *ref*  
    *out*  
    *this*

*parameter-array:*  
    *attributes<sub>opt</sub>* *params* *array-type* *identifier*

*property-declaration:*  
    *attributes<sub>opt</sub>* *property-modifiers<sub>opt</sub>* *type* *member-name* { *accessor-declarations* }

*property-modifiers:*  
    *property-modifier*  
    *property-modifiers* *property-modifier*

*property-modifier:*  
    *new*  
    *public*  
    *protected*  
    *internal*  
    *private*  
    *static*  
    *virtual*  
    *sealed*  
    *override*  
    *abstract*  
    *extern*

*member-name:*  
    *identifier*  
    *interface-type* . *identifier*

*accessor-declarations:*  
    *get-accessor-declaration* *set-accessor-declaration<sub>opt</sub>*  
    *set-accessor-declaration* *get-accessor-declaration<sub>opt</sub>*

*get-accessor-declaration:*  
    *attributes<sub>opt</sub>* *accessor-modifier<sub>opt</sub>* **get** *accessor-body*

*set-accessor-declaration:*  
    *attributes<sub>opt</sub>* *accessor-modifier<sub>opt</sub>* **set** *accessor-body*

```

accessor-modifier:
protected
internal
private
protected internal
internal protected

accessor-body:
block
;

event-declaration:
attributesopt event-modifiersopt event type variable-declarators ;
attributesopt event-modifiersopt event type member-name { event-accessor-declarations }

event-modifiers:
event-modifier
event-modifiers event-modifier

event-modifier:
new
public
protected
internal
private
static
virtual
sealed
override
abstract
extern

event-accessor-declarations:
add-accessor-declaration remove-accessor-declaration
remove-accessor-declaration add-accessor-declaration

add-accessor-declaration:
attributesopt add block

remove-accessor-declaration:
attributesopt remove block

indexer-declaration:
attributesopt indexer-modifiersopt indexer-declarator { accessor-declarations }

indexer-modifiers:
indexer-modifier
indexer-modifiers indexer-modifier

```

*indexer-modifier:*

new  
public  
protected  
internal  
private  
virtual  
sealed  
override  
abstract  
extern

*indexer-declarator:*

type this [ formal-parameter-list ]  
type interface-type . this [ formal-parameter-list ]

*operator-declaration:*

attributes<sub>opt</sub> operator-modifiers operator-declarator operator-body

*operator-modifiers:*

operator-modifier  
operator-modifiers operator-modifier

*operator-modifier:*

public  
static  
extern

*operator-declarator:*

unary-operator-declarator  
binary-operator-declarator  
conversion-operator-declarator

*unary-operator-declarator:*

type operator overloadable-unary-operator ( type identifier )

*overloadable-unary-operator:* one of

+ - ! ~ ++ -- true false

*binary-operator-declarator:*

type operator overloadable-binary-operator ( type identifier , type identifier )

*overloadable-binary-operator:*

```
+  
-  
*  
/  
%  
&  
|  
^  
<<  
right-shift  
==  
!=  
>  
<  
>=  
=<
```

*conversion-operator-declarator:*

```
implicit operator type ( type identifier )  
explicit operator type ( type identifier )
```

*operator-body:*

```
block  
;
```

*constructor-declaration:*

```
attributesopt constructor-modifiersopt constructor-declarator constructor-body
```

*constructor-modifiers:*

```
constructor-modifier  
constructor-modifiers constructor-modifier
```

*constructor-modifier:*

```
public  
protected  
internal  
private  
extern
```

*constructor-declarator:*

```
identifier ( formal-parameter-listopt ) constructor-initializeropt
```

*constructor-initializer:*

```
: base ( argument-listopt )  
: this ( argument-listopt )
```

*constructor-body:*

```
block  
;
```

*static-constructor-declaration:*

```
attributesopt static-constructor-modifiers identifier ( ) static-constructor-body
```

*static-constructor-modifiers:*

```
externopt static  
static externopt
```

```
static-constructor-body:  
    block  
    ;  
  
destructor-declaration:  
    attributesopt externopt ~ identifier ( ) destructor-body  
  
destructor-body:  
    block  
    ;
```

## B.2.8 構造体

```
struct-declaration:  
    attributesopt struct-modifiersopt partialopt struct identifier type-parameter-listopt  
        struct-interfacesopt type-parameter-constraints-clausesopt struct-body ;opt  
  
struct-modifiers:  
    struct-modifier  
    struct-modifiers struct-modifier  
  
struct-modifier:  
    new  
    public  
    protected  
    internal  
    private  
  
struct-interfaces:  
    : interface-type-list  
  
struct-body:  
    { struct-member-declarationsopt }  
  
struct-member-declarations:  
    struct-member-declaration  
    struct-member-declarations struct-member-declaration  
  
struct-member-declaration:  
    constant-declaration  
    field-declaration  
    method-declaration  
    property-declaration  
    event-declaration  
    indexer-declaration  
    operator-declaration  
    constructor-declaration  
    static-constructor-declaration  
    type-declaration
```

## B.2.9 配列

```
array-type:  
    non-array-type rank-specifiers  
  
non-array-type:  
    type
```

```

rank-specifiers:
  rank-specifier
  rank-specifiers rank-specifier

rank-specifier:
  [ dim-separatorsopt ]

dim-separators:
  ,
  dim-separators ,

array-initializer:
  { variable-initializer-listopt }
  { variable-initializer-list , }

variable-initializer-list:
  variable-initializer
  variable-initializer-list , variable-initializer

variable-initializer:
  expression
  array-initializer

```

## B.2.10 インターフェイス

```

interface-declaration:
  attributesopt interface-modifiersopt partialopt interface
    identifier variant-type-parameter-listopt interface-baseopt
    type-parameter-constraints-clausesopt interface-body ;opt

interface-modifiers:
  interface-modifier
  interface-modifiers interface-modifier

interface-modifier:
  new
  public
  protected
  internal
  private

variant-type-parameter-list:
  < variant-type-parameters >

variant-type-parameters:
  attributesopt variance-annotationopt type-parameter
  variant-type-parameters , attributesopt variance-annotationopt type-parameter

variance-annotation:
  in
  out

interface-base:
  : interface-type-list

interface-body:
  { interface-member-declarationsopt }

```

```
interface-member-declarations:
    interface-member-declaration
    interface-member-declarations  interface-member-declaration

interface-member-declaration:
    interface-method-declaration
    interface-property-declaration
    interface-event-declaration
    interface-indexer-declaration

interface-method-declaration:
    attributesopt newopt return-type identifier type-parameter-list
        ( formal-parameter-listopt ) type-parameter-constraints-clausesopt ;

interface-property-declaration:
    attributesopt newopt type identifier { interface-accessors }

interface-accessors:
    attributesopt get ;
    attributesopt set ;
    attributesopt get ; attributesopt set ;
    attributesopt set ; attributesopt get ;

interface-event-declaration:
    attributesopt newopt event type identifier ;

interface-indexer-declaration:
    attributesopt newopt type this [ formal-parameter-list ] { interface-accessors }
```

## B.2.11 列舉型

```
enum-declaration:
    attributesopt enum-modifiersopt enum identifier enum-baseopt enum-body ;opt

enum-base:
    : integral-type

enum-body:
    { enum-member-declarationsopt }
    { enum-member-declarations , }

enum-modifiers:
    enum-modifier
    enum-modifiers enum-modifier

enum-modifier:
    new
    public
    protected
    internal
    private

enum-member-declarations:
    enum-member-declaration
    enum-member-declarations , enum-member-declaration
```

*enum-member-declaration:*  
*attributes<sub>opt</sub>* *identifier*  
*attributes<sub>opt</sub>* *identifier* = *constant-expression*

### B.2.12 デリゲート

*delegate-declaration:*  
*attributes<sub>opt</sub>* *delegate-modifiers<sub>opt</sub>* **delegate** *return-type*  
*identifier* *variant-type-parameter-list<sub>opt</sub>*  
*( formal-parameter-list<sub>opt</sub> ) type-parameter-constraints-clauses<sub>opt</sub>* ;  
*delegate-modifiers:*  
*delegate-modifier*  
*delegate-modifiers delegate-modifier*  
*delegate-modifier:*  
**new**  
**public**  
**protected**  
**internal**  
**private**

### B.2.13 属性

*global-attributes:*  
*global-attribute-sections*  
*global-attribute-sections:*  
*global-attribute-section*  
*global-attribute-sections global-attribute-section*  
*global-attribute-section:*  
*[ global-attribute-target-specifier attribute-list ]*  
*[ global-attribute-target-specifier attribute-list , ]*  
*global-attribute-target-specifier:*  
*global-attribute-target :*  
*global-attribute-target:*  
**assembly**  
**module**  
*attributes:*  
*attribute-sections*  
*attribute-sections:*  
*attribute-section*  
*attribute-sections attribute-section*  
*attribute-section:*  
*[ attribute-target-specifier<sub>opt</sub> attribute-list ]*  
*[ attribute-target-specifier<sub>opt</sub> attribute-list , ]*  
*attribute-target-specifier:*  
*attribute-target :*

```
attribute-target:  
    field  
    event  
    method  
    param  
    property  
    return  
    type  
  
attribute-list:  
    attribute  
    attribute-list , attribute  
  
attribute:  
    attribute-name attribute-argumentsopt  
  
attribute-name:  
    type-name  
  
attribute-arguments:  
    ( positional-argument-listopt )  
    ( positional-argument-list , named-argument-list )  
    ( named-argument-list )  
  
positional-argument-list:  
    positional-argument  
    positional-argument-list , positional-argument  
  
positional-argument:  
    argument-nameopt attribute-argument-expression  
  
named-argument-list:  
    named-argument  
    named-argument-list , named-argument  
  
named-argument:  
    identifier = attribute-argument-expression  
  
attribute-argument-expression:  
    expression
```

### B.3 安全でないコードのための文法

```
class-modifier:  
    ...  
    unsafe  
  
struct-modifier:  
    ...  
    unsafe  
  
interface-modifier:  
    ...  
    unsafe  
  
delegate-modifier:  
    ...  
    unsafe
```

*field-modifier:*

...  
**unsafe**

*method-modifier:*

...  
**unsafe**

*property-modifier:*

...  
**unsafe**

*event-modifier:*

...  
**unsafe**

*indexer-modifier:*

...  
**unsafe**

*operator-modifier:*

...  
**unsafe**

*constructor-modifier:*

...  
**unsafe**

*destructor-declaration:*

$\text{attributes}_{\text{opt}} \text{ extern}_{\text{opt}} \text{ unsafe}_{\text{opt}} \sim \text{identifier} (\ ) \text{ destructor-body}$   
 $\text{attributes}_{\text{opt}} \text{ unsafe}_{\text{opt}} \text{ extern}_{\text{opt}} \sim \text{identifier} (\ ) \text{ destructor-body}$

*static-constructor-modifiers:*

**extern**  $\text{unsafe}_{\text{opt}}$  **static**  
**unsafe**  $\text{extern}_{\text{opt}}$  **static**  
**extern**  $\text{static}$   $\text{unsafe}_{\text{opt}}$   
**unsafe**  $\text{static}$   $\text{extern}_{\text{opt}}$   
**static**  $\text{extern}_{\text{opt}}$   $\text{unsafe}_{\text{opt}}$   
**static**  $\text{unsafe}_{\text{opt}}$   $\text{extern}_{\text{opt}}$

*embedded-statement:*

...  
*unsafe-statement*  
*fixed-statement*

*unsafe-statement:*

**unsafe** *block*

*type:*

...  
*pointer-type*

*pointer-type:*

*unmanaged-type* \*  
**void** \*

*unmanaged-type:*

*type*

## C# LANGUAGE SPECIFICATION

*primary-no-array-creation-expression:*

...

*pointer-member-access*

*pointer-element-access*

*sizeof-expression*

*unary-expression:*

...

*pointer-indirection-expression*

*addressof-expression*

*pointer-indirection-expression:*

\* *unary-expression*

*pointer-member-access:*

*primary-expression* → *identifier* *type-argument-list<sub>opt</sub>*

*pointer-element-access:*

*primary-no-array-creation-expression* [ *expression* ]

*addressof-expression:*

& *unary-expression*

*sizeof-expression:*

**sizeof** ( *unmanaged-type* )

*fixed-statement:*

**fixed** ( *pointer-type* *fixed-pointer-declarators* ) *embedded-statement*

*fixed-pointer-declarators:*

*fixed-pointer-declarator*  
  *fixed-pointer-declarators* , *fixed-pointer-declarator*

*fixed-pointer-declarator:*

*identifier* = *fixed-pointer-initializer*

*fixed-pointer-initializer:*

& *variable-reference*  
  *expression*

*struct-member-declaration:*

...

*fixed-size-buffer-declaration*

*fixed-size-buffer-declaration:*

*attributes<sub>opt</sub>* *fixed-size-buffer-modifiers<sub>opt</sub>* **fixed** *buffer-element-type*  
  *fixed-size-buffer-declarators* ;

*fixed-size-buffer-modifiers:*

*fixed-size-buffer-modifier*  
  *fixed-size-buffer-modifier* *fixed-size-buffer-modifiers*

*fixed-size-buffer-modifier:*

new  
public  
protected  
internal  
private  
unsafe

*buffer-element-type:*  
    *type*

*fixed-size-buffer-declarators:*  
    *fixed-size-buffer-declarator*  
    *fixed-size-buffer-declarator fixed-size-buffer-declarators*

*fixed-size-buffer-declarator:*  
    *identifier* [ *constant-expression* ]

*local-variable-initializer:*  
    ...  
    *stackalloc-initializer*

*stackalloc-initializer:*  
    **stackalloc** *unmanaged-type* [ *expression* ]



## C. リファレンス

Unicode Consortium について。『*The Unicode Standard, Version 3.0*』。Addison-Wesley, Reading, Massachusetts, 2000, ISBN 0-201-616335-5.

IEEE について。『*IEEE Standard for Binary Floating-Point Arithmetic*』。『ANSI/IEEE Standard 754-1985』。<http://www.ieee.org> から入手できます。

ISO/IEC について。C++. 『ANSI/ISO/IEC 14882:1998』。